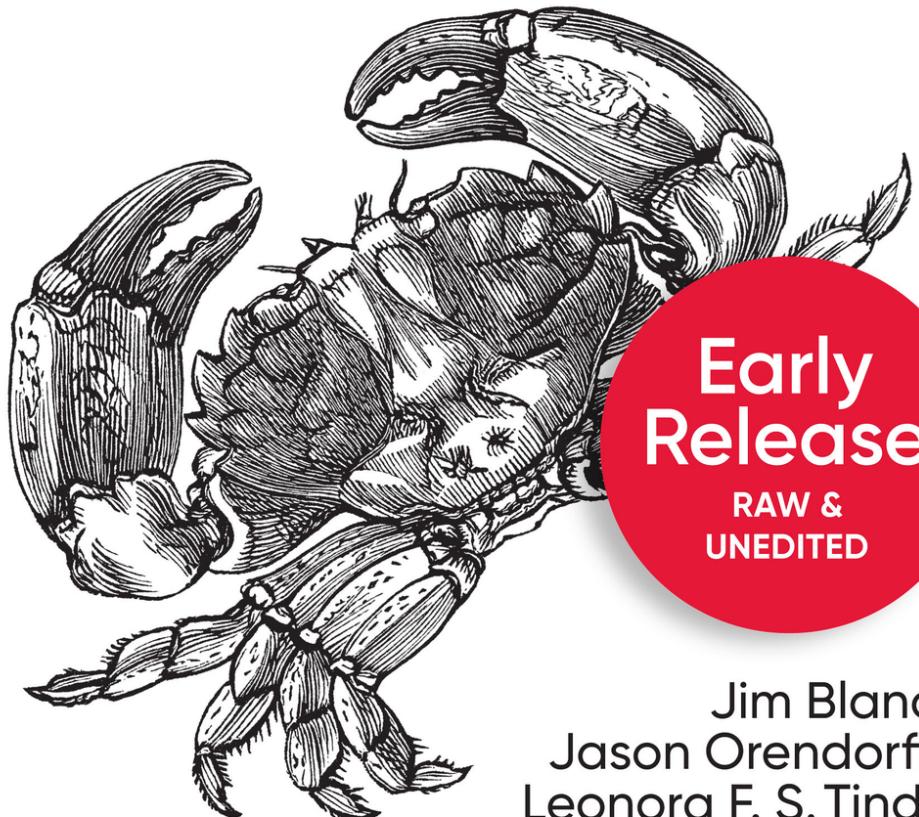


O'REILLY®

Second
Edition

Programming Rust

Fast, Safe Systems Development



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

Programming Rust

SECOND EDITION

Fast, Safe Systems Development

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

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

Programming Rust

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

Copyright © 2020 Jim Blandy, Leonora F.S. Tindall, Jason Orendorff. All rights reserved.

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://oreilly.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Acquisitions Editor: Suzanne McQuade

Developmental Editor: Jeff Bleiel

Production Editor: Kristen Brown

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Rebecca Demarest

Revision History for the Early Release

- 2020-02-03: First Early Release
- 2020-04-23: Second Early Release
- 2020-06-11: Third Early Release
- 2020-08-26: Fourth Early Release

- 2020-10-16: Fifth Early Release
- 2020-12-07: Sixth Early Release

See <http://oreilly.com/catalog/errata.csp?isbn=9781492052593> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc.
Programming Rust, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

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

978-1-492-05259-3

[LSI]

Chapter 1. Why Rust?

A NOTE FOR EARLY RELEASE READERS

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

This will be the 1st chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at jimb@red-bean.com.

In certain contexts—for example the context Rust is targeting—being 10x or even 2x faster than the competition is a make-or-break thing. It decides the fate of a system in the market, as much as it would in the hardware market.

—Graydon Hoare

*All computers are now parallel...
Parallel programming **is** programming.*

—Michael McCool et al., *Structured Parallel Programming*

*TrueType parser flaw
used by nation-state attacker for surveillance;
all software is security-sensitive.*

—Andy Wingo

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 problems in particular have proven difficult to crack:

- It's difficult to write secure code. It's especially difficult to manage memory correctly in C and C++. Users have been suffering with the consequences for decades, in the form of security holes dating back at least as far as the 1988 Morris worm.
- It's very difficult to write multithreaded code, which is the only way to exploit the abilities of modern machines. Even experienced programmers approach threaded code with caution: concurrency can introduce broad new classes of bugs and make ordinary bugs much harder to reproduce.

Enter Rust: a safe, concurrent language with the performance of C and C++.

Rust is a new systems programming language developed by Mozilla and a community of contributors. Like C and C++, Rust gives developers 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 trustworthy concurrency.

The key to meeting all these promises is Rust's novel system of *ownership*, *moves*, and *borrowing*, 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. Moves transfer values from one owner to another, and borrowing lets code use a value temporarily without affecting its ownership. Since many

programmers will have never encountered these features in this form before, we explain them in detail in Chapters 3 and 4.

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 given it to another thread; Rust checks that you don't. Rust is able to prevent data races at compile time.

Rust is not really an object-oriented language, although it has some object-oriented characteristics. Rust is not a functional language, although it does tend to make the influences on a computation's result more explicit, as functional languages do. Rust resembles C and C++ to an extent, but many idioms from those languages don't apply, so typical Rust code does not deeply resemble C or C++ code. It's probably best to reserve judgement about what sort of language Rust is, and see what you think once you've become comfortable with the language.

To get feedback on the design in a real-world setting, Mozilla has developed Servo, a new web browser engine, in Rust. Servo's needs and Rust's goals are well matched: a browser must perform well and handle untrusted data securely. Servo uses Rust's safe concurrency to put the full machine to work on tasks that would be impractical to parallelize in C or C++. In fact, Servo and Rust have grown up together, with Servo using the latest new language features, and Rust evolving based on feedback from Servo's developers.

Type Safety

Rust is a type-safe language. 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:

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. When we ran this program on Jim's laptop, it produced the following output:

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

Then it crashed. Jim's laptop doesn't even have a `.netrc` file. If you try it yourself, it will probably do something entirely different.

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 programmer responsible for ensuring those conditions never arise in the first place.

Empirically speaking, we're not very good at that. 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. And undefined behavior often leads to exploitable security holes in practice. The Morris worm propagated itself from one machine to another using an elaboration of the technique shown before, and this kind of exploit remains in widespread use today.

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 safety checks ensure that every program is well defined, we say that language is *type safe*.

A carefully written C or C++ program might be well defined, but C and C++ are not type safe: the program shown earlier 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] = 0xfffff7b36ceb
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. Certainly, a module like `ctypes` that provides unconstrained access to the machine can introduce undefined behavior into Python, but the core language itself is type safe. Java, JavaScript, Ruby, and Haskell are similar in this way.

Note that being type safe is 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 run time, and is type safe.

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.

Rust's particular form of type safety has surprising 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 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 use concurrency without worrying that you've made your code impossible for any but the most accomplished programmers to work on.

Rust has an escape valve from the safety rules, for when you absolutely have to use a raw pointer. This is called *unsafe code*, and while most Rust programs don't need it, we'll show how to use it and how it fits into Rust's overall safety scheme in XREF HERE.

Like those of other statically typed languages, Rust's types can do much more than simply prevent undefined behavior. An accomplished Rust programmer uses types to ensure values are used not just safely but meaningfully, in a way that's consistent with the application's intent. In particular, Rust's traits and generics, described in Chapter 10, provide a succinct, flexible, and performant way to describe characteristics that a group of types has in common, and then take advantage of those commonalities.

Our aim in this book is to give you the insights you need not just to write programs in Rust, but to put the language to work ensuring that those programs are both safe and correct, and to anticipate how they will perform. In our experience, Rust is a major step forward in systems programming, and we want to help you take advantage of it.

Chapter 2. Basic Types

A NOTE FOR EARLY RELEASE READERS

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

This will be the 3rd chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at jimb@red-bean.com.

There are many, many types of books in the world, which makes good sense, because there are many, many types of people, and everybody wants to read something different.

—Lemony Snicket

To a great extent, the Rust language is designed around its types. Its memory and thread safety guarantees rest on the soundness of its type system. Its flexibility stems from its generic types and traits. And its performance arises from letting developers choose representations for their data with the right balance between flexibility and cost.

This chapter is about that third aspect: Rust’s fundamental types for representing values. These source-level types have concrete machine-level counterparts with predictable costs and performance. Although Rust doesn’t promise it will represent things exactly as you’ve requested, it takes care to deviate from your requests only when it’s a reliable improvement.

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 constructs. 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);
    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);
    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*: a single function can work on values of many different types.

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 nongeneric 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, then moving on to types that hold more data: boxes, tuples, arrays, and strings.

Here's a summary of the sorts of types you'll see in Rust. This table shows Rust's primitive types, some very common types from the standard library, and some examples of user-defined types:

Type	Description	Values
i8, i16, i32, i64, i128	Signed and unsigned integers, of given bit width	42, -5i8, 0x400u16, 0o100i16, 20_922_789_888_000u64, b'*' (u8 byte literal)
u8, u16, u32, u64, u128		
isize, usize	Signed and unsigned integers, the same size as an address on the machine (32 or 64 bits)	137, -0b0101_0010 isize, 0xffff_fc00 usize
f32, f64	IEEE floating-point numbers, single and double precision	1.61803, 3.14f32, 6.0221e23f64
bool	Boolean	true, false
char	Unicode character, 32 bits wide	'*', '\n', '字', '\x7f', '\u{CA0}'
(char, u8, i32)	Tuple: mixed types allowed	('%', 0x7f, -1)
()	“unit” (empty tuple)	()
struct S { x: f32, y: f32 }	Named-field struct	S { x: 120.0, y: 20.0 }
struct T(i32, char);	Tuple-like struct	T(120, 'X')
struct E;	Unit-like struct; has no fields	E
enum Attend { OnTime, Late(u32) }	Enumeration, algebraic data type	Attend::Late(5), Attend::OnTime
Box<Attend>	Box: owning pointer to value in heap	Box::new(Late(15))
&i32, &mut i32	Shared and mutable references: nonowning pointers that must not outlive their referent	&s.y, &mut v
String	UTF-8 string, dynamically sized	"ラーメン: ramen".to_string()

Type	Description	Values
<code>&str</code>	Reference to <code>str</code> : nonowning pointer to UTF-8 text	"そば: soba", <code>&s[0..12]</code>
<code>[f64; 4], [u8; 256]</code>	Array, fixed length; elements all of same type	<code>[1.0, 0.0, 0.0, 1.0], [b' ';</code> <code>256]</code>
<code>Vec<f64></code>	Vector, varying length; elements all of same type	<code>vec![0.367, 2.718, 7.389]</code>
<code>&[u8], &mut [u8]</code>	Reference to slice: reference to a portion of an array or vector, comprising pointer and length	<code>&v[10..20], &mut a[..]</code>
<code>Option<&str></code>	Optional value: either <code>None</code> (absent) or <code>Some(v)</code> (present, with value v)	<code>Some("Dr."), None</code>
<code>Result<u64, Error></code>	Result of operation that may fail: either a success value <code>Ok(v)</code> , or an error <code>Error(e)</code>	<code>Ok(4096), Err(Error::last_os_error())</code>
<code>&dyn Any, &mut dyn Read</code>	Trait object: reference to any value that implements a given set of methods	<code>value as &dyn Any,</code> <code>&mut file as &mut dyn Read</code>
<code>fn(&str) -> bool</code>	Pointer to function	<code>str::is_empty</code>
(Closure types have no written form)	Closure	<code> a, b { a*a + b*b }</code>

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

- We give `struct` types their own chapter, [Chapter 8](#).
- We give enumerated types their own chapter, [Chapter 9](#).
- We describe trait objects in [Chapter 10](#).
- We describe the essentials of `String` and `&str` here, but provide more detail in XREF HERE.
- We cover function and closure types in XREF HERE.

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:

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

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

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

Integer Types

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

Type	Range
<code>u8</code>	0 to $2^8 - 1$ (0 to 255)
<code>u16</code>	0 to $2^{16} - 1$ (0 to 65,535)
<code>u32</code>	0 to $2^{32} - 1$ (0 to 4,294,967,295)
<code>u64</code>	0 to $2^{64} - 1$ (0 to 18,446,744,073,709,551,615, or 18 quintillion)
<code>u128</code>	0 to $2^{128} - 1$ (0 to around 3.4×10^{38})
<code>usize</code>	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
<code>i8</code>	-2^7 to $2^7 - 1$ (-128 to 127)
<code>i16</code>	-2^{15} to $2^{15} - 1$ (-32,768 to 32,767)
<code>i32</code>	-2^{31} to $2^{31} - 1$ (-2,147,483,648 to 2,147,483,647)
<code>i64</code>	-2^{63} to $2^{63} - 1$ (-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807)
<code>i128</code>	-2^{127} to $2^{127} - 1$ (roughly -1.7×10^{38} to $+1.7 \times 10^{38}$)
<code>isize</code>	Either -2^{31} to $2^{31} - 1$, or -2^{63} to $2^{63} - 1$

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

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 “[Characters](#)”.

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

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

```
let mut i = 1;
loop {
    i *= 10; // 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`. If an integer literal lacks a type suffix, Rust puts off determining its type until it finds the value being used in a way that pins it down: stored in a variable of a particular type, passed to a function that expects a particular type, compared with another value of a particular type, or something like that. In the end, if multiple types could work, 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, so you can break hexadecimal or binary numbers into groups of four digits rather than three, as in `0xffff_ffff`, or set off the type suffix from the digits, as in `127_u8`.

Some examples of integer literals:

Literal	Type	Decimal value
<code>116i8</code>	<code>i8</code>	116
<code>0xcafeu32</code>	<code>u32</code>	51966
<code>0b0010_1010</code>	Inferred	42
<code>0o106</code>	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. Only ASCII characters may appear in byte literals.

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 can only be written using a stand-in notation, introduced by a backslash:

Character	Byte literal	Numeric equivalent
Single quote, '	<code>b'\''</code>	<code>39u8</code>
Backslash, \	<code>b'\\'</code>	<code>92u8</code>
Newline	<code>b'\n'</code>	<code>10u8</code>
Carriage return	<code>b'\r'</code>	<code>13u8</code>
Tab	<code>b'\t'</code>	<code>9u8</code>

For characters that are hard to write or read, you can write their code in hexadecimal instead. A byte literal of the form `b'\xHH'`, where HH is any two-digit hexadecimal number, represents the byte whose value is HH. For example, you can write a byte literal for the ASCII “escape” control character as `b'\x1b'`, since the ASCII code for “escape” is 27, or 1B in hexadecimal. Since byte literals are just another notation for `u8` values, consider whether a simple numeric literal might be more legible: it probably makes sense to use `b'\x1b'` instead of simply 27 only when you want to emphasize that the value represents an ASCII code.

You can convert from one integer type to another using the `as` operator. We explain how conversions work in “[Type Casts](#)”, but here are some examples:

```
assert_eq!( 10_i8 as u16,    10_u16); // in range
assert_eq!( 2525_u16 as i16,  2525_i16); // in range

assert_eq!( -1_i16 as i32,   -1_i32); // sign-extended
assert_eq!(65535_u16 as i32, 65535_i32); // zero-extended

// Conversions that are out of range for the destination
// produce values that are equivalent to the original modulo 2^N,
// where N is the width of the destination in bits. This
// is sometimes called "truncation".
assert_eq!( 1000_i16 as u8,   232_u8);
assert_eq!(65535_u32 as i16, -1_i16);

assert_eq!( -1_i8  as u8,    255_u8);
assert_eq!( 255_u8 as i8,    -1_i8);
```

Like any other sort of value, integers can have methods. The standard library provides some basic operations, which you can look up in the online documentation. Note that the documentation contains separate pages for the type itself (search for “`i32` (primitive type)”, say), and for the module dedicated to that type (search for “`std::i32`”). For example:

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

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

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

Rust complains:

```
error: no method named `abs` found for type `<integer>` in the current scope
```

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

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

Note that method calls have a higher precedence than unary prefix operators, so be careful when applying methods to negated values. Without the parentheses around `-4i32` in the second assertion, `-4i32.abs()` would apply the `abs` method to the positive value 4, producing positive 4, and then negate that, producing -4.

Checked, Wrapping, and Saturating Arithmetic

Sometimes Rust's default rules for dealing with arithmetic overflow aren't what you need. For such cases, the integer types provide methods that let you spell out exactly how you want overflows handled. These methods fall in three general categories:

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

```
// The sum of 10 and 20 can be represented as a u8.
```

```

assert_eq!(10_u8.checked_add(20), Some(30));

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

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

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

```

- *Wrapping* operations return the value equivalent to the mathematically correct result modulo the range of the value, in both debug and release builds:

```

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

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

// In bitwise shift operations, the shift distance is wrapped to fall
// within
// the size of the value. So a shift of 17 bits in a 16-bit type is a
// shift
// of 1.
assert_eq!(5_i16.wrapping_shl(17), 10);

```

- *Saturating* operations return the representable value that is closest to the mathematically correct result:

```

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

```

The operation names that follow the `checked_`, `wrapping_`, or `saturating_` prefix are as follows:

Operation	Name Suffix	Example
addition	add	100_i8.checked_add(27) == Some(127)
Subtraction	sub	10_u8.checked_sub(11) == None
Multiplication	mul	128_u8.saturating_mul(3) == 255
Division	div	64_u16.wrapping_div(8) == 8
Remainder	rem	(-32768_i16).wrapping_rem(-1) == 0
Negation	neg	(-128_i8).checked_neg() == None
Absolute value	abs	(-32768_i16).wrapping_abs() == -32768
Exponentiation	pow	3_u8.checked_pow(4) == Some(81)
Bitwise left shift	shl	10_u32.wrapping_shl(34) == 40
Bitwise right shift	shr	40_u64.wrapping_shr(66) == 10

Floating-Point Types

Rust provides IEEE single- and double-precision floating-point types. 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 $+1.8 \times 10^{308}$

Rust's `f32` and `f64` correspond to the `float` and `double` types in C and C++ implementations that support IEEE floating point, and in Java, which always uses IEEE floating point.

Floating-point literals have the general form diagrammed in [Figure 2-1](#).

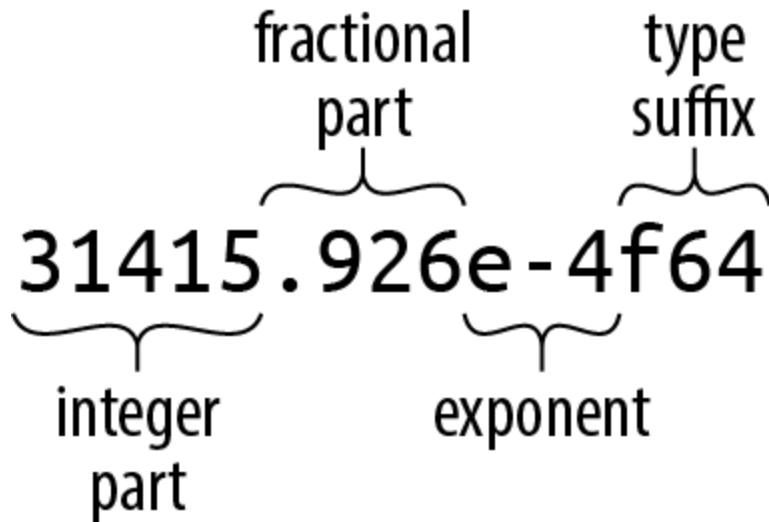


Figure 2-1. A floating-point literal

Every part of a floating-point number after the integer part is optional, but at least one of the fractional part, exponent, or type suffix must be present, to distinguish it from an integer literal. The fractional part may consist of a lone decimal point, so `5.` is a valid floating-point constant.

If a floating-point literal lacks a type suffix, Rust checks the context to see how the values are used, much as it does for integer literals. If it ultimately finds that either floating-point type could fit, it chooses `f64` by default.

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
<code>-1.5625</code>	Inferred	$-(1 \frac{9}{16})$
<code>2.</code>	Inferred	2
<code>0.25</code>	Inferred	$\frac{1}{4}$
<code>1e4</code>	Inferred	10,000
<code>40f32</code>	<code>f32</code>	40
<code>9.109_383_56e-31f64</code>	<code>f64</code>	Roughly $9.10938356 \times 10^{-31}$

The standard library's `std::f32` and `std::f64` modules define constants for the IEEE-required special values like `INFINITY`, `NEG_INFINITY` (negative infinity), `NAN` (the not-a-number value), and `MIN` and `MAX` (the largest and smallest finite values). The `std::f32::consts` and `std::f64::consts` modules provide various commonly used mathematical constants like `E`, `PI`, and the square root of two.

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 names “`f32` (primitive type)” and “`f64` (primitive type)”. Some examples:

```
assert_eq!(5f32.sqrt() * 5f32.sqrt(), 5.); // exactly 5.0, per IEEE
assert_eq!((-1.01f64).floor(), -2.0);
assert!((-1. / std::f32::INFINITY).is_sign_negative());
```

Again, method calls have a higher precedence than prefix operators, so be sure to correctly parenthesize method calls on negated values.

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

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

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

The lack of implicit conversions sometimes makes a Rust expression more verbose than the analogous C or C++ code would be. However, implicit

integer conversions have a well-established record of causing bugs and security holes, especially when the integers in question represent the size of something in memory, and an unanticipated overflow occurs. In our experience, the act of writing out numeric conversions in Rust has alerted us to problems we would otherwise have missed.

We explain exactly how conversions behave in “[Type Casts](#)”.

The `bool` Type

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

Many languages are lenient about using values of other types in contexts that require a Boolean value: C and C++ implicitly convert characters, integers, floating-point numbers, and pointers to Boolean values, so they can be used directly as the condition in an `if` or `while` statement. Python permits strings, lists, dictionaries, and even sets in Boolean contexts, treating such values as true if they’re nonempty. 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.

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, not as an array of characters.

Character literals are characters enclosed in single quotes, like '`'8'`' or '`'!'`'. You can use the full breadth of Unicode: '`'鏽'`' is a `char` literal representing the Japanese kanji for *sabi* (rust).

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 code point in hexadecimal:

- If the character's code point 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 code point of the character `*` is 42, or 2A in hexadecimal.
- You can write any Unicode character as '`\u{HHHHHH}`', where HHHHHH is a hexadecimal number up to six digits long, with underscores allowed for grouping as usual. For example, the character literal '`\u{CA0}`' represents the character “ಠ”, a Kannada character used in the Unicode Look of Disapproval, “ಠ_ಠ”. The same literal could also be simply written as '`ಠ`'.

A `char` always holds a Unicode code point 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 codespace (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 code points, so those conversions would require runtime 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 code point, 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 online documentation by searching for “`char` (primitive type)”, and for the module “`std::char`”. For example:

```
assert_eq!( '*' .is_alphabetic(), false );
assert_eq!( 'β' .is_alphabetic(), true );
assert_eq!( '8' .to_digit(10), Some(8));
assert_eq!( '🦀' .len_utf8(), 3);
assert_eq!( std::char::from_digit(2, 10), Some('2'));
```

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 in “String Types”.

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 aren’t much like 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, like **t.4**. You can’t write **t.i** or **t[i]** to get the **i**’th element.

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 XREF HERE, we needed to pass the width and height of the image to the functions that plot it and write it to disk. We could declare a struct with `width` and `height` members, but that's pretty heavy notation for something so obvious, so we just used a tuple:

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

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

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

For example, a function that returns no value has a return type of `()`. The standard library's `std::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_image` example we mentioned before has a return type of `Result<(), std::io::Error>`, meaning that the function returns a `std::io::Error` value if something goes wrong, but returns no value on success.

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)`. Rust consistently permits an extra trailing comma everywhere commas are used: function arguments, arrays, struct and enum definitions, and so on. This may look odd to human readers, but it can make diffs easier to read when entries are added and removed at the end of a list.

For consistency's 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.

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 Rectangle` contains a field `Point upperLeft;`, then `upperLeft` is a reference to another separately created `Point` 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` (pronounced “ref String”) is a reference to a `String` value, a `&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. At run time, a reference to an `i32` is a single machine word holding the address of the `i32`, which may be on the stack or in the heap. The expression `&x` produces a reference to `x`; in Rust terminology, we say that it *borrow*s a reference to `x`. Given a reference `r`, the expression `*r` refers to the value `r` points to. These are very much like the `&` and `*` operators in C and C++. And like a C pointer, a reference does not automatically free any resources when it goes out of scope.

Unlike C pointers, however, Rust references are never null: there is simply no way to produce a null reference in safe Rust. And unlike C, Rust tracks the ownership and lifetimes of values, so mistakes like dangling pointers, double frees, and pointer invalidation are ruled out at compile time.

Rust references come in two flavors:

`&T`

A shared reference. You can have many shared references to a given value at a time, but they are read-only: modifying the value they point to is forbidden, as with `const T*` in C.

`&mut T`

A mutable reference. You can read and modify the value it points to, as with a `T*` in C. But for as long as the reference exists, you may not have any other references of any kind to that value. In fact, the only way you may access the value at all is through the mutable reference.

Rust uses this dichotomy between shared and mutable references to enforce a “single writer *or* multiple readers” rule: either you can read and write the value, or it can be shared by any number of readers, but never both at the same time. This separation, enforced by compile-time checks, is central to Rust’s safety guarantees. [Chapter 4](#) 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. Moves are essential to the way Rust handles heap-allocated values; we explain all this in detail in [Chapter 3](#).

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 it points to. For example, raw pointers may be null, or they may point to memory that has been freed or that now

contains a value of a different type. All the classic pointer mistakes of C++ are offered for your enjoyment.

However, you may only dereference raw pointers within an `unsafe` block. An `unsafe` block is Rust's opt-in mechanism for advanced language features whose safety is up to you. If your code has no `unsafe` blocks (or if those it does are written correctly), then the safety guarantees we emphasize throughout this book still hold. For details, see XREF HERE.

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

Given a value `v` of any of these three types, the expression `v.len()` gives the number of elements in `v`, and `v[i]` refers to the `i`'th element of `v`. The first element is `v[0]`, and the last element is `v[v.len() - 1]`. Rust checks that `i` always falls within this range; if it doesn't, the expression panics. The

length of `v` may be zero, in which case any attempt to index it will panic. `i` must be a `usize` value; you can't use any other integer type as an index.

Arrays

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

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

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

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

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

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

You'll see this syntax used for fixed-size buffers: `[0u8; 1024]` 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.)

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

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

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

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

Vectors

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

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

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

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

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

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

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

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

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

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

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

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

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

```
// A palindrome!
let mut palindrome = vec!["a man", "a plan", "a canal", "panama"];
```

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

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

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

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

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

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

Just as a vector's `len` method returns the number of elements it contains now, its `capacity` method returns the number of elements it could hold

without reallocation:

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

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

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

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

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

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

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

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

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

```
let mut v = vec!["Snow Puff", "Glass Gem"];
assert_eq!(v.pop(), Some("Glass Gem"));
```

```
assert_eq!(v.pop(), Some("Snow Puff"));
assert_eq!(v.pop(), None);
```

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

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

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

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

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

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

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 the `&Vec<f64>` reference and the `&[f64; 4]` reference to slice references that point directly to the data.

By the end, memory looks like [Figure 2-2](#).

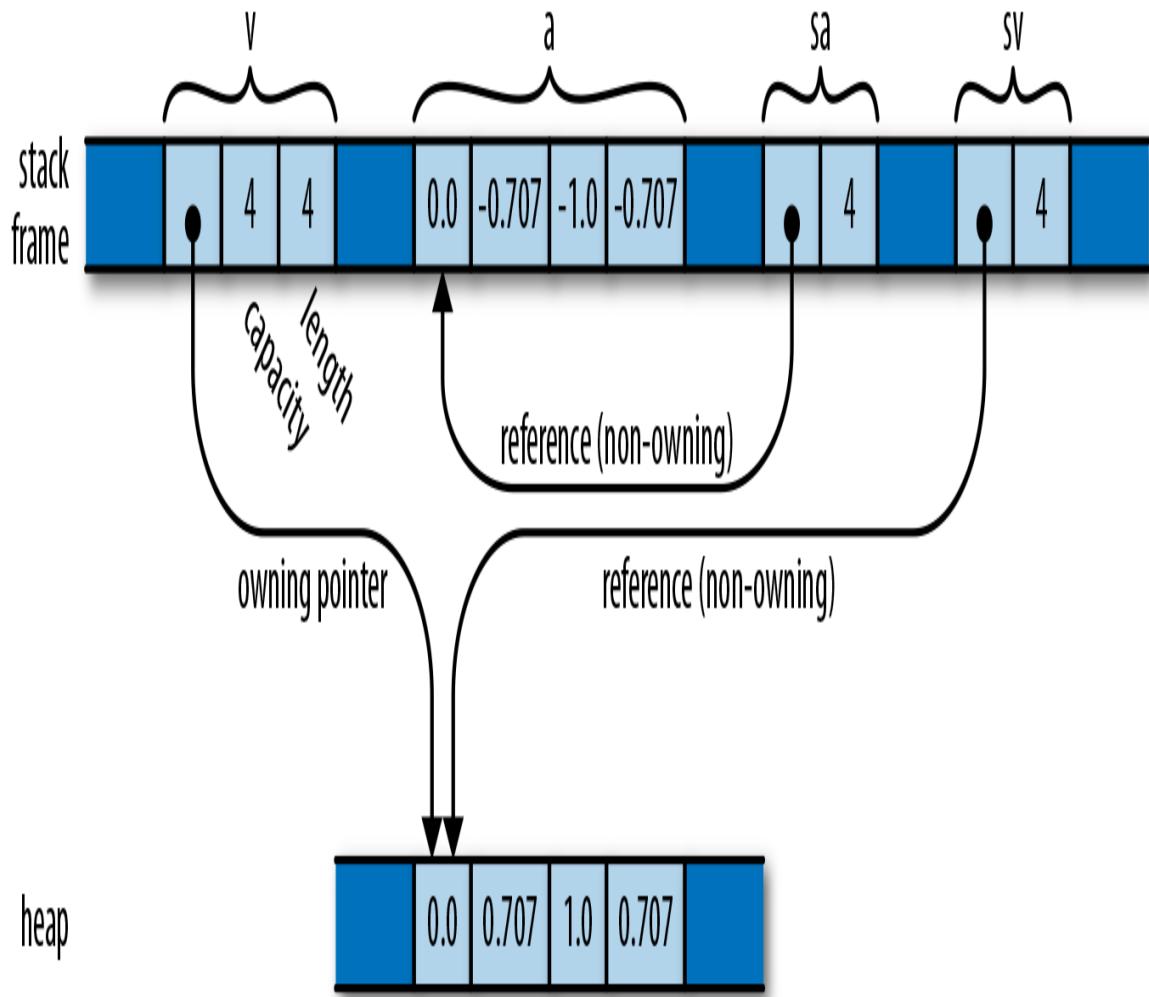


Figure 2-2. A vector v and an array a in memory, with slices sa and sv referring to each

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

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

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

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

Since slices almost always appear behind references, we often just refer to types like `&[T]` or `&str` as 'slices', using the shorter name for the more common concept.

String 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 introduce Rust's two string types. We provide more detail about strings and text handling in XREF HERE.

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";
```

In string literals, unlike `char` literals, single quotes don't need a backslash escape, and double quotes do.

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 paths.) For these cases, Rust offers *raw strings*. A raw string is tagged with the lowercase letter `r`. All backslashes and whitespace characters inside a raw string are included verbatim in the string. No escape sequences are recognized.

```
let default_win_install_path = r"C:\Program Files\Gorillas";  
  
let pattern = Regex::new(r"\d+(\.\d+)*");
```

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

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

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

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: byte strings can span multiple lines, use escape sequences, and use backslashes to join lines. Raw byte strings start with `b|r"`.

Byte strings can't contain arbitrary Unicode characters. They must make do with ASCII and `\xHH` escape sequences.

The type of `method` shown here is `&[u8; 3]`: it's a reference to an array of three 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

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

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

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

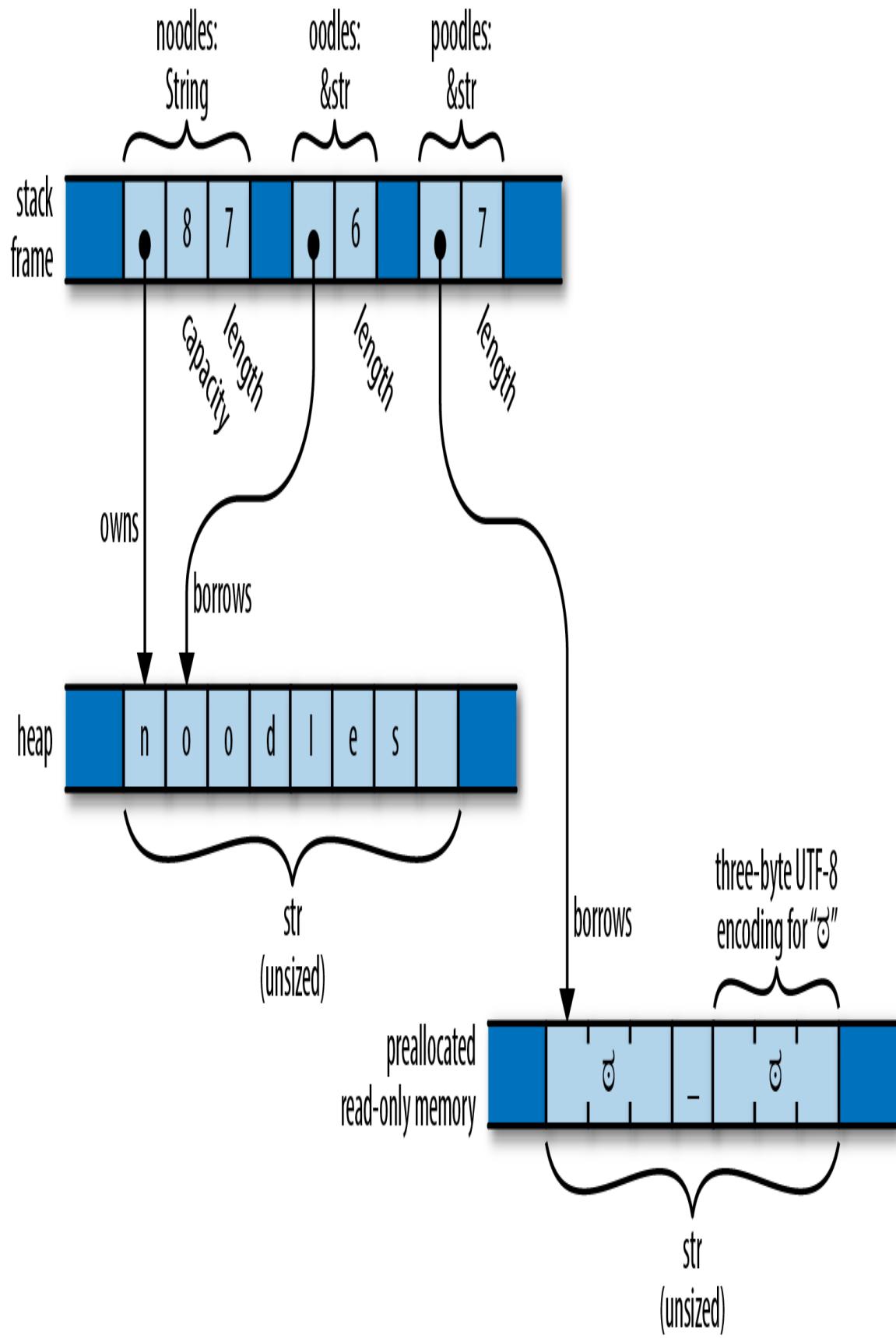


Figure 2-3. String, &str, and str

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

A `&str` (pronounced “stir” or “string slice”) is a reference to a run of UTF-8 text owned by someone else: it “borrows” the text. In the example, `oodles` is a `&str` referring to the last six bytes of the text belonging to `noodles`, so it represents the text “oodles”. Like other slice references, a `&str` is a fat pointer, containing both the address of the actual data and its length. You can think of a `&str` as being nothing more than a `&[u8]` that is guaranteed to hold well-formed UTF-8.

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

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

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

It is impossible to modify a `&str`:

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

For creating new strings at run time, use `String`.

The type `&mut str` does exist, but it is not very useful, since almost any operation on UTF-8 can change its overall byte length, and a slice cannot

reallocates its referent. In fact, the only operations available on `&mut str` are `make_ascii_uppercase` and `make_ascii_lowercase`, which modify the text in place and affect only single-byte characters, by definition.

String

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

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

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

There are several ways to create `String`s:

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

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

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

```
assert_eq!(format!("{}°{:02}'{:02}N", 24, 5, 23),  
          "24°05'23N".to_string());
```

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

```
let bits = vec!["veni", "vidi", "vici"];  
assert_eq!(bits.concat(), "venividivici");  
assert_eq!(bits.join(", "), "veni, 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 suffice to point out that a `&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!("ONE".to_lowercase() == "one");
```

Strings also support the comparison operators `<`, `<=`, `>`, and `>=`, as well as many useful methods and functions that you can find in the online documentation by searching for “`str` (primitive type)” or the “`std::str`” module (or just flip to XREF HERE). Here are a few examples:

```
assert!("peanut".contains("nut"));
```

```
assert_eq!("ດອງ".replace("ດ", "ນ"), "ນົມ");
assert_eq!("    ດາວໂຫຼວນ".trim(), "ດາວໂຫຼວນ");

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

Keep in mind that, given the nature of Unicode, simple `char`-by-`char` comparison does *not* always give the expected answers. For example, the Rust strings `"th\ufe9e"` and `"the\u{301}"` are both valid Unicode representations for thé, the French word for tea. Unicode says they should both be displayed and processed in the same way, but Rust treats them as two completely distinct strings. Similarly, Rust's ordering operators like `<` use a simple lexicographical order based on character code point values. This ordering only sometimes resembles the ordering used for text in the user's language and culture. We discuss these issues in more detail in XREF HERE.

Other String-Like Types

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

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

- Stick to `String` and `&str` for Unicode text.
- 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 working with environment variable names and command-line arguments in the native form presented by the operating

system, use `OsString` and `&OsStr`.

- 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 types, and we'll cover them in three successive chapters: structs in [Chapter 8](#), enums in [Chapter 9](#), and traits in [Chapter 10](#).

Functions and closures have their own types, covered in XREF HERE. And the types that make up the standard library are covered throughout the book. For example, XREF HERE presents the standard collection types.

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

Chapter 3. Ownership

A NOTE FOR EARLY RELEASE READERS

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

This will be the 4th chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at jimb@red-bean.com.

I've found that Rust has forced me to learn many of the things that I was slowly learning as 'good practice' in C/C++ before I could even compile my code. ... I want to stress that Rust isn't the kind of language you can learn in a couple days and just deal with the hard/technical/good-practice stuff later. You will be forced to learn strict safety immediately and it will probably feel uncomfortable at first. However in my own experience, this has led me towards feeling like compiling my code actually means something to me again.

—Mitchell Nordine

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 in 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: pointer misuse has been a common culprit in public databases of reported security problems for as long as that data has been collected.

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, and understanding why memory wasn’t freed when you expected can be 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. This chapter and the next are devoted to explaining exactly what these restrictions are and why they work. For now, suffice it to say that 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 to allow Rust’s compile-time checks to verify that your program is free of memory safety 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. Using Rust’s carefully designed threading primitives, the rules that ensure your code uses memory correctly also serve to prove that it is free of data races. A bug in a Rust program cannot cause one thread to corrupt another’s data, introducing hard-to-reproduce failures in unrelated parts of the system. The nondeterministic behavior inherent in multithreaded code is isolated to those features designed to handle it— mutexes, message channels, atomic values, and so on—rather than appearing in ordinary memory references. Multithreaded code in C and C++ has earned its ugly reputation, but Rust rehabilitates it quite nicely.

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

Rust’s rules are probably unlike what you’ve seen in other programming languages. Learning how to work with them and turn them to your advantage is, in our opinion, the central challenge of learning Rust. In this chapter, we’ll first 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 destroys its possessions along with it.

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

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

The string `s` is usually represented in memory as shown in [Figure 3-1](#).

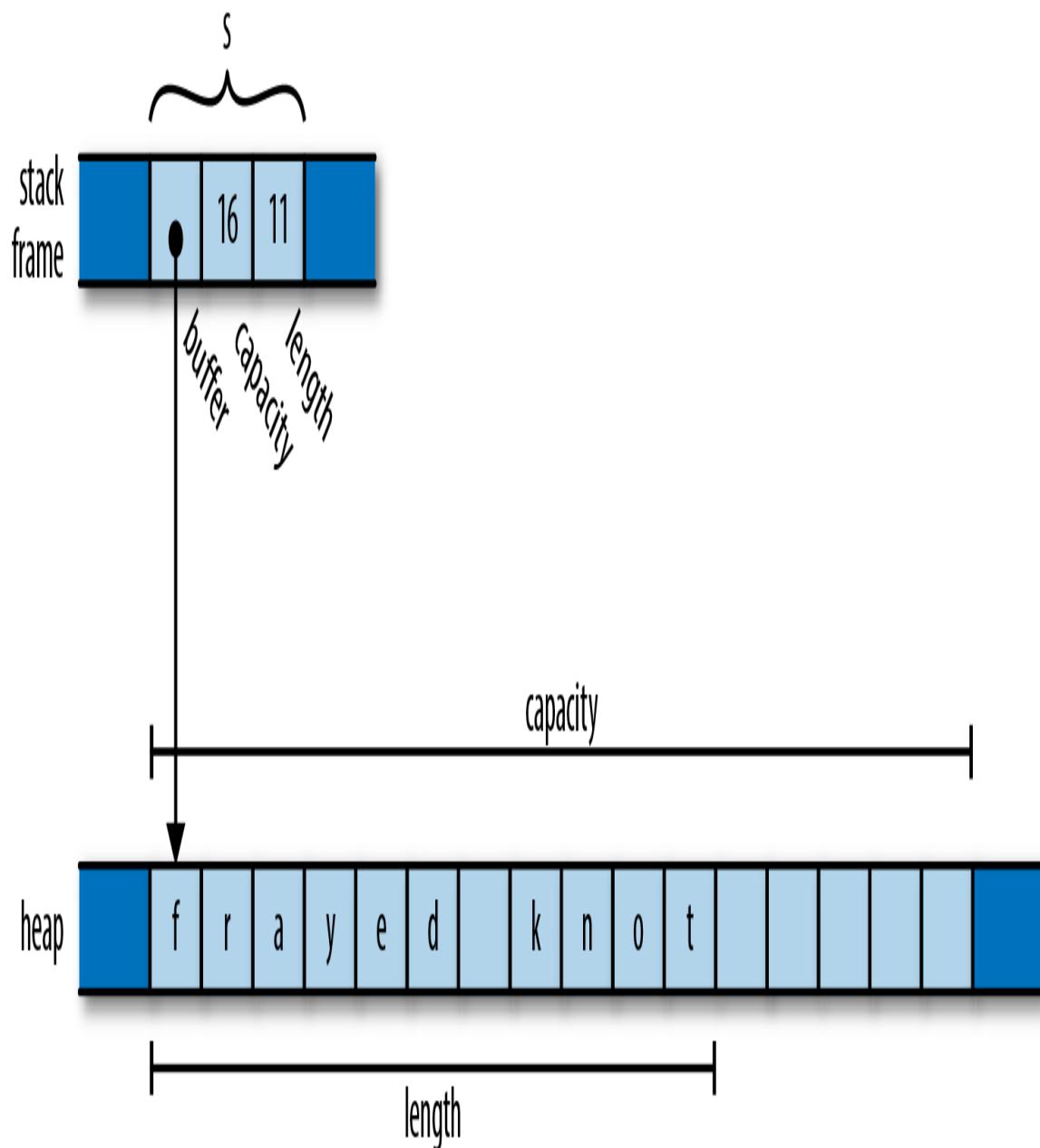


Figure 3-1. A C++ `std::string` value on the stack, pointing to its heap-allocated buffer

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

A `std::string` owns its buffer: when the program destroys the string, the string's destructor frees the buffer. In the past, some C++ libraries shared a single buffer among several `std::string` values, using a reference count to decide when the buffer should be freed. Newer versions of the C++ specification effectively preclude that representation; all modern C++ libraries use the approach shown here. In these situations it's generally understood that, although it's fine for other code to create temporary pointers to the owned memory, it is that code's responsibility to make sure its pointers are gone before the owner decides to destroy the owned object. You can create a pointer to a character living in a `std::string`'s buffer, but when the string is destroyed, your pointer becomes invalid, and it's up to you to make sure you don't use it anymore. The owner determines the lifetime of the owned, and everyone else must respect its decisions.

Rust takes this principle out of the comments and makes it explicit in the language. In Rust, every value has a single owner that determines its lifetime. When the owner is freed—*dropped*, in Rust terminology—the owned value is dropped too. These rules are meant to make it easy for you to find any given value's lifetime simply by inspecting the code, giving you the control over its lifetime that a systems language should provide.

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

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

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

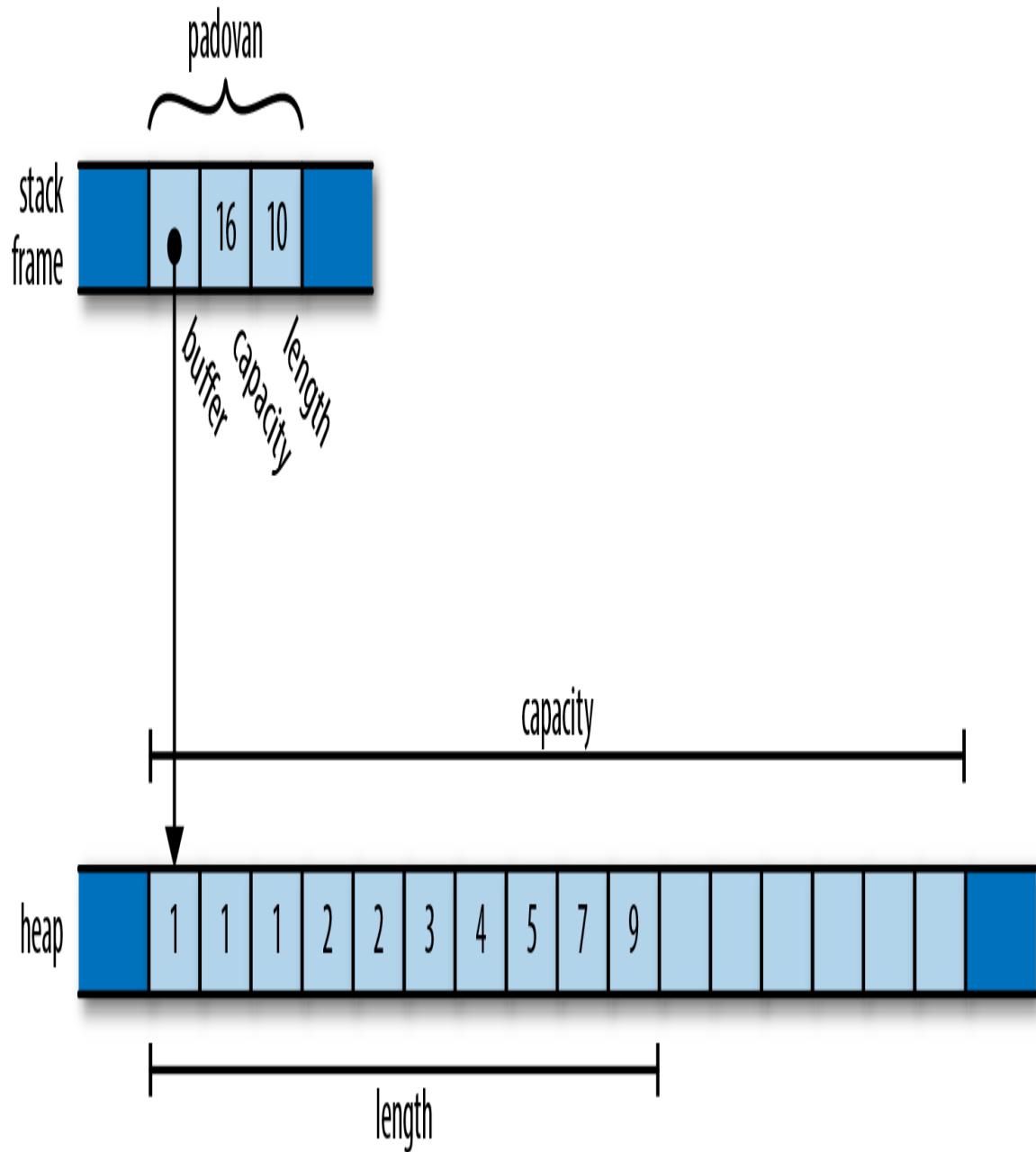


Figure 3-2. A `Vec<i32>` on the stack, pointing to its buffer in the heap

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

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

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

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

When the program calls `Box::new`, it allocates space for a tuple of two `f64` values 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 [Figure 3-3](#).

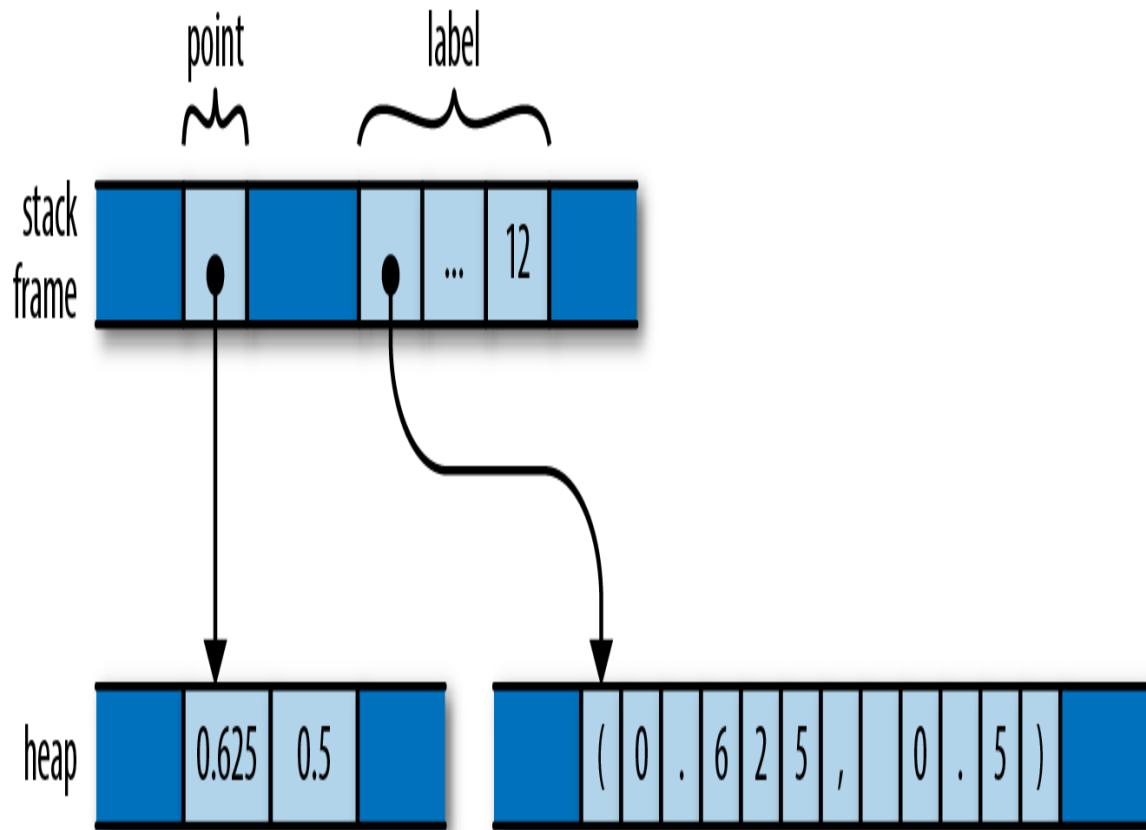


Figure 3-3. Two local variables, each owning memory in the heap

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

Just as variables own their values, structs own their fields, 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 structs, each of which holds a string and a number. In memory, the final value of `composers` looks like [Figure 3-4](#).

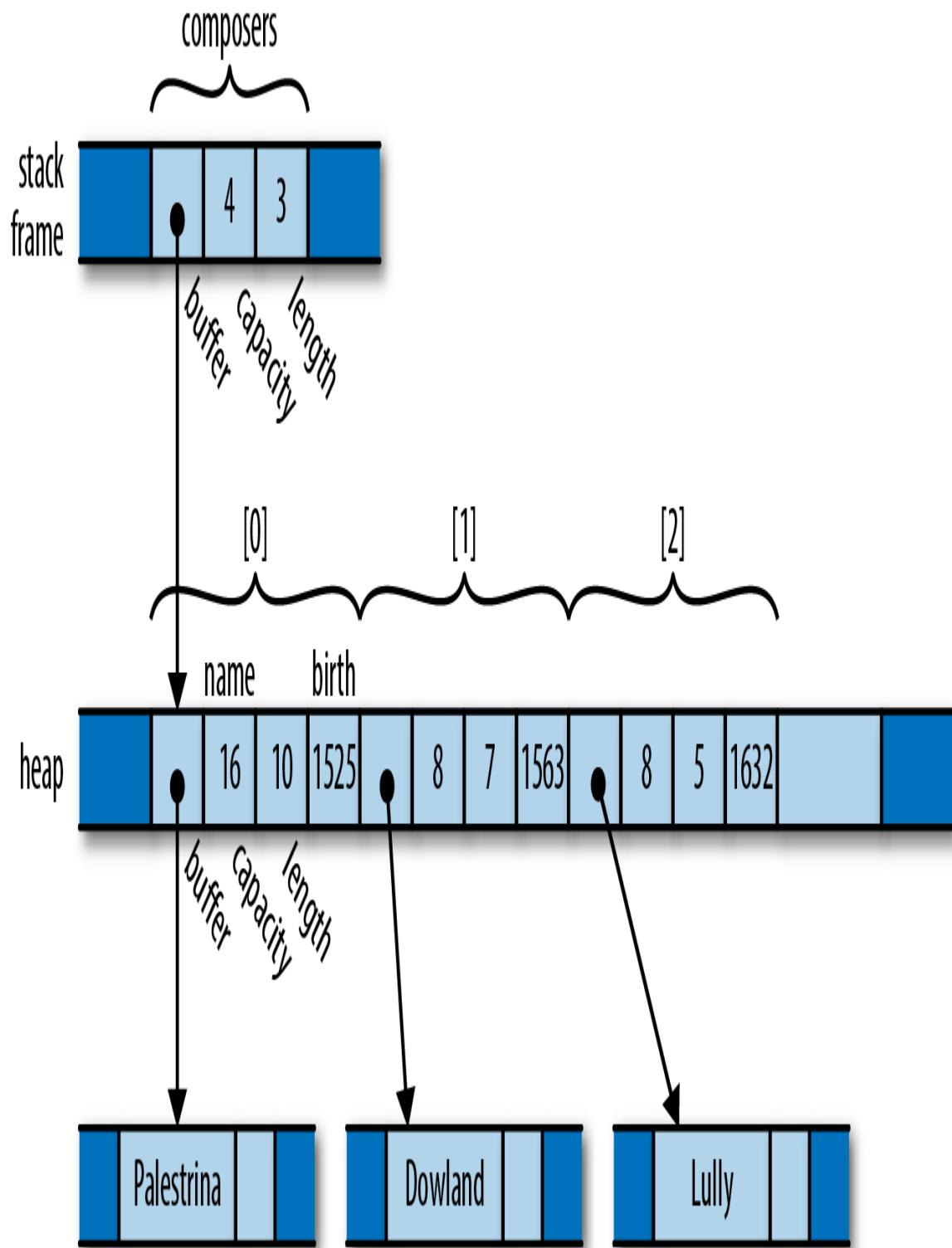


Figure 3-4. A more complex tree of ownership

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

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

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

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

In a certain sense, Rust is less powerful than other languages: every other practical programming language lets you build arbitrary graphs of objects that point to each other in whatever way you see fit. But it is exactly

because Rust is less powerful than the analyses the language can carry out on your programs can be more powerful. Rust's safety guarantees are possible exactly because the relationships it may encounter in your code are more tractable. This is part of Rust's “radical wager” we mentioned earlier: in practice, Rust claims, there is usually more than enough flexibility in how one goes about solving a problem to ensure that at least a few perfectly fine solutions fall within the restrictions the language imposes.

That said, 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.
- The standard library provides the reference-counted pointer types `Rc` and `Arc`, which allow values to have multiple owners, under some restrictions.
- You can “borrow a reference” to a value; references are nonowning pointers, with limited lifetimes.

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

Moves

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

You may be surprised that Rust would change the meaning of such fundamental operations; surely assignment is something that should be pretty well nailed down at this point in history. However, if you look closely

at how different languages have chosen to handle assignment, you'll see that there's actually significant variation from one school to another. The comparison also makes the meaning and consequences of Rust's choice easier to see.

Consider the following Python code:

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

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

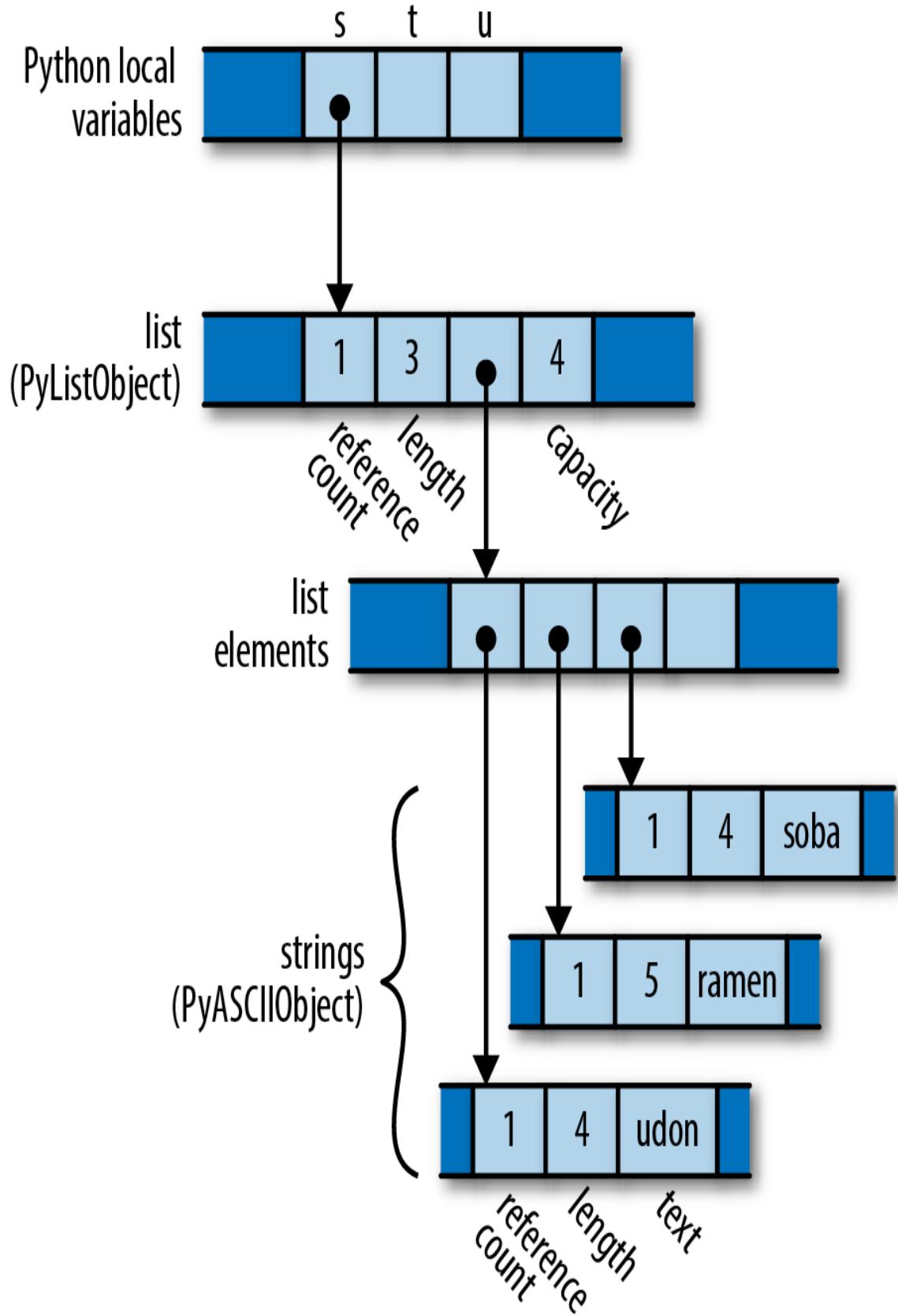


Figure 3-5. How Python represents a list of strings in memory

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

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

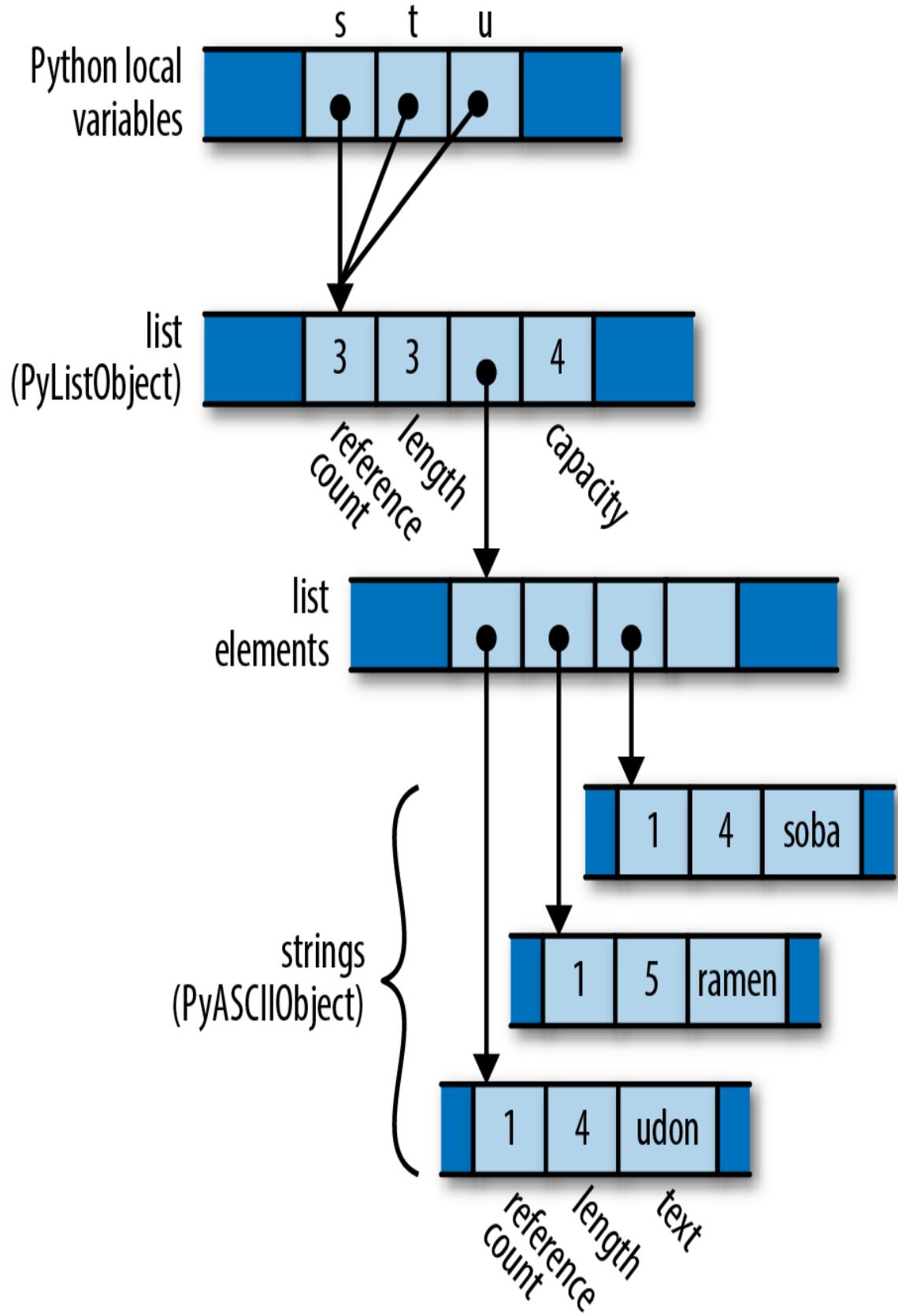


Figure 3-6. The result of assigning s to both t and u in Python

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

Now consider the analogous C++ code:

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

The original value of `s` looks like [Figure 3-7](#) in memory.

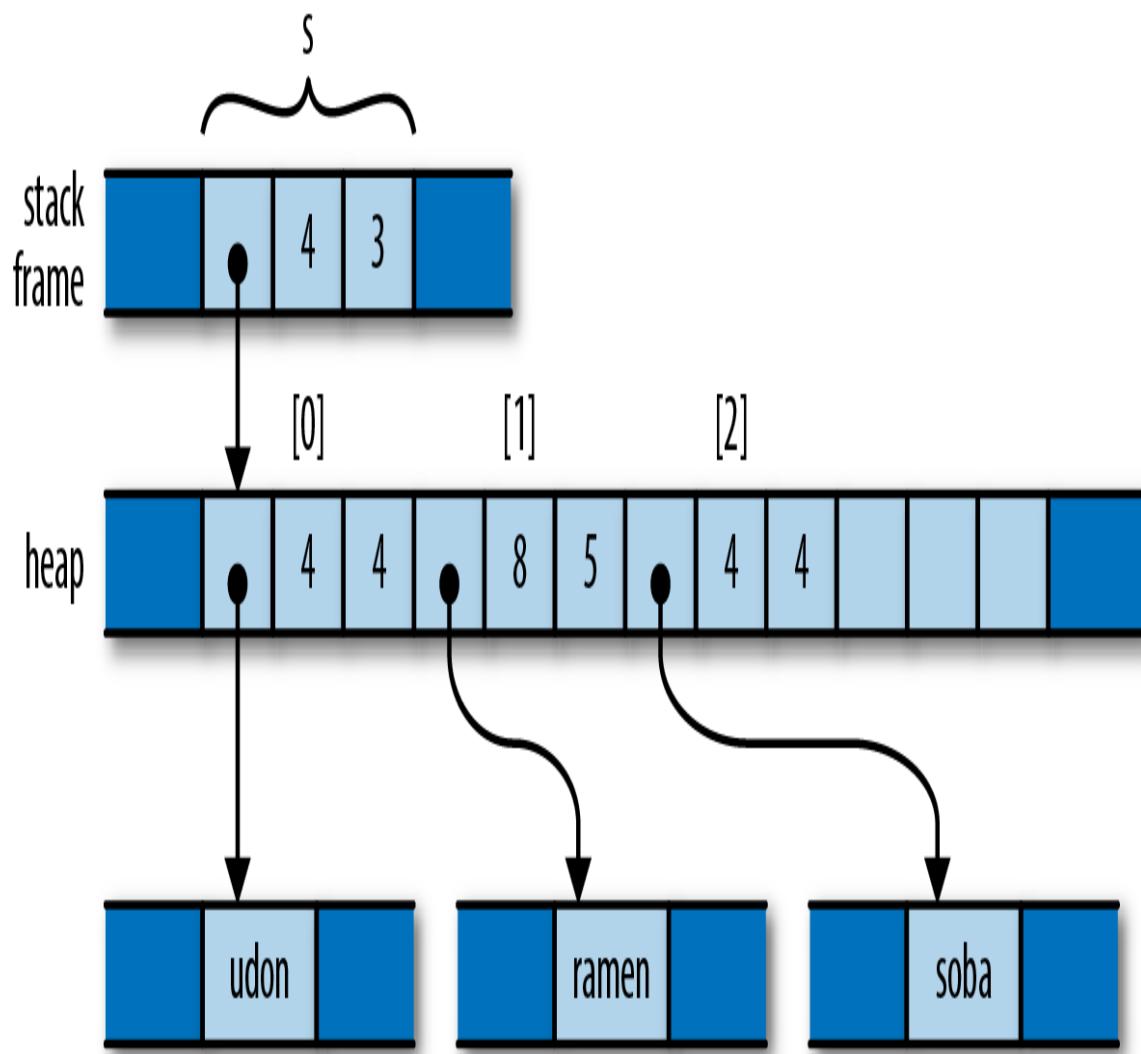


Figure 3-7. How C++ represents a vector of strings in memory

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

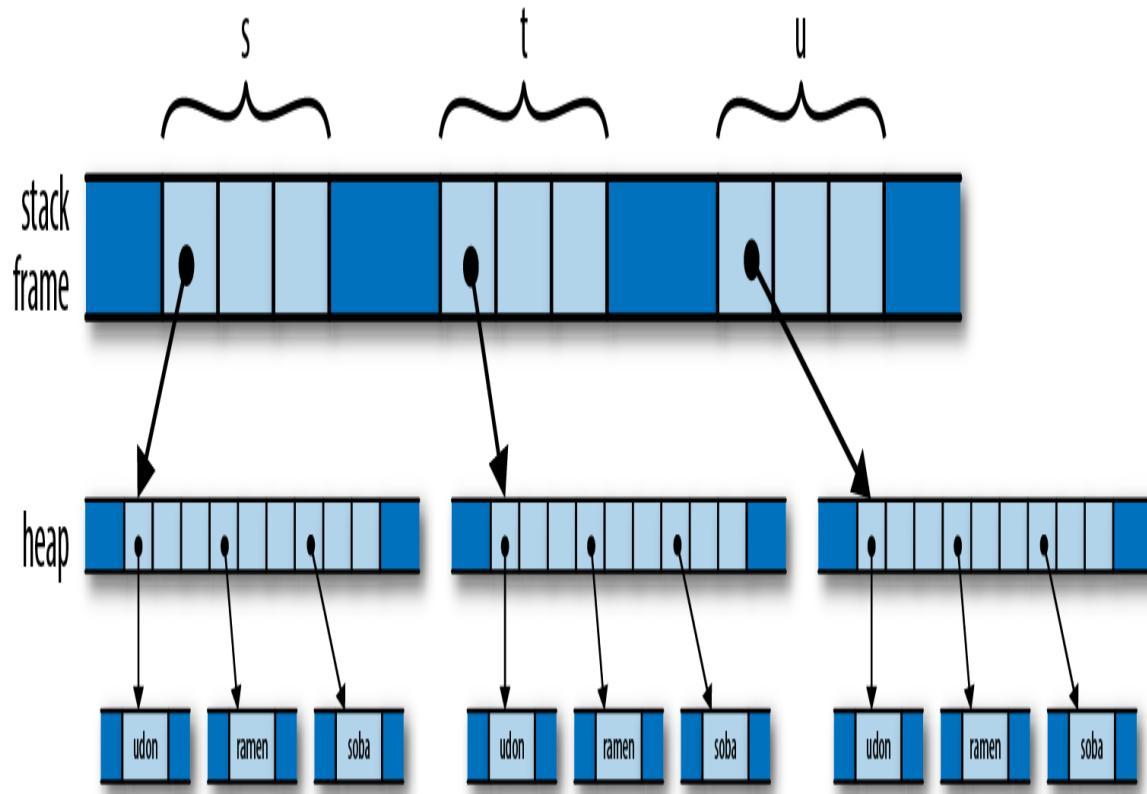


Figure 3-8. The result of assigning *s* to both *t* and *u* in C++

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

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

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

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

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

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

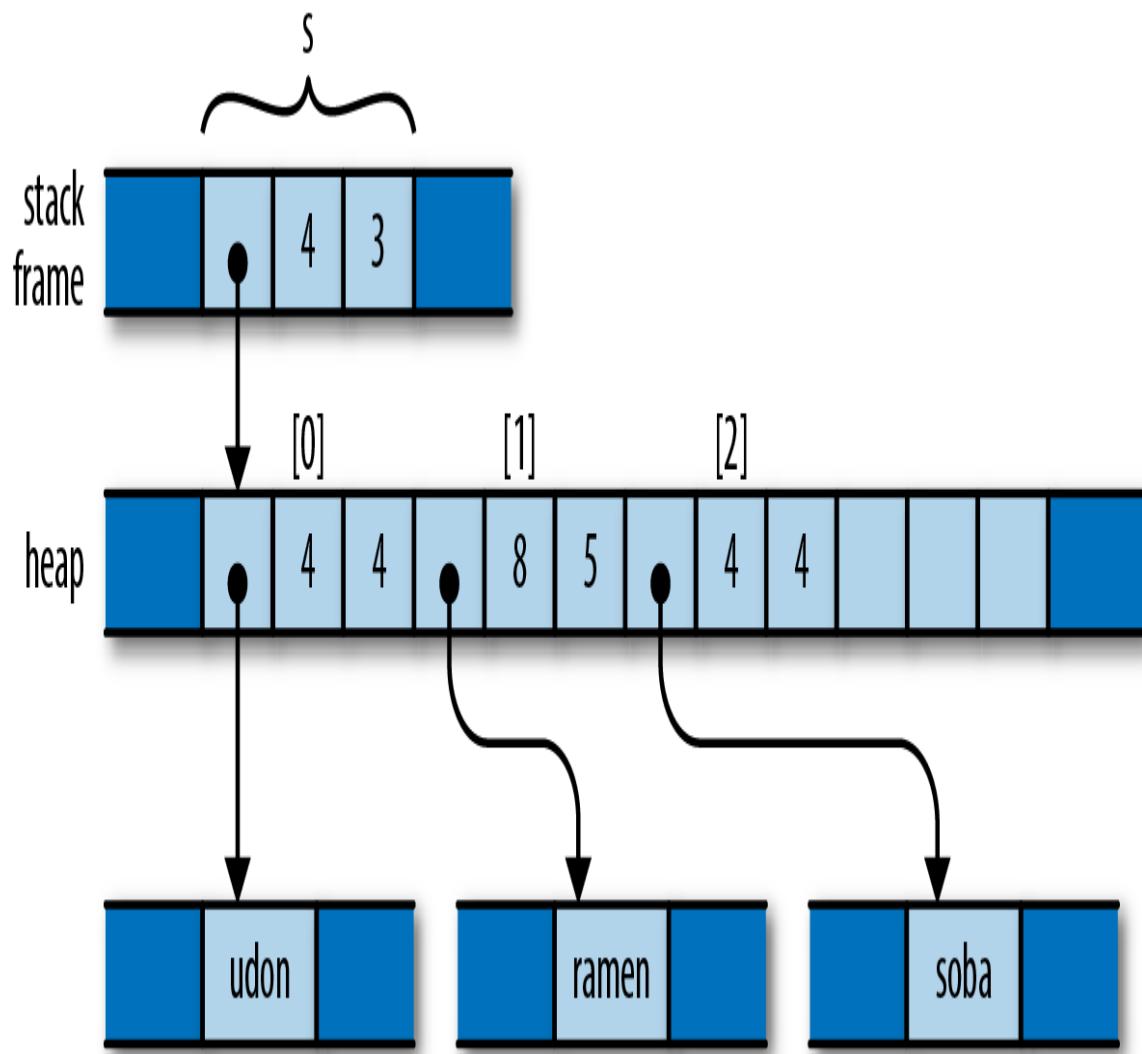


Figure 3-9. How Rust represents a vector of strings in memory

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

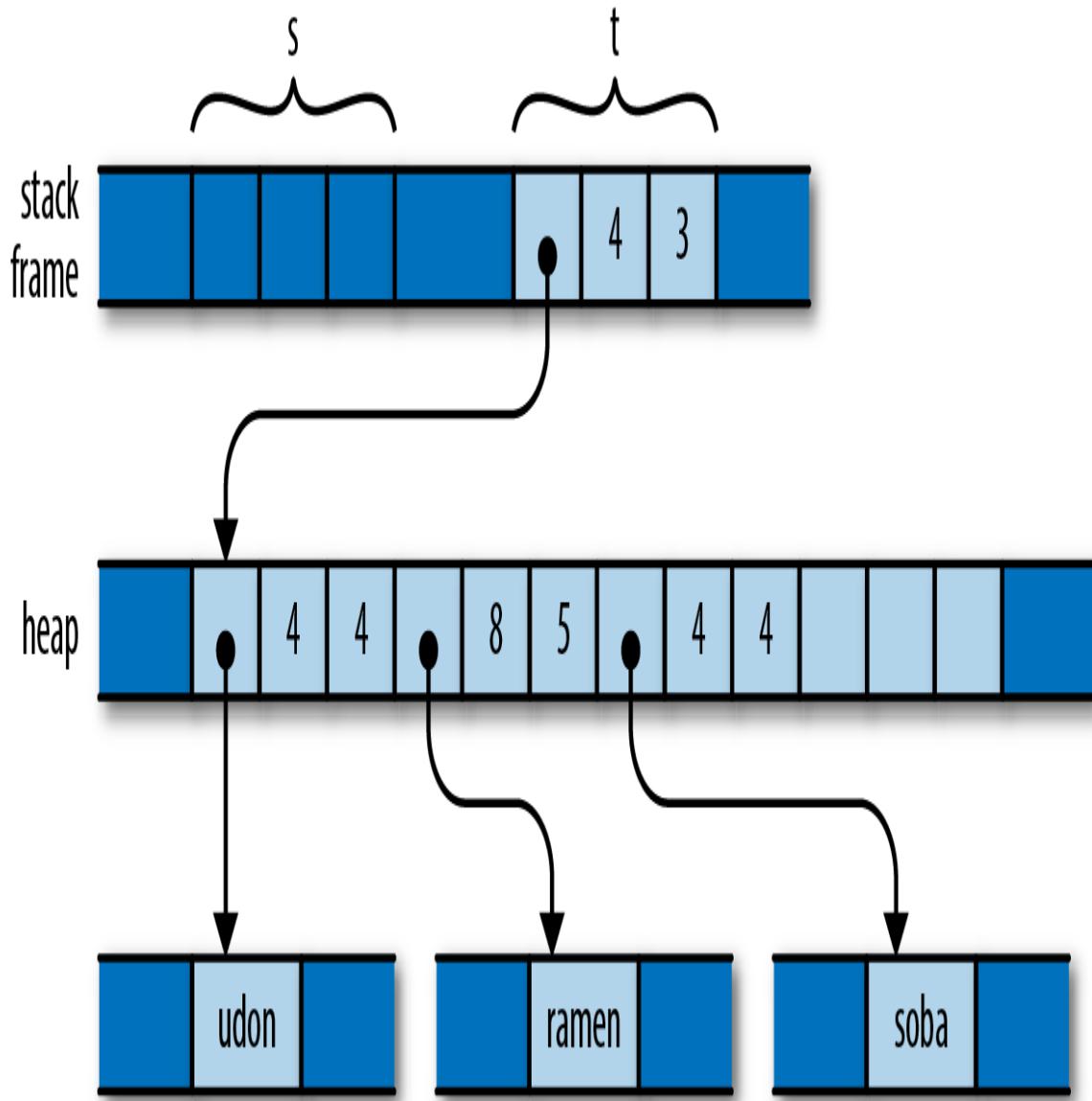


Figure 3-10. The result of assigning `s` to `t` in Rust

What has happened here? The initialization `let t = s;` moved the vector's three header fields from `s` to `t`; now `t` owns the vector. The vector's elements stayed just where they were, and nothing happened to the strings either. Every value still has a single owner, although one has changed

hands. There were no reference counts to be adjusted. And the compiler now considers `s` uninitialized.

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

```
error[E0382]: use of moved value: `s`
--> ownership_double_move.rs:9:9
|
8 |     let t = s;
|         - value moved here
9 |     let u = s;
|         ^ value used here after move
|
```

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

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

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

You could also re-create Python's behavior by using Rust's reference-counted pointer types; we'll discuss those shortly in “[Rc and Arc: Shared Ownership](#)”.

More Operations That Move

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

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

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

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

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

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

You may now have 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, not a pointer to the vector, but the vector itself: its ownership moves from `Vec::new` to the variable `composers`. Similarly, the `to_string` call returns a fresh `String` instance.

Constructing new values

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

Passing values to a function

The entire `Person` structure, not just a pointer, is passed 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, the moves always apply to the value proper, not the heap storage they own. For vectors and strings, the *value proper* is the three-word header alone; the potentially large element arrays and text buffers sit where they are in the heap. Second, the Rust compiler's code generation is good at "seeing through" all these moves; in practice, the machine code often stores the value directly where it belongs.

Moves and Control Flow

The previous examples all have very simple control flow; how do moves interact with more complicated code? The general principle is that, if it's possible for a variable to have had its value moved away, and it hasn't definitely been given a new value since, it's considered uninitialized. For

example, if a variable still has a value after evaluating an `if` expression's condition, then we can use it in both branches:

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

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

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

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

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

Moves and Indexed Content

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

```
// Build a vector of the strings "101", "102", ... "105"
let mut v = Vec::new();
```

```

for i in 101 .. 106 {
    v.push(i.to_string());
}

// Pull out random elements from the vector.
let third = v[2];
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 preceding code with the following error:

```

error[E0507]: cannot move out of index of `Vec<String>`
|
14 |     let third = v[2];
      ^^^^
      |
      |
      move occurs because value has type `String`  

      which does not implement the `Copy` trait
      help: consider borrowing here: `&v[2]`

```

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

```

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

```

```

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

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

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

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

```

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

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

```

let v = vec![
    "liberté".to_string(),
    "égalité".to_string(),
    "fraternité".to_string()];
for mut s in v {
    s.push('!');
    println!("{}", s);
}

```

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

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

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

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

You can't do this:

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

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

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

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

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

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

Copy Types: The Exception to Moves

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

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

```
let str1 = "somnambulance".to_string();
let str2 = str1;

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

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

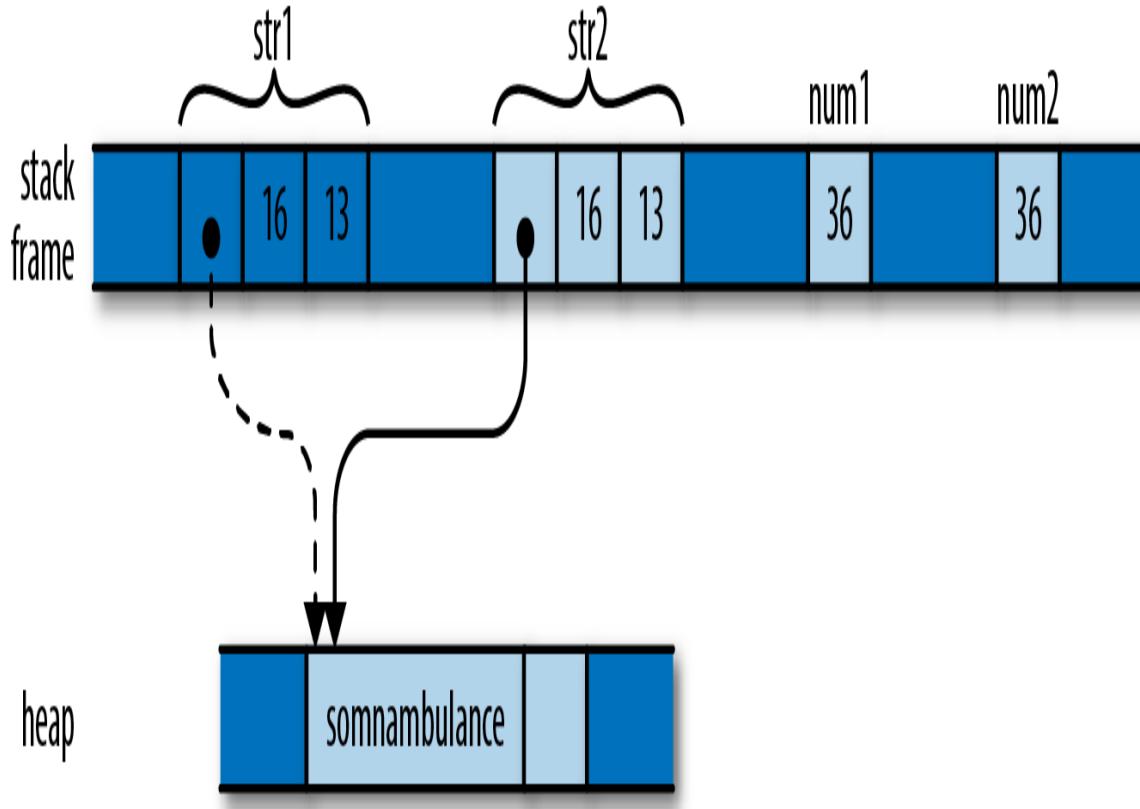


Figure 3-11. Assigning a string moves the value, whereas assigning an `i32` copies it

As with the vectors earlier, assignment *moves* `str1` to `str2`, so that we don't end up with two strings responsible for freeing the same buffer. However, the situation with `num1` and `num2` is different. An `i32` is simply a pattern of bits in memory; it doesn't own any heap resources, or really depend on anything other than the bytes it comprises. By the time we've moved its bits to `num2`, we've made a completely independent copy of `num1`.

Moving a value leaves the source of the move uninitialized. But whereas it serves an essential purpose to treat `str1` as valueless, treating `num1` that way is pointless; no harm could result from continuing to use it. The advantages of a move don't apply here, and it's inconvenient.

Earlier we were careful to say that *most* types are moved; now we've come to the exceptions, the types Rust designates as *Copy types*. Assigning a value of a *Copy type* copies the value, rather than moving it. The source of

the assignment remains initialized and usable, with the same value it had before. Passing `Copy` types to functions and constructors behaves similarly.

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

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

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

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

```
struct Label { number: u32 }

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

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

This won't compile; Rust complains:

```
error: borrow of moved value: `l`
|
10 |     let l = Label { number: 3 };
```

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

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

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

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

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

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

elicits this error:

```
error[E0204]: the trait `Copy` may not be implemented for this type
--> ownership_string_label.rs:7:10
|
7 | #[derive(Copy, Clone)]
|     ^
8 | struct StringLabel { name: String }
|                         ----- this field does not implement `Copy`
```

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

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

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

In this section, we've talked about `Copy` and `Clone` in vague terms as characteristics a type might have. They are actually examples of *traits*, Rust's open-ended facility for categorizing types based on what you can do with them. We describe traits in general in [Chapter 10](#), and `Copy` and `Clone` in particular in XREF HERE.

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 preceding code produces the situation illustrated in [Figure 3-12](#) 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.

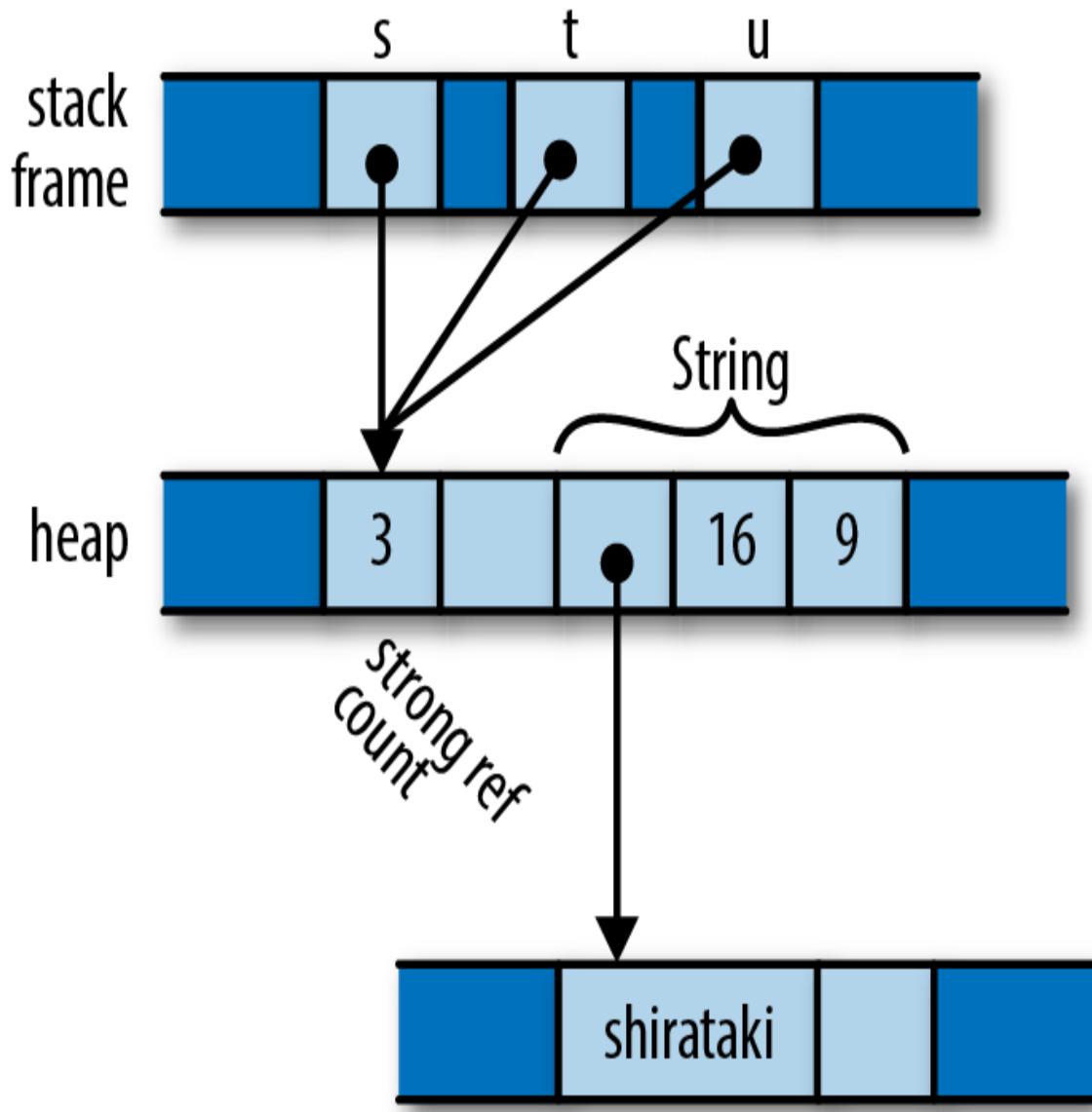


Figure 3-12. A reference-counted string, with three references

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

```
assert!(s.contains("shira"));
```

```
assert_eq!(t.find("taki"), Some(5));
println!("{} 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 data in a `&` reference as mutable
|
13 |     s.push_str(" noodles");
|     ^ cannot borrow as mutable
```

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

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

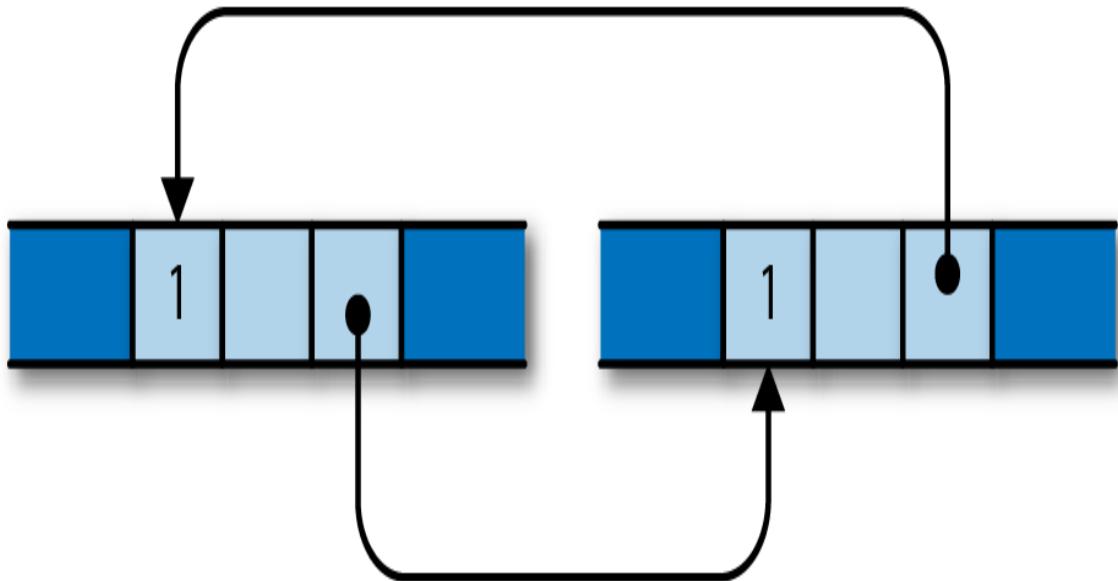


Figure 3-13. A reference-counting loop; these objects will not be freed

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

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

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

Chapter 4. References

A NOTE FOR EARLY RELEASE READERS

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

This will be the 5th chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at jimb@red-bean.com.

Libraries cannot provide new inabilities.

—Mark Miller

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 nonowning pointer types called *references*, which have no effect on their referents’ lifetimes.

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

If you felt a moment of skepticism when reading the phrase “You must make it apparent in your code,” you’re in excellent company. The references themselves are nothing special—under the hood, they’re just addresses. But the rules that keep them safe are novel to Rust; outside of

research languages, you won't have seen anything like them before. And although these rules are the part of Rust that requires the most effort to master, the breadth of classic, absolutely everyday bugs they prevent is surprising, and their effect on multithreaded programming is liberating. This is Rust's radical wager, again.

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

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

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

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

```
fn main() {
    let mut table = Table::new();
    table.insert("Gesualdo".to_string(),
                vec![ "many madrigals".to_string(),
                      "Tenebrae Responsoria".to_string()]);
    table.insert("Caravaggio".to_string(),
                vec![ "The Musicians".to_string(),
                      "The Calling of St. Matthew".to_string()]);
}
```

```

        table.insert("Cellini".to_string(),
                      vec![ "Perseus with the head of Medusa".to_string(),
                            "a salt cellar".to_string()]);
    }

    show(table);
}

```

And it all works fine:

```

$ cargo run
     Running `./home/jimb/rust/book/fragments/target/debug/fragments`
works by Gesualdo:
    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)`, 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 anymore:

```

error: borrow of moved value: `table`
|
20 |     let mut table = Table::new();
|           ----- move occurs because `table` has type `HashMap<String,

```

```

Vec<String>>,                                which does not implement the `Copy` trait
|
...
31 |     show(table);
|     ----- value moved here
32 |     assert_eq!(table["Gesualdo"][0], "many madrigals");
|             ^^^^^ value borrowed here after move

```

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

You can think of the distinction between shared and mutable references as a way to enforce a *multiple readers or single writer* rule at compile time. In fact, this rule doesn’t apply only to references; it covers the borrowed value’s owner as well. As long as there are shared references to a value, not even its owner can modify it; the value is locked down. Nobody can modify `table` while `show` is working with it. Similarly, if there is a mutable reference to a value, it has exclusive access to the value; you can’t use the

owner at all, until the mutable reference goes away. Keeping sharing and mutation fully separate turns out to be essential to memory safety, for reasons we'll go into later in the chapter.

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

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

References are nonowning 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 nonowning references.

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

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

And we need to pass it one:

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

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

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

References as Values

The preceding example shows a pretty typical use for references: allowing functions to access or manipulate a structure without taking ownership. But

references are more flexible than that, so let's look at some examples to get a more detailed view of what's going on.

Rust References Versus C++ References

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

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

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

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

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

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

```
let mut y = 32;
let m = &mut y;        // &mut y is a mutable reference to y
*m += 32;             // explicitly dereference m to set y's value
assert!(*m == 64);    // and to see y's new value
```

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

operator. Why is that?

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

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

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

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

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

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

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

Assigning References

Assigning 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 code points it at `y` instead, as illustrated in [Figure 4-1](#).

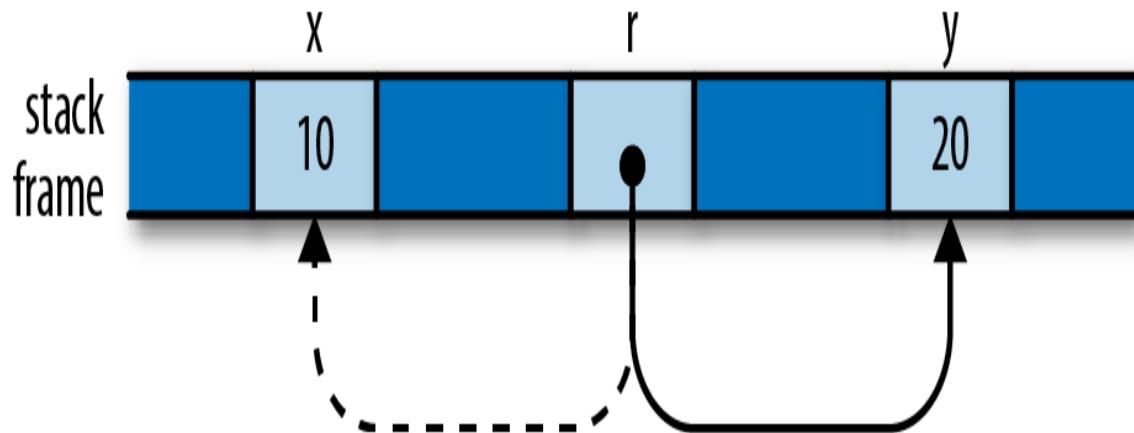


Figure 4-1. The reference `r`, now pointing to `y` instead of `x`

This is very different from C++, where assigning to a reference stores the value in its referent. There's no way to point a C++ reference to a location other than the one it was initialized with.

References to References

Rust permits references to references:

```

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

```

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

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

In memory, the references are arranged as shown in [Figure 4-2](#).

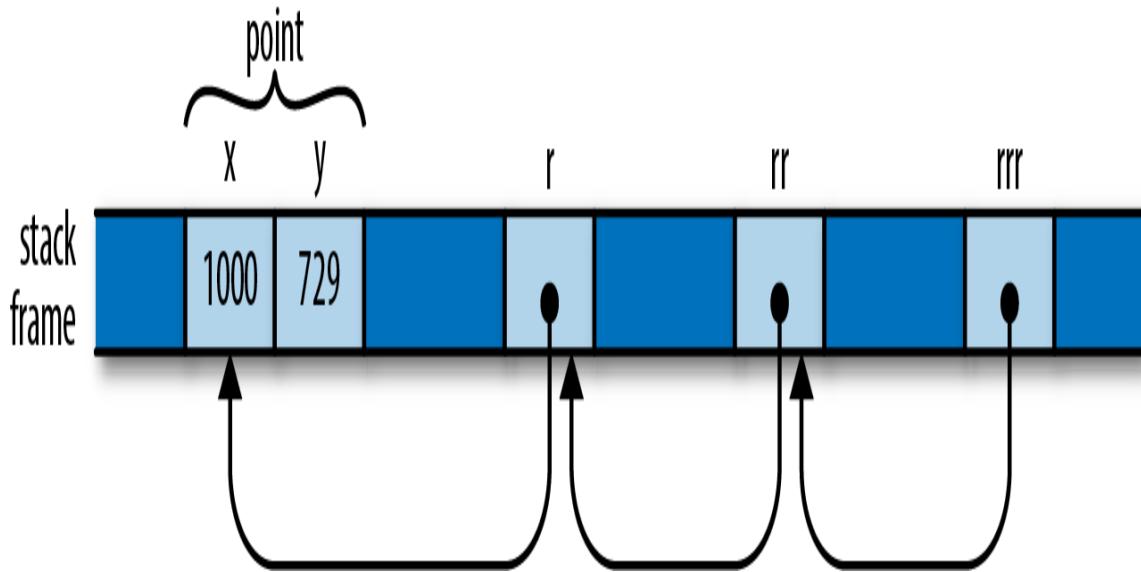


Figure 4-2. A chain of references to references

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

Comparing References

Like the `.` operator, Rust's comparison operators “see through” any number of references, 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 `rx` and `ry` point at different values (namely, `x` and `y`), because the `==` operator follows all the references and performs the comparison on their final targets, `x` and `y`. This is almost always the behavior you want, especially when writing generic functions. If you actually want to know whether two references point to the same memory, you can use `std::ptr::eq`, which compares them as addresses:

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

References Are Never Null

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

C and C++ code often uses a null pointer to indicate the absence of a value: for example, the `malloc` function 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 a `&T` value, as the nonzero address, so `Option<&T>` is just as efficient as a nullable pointer in C or C++, even though it's safer: its type requires you to check whether it's `None` before you can use it.

Borrowing References to Arbitrary Expressions

Whereas C and C++ only let you apply the `&` operator to certain kinds of expressions, Rust lets you borrow a reference to the value of any sort of expression at all:

```
fn factorial(n: usize) -> usize {
    (1..n+1).fold(1, |a, b| a * b)
}
let r = &factorial(6);
// Arithmetic operators can see through one level of references.
assert_eq!(r + &1009, 1729);
```

In situations like this, Rust simply creates an anonymous variable to hold the expression's value, and makes the reference point to that. The lifetime of this anonymous variable depends on what you do with the reference:

- If you immediately assign the reference to a variable in a `let` statement (or make it part of some struct or array that is being immediately assigned), then Rust makes the anonymous variable live as long as the variable the `let` initializes. In the preceding example, Rust would do this for the referent of `r`.
- Otherwise, the anonymous variable lives to the end of the enclosing statement. In our example, the anonymous variable created to hold `1009` lasts only to the end of the `assert_eq!` statement.

If you're used to C or C++, this may sound error-prone. But remember that Rust will never let you write code that would produce a dangling reference. If the reference could ever be used beyond the anonymous variable's lifetime, Rust will always report the problem to you at compile time. You can then fix your code to keep the referent in a named variable with an appropriate lifetime.

References to Slices and Trait Objects

The references we've shown so far are all simple addresses. However, Rust also includes two kinds of *fat pointers*, two-word values carrying the address of some value, along with some further information necessary to put the value to use.

A reference to a slice is a fat pointer, carrying the starting address of the slice and its length. We described slices in detail in [Chapter 2](#).

Rust's other kind of fat pointer is a *trait object*, a reference to a value that implements a certain trait. A trait object carries a value's address and a pointer to the trait's implementation appropriate to that value, for invoking the trait's methods. We'll cover trait objects in detail in [“Trait Objects”](#).

Aside from carrying this extra data, slice and trait object references behave just like the other sorts of references we've shown so far in this chapter: they don't own their referents; they are not allowed to outlive their referents; they may be mutable or shared; and so on.

Reference Safety

As we've presented them so far, references look pretty much like ordinary pointers in C or C++. But those are unsafe; how does Rust keep its references under control? Perhaps the best way to see the rules in action is to try to break them. 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 variable and take it out of the variable's scope:

```
{  
    let r;  
    {  
        let x = 1;  
        r = &x;  
    }  
    assert_eq!(*r, 1); // bad: reads memory `x` used to occupy  
}
```

The Rust compiler rejects this program, with a detailed error message:

```

error: `x` does not live long enough
--> references_dangling.rs:8:5
|
7 |     r = &x;
|         - borrow occurs here
8 | }
|     ^ `x` dropped here while still borrowed
9 |     assert_eq!(*r, 1); // bad: reads memory `x` used to occupy
10| }
| - borrowed value needs to live until here

```

Rust's complaint is that `x` lives only until the end of the inner block, whereas the reference remains alive until the end of the outer block, making it a dangling pointer, which is verboten.

While it's obvious to a human reader that this program is broken, it's worth looking at how Rust itself reached that conclusion. Even this simple example shows the logical tools Rust uses to check much more complex code.

Rust tries to assign each reference type in your program a *lifetime* that meets the constraints imposed by how it is used. A lifetime is some stretch of your program for which a reference could be safe to use: a statement, an expression, the scope of some variable, or the like. Lifetimes are entirely figments of Rust's compile-time imagination. At run time, a reference is nothing but an address; its lifetime is part of its type and has no run-time representation.

In this example, there are three lifetimes whose relationships we need to work out. The variables `r` and `x` each have a lifetime, extending from the point at which they're initialized until the point that they go out of scope. The third lifetime is that of a reference type: the type of the reference we borrow to `x`, and store in `r`.

Here's one constraint that should seem pretty obvious: if you have a variable `x`, then a reference to `x` must not outlive `x` itself, as shown in [Figure 4-3](#).

```
{  
    let r;  
    {  
        let x = 1;  
        ...  
        r = &x;  
        ...  
    }  
    assert_eq!(*r, 1);  
}
```

lifetime of `&x` must not exceed this range

Figure 4-3. Permissible lifetimes for `&x`

Beyond the point where `x` goes out of scope, the reference would be a dangling pointer. We say that the variable's lifetime must *contain* or *enclose* that of the reference borrowed from it.

Here's another kind of constraint: if you store a reference in a variable `r`, the reference's type must be good for the entire lifetime of the variable, from the point it is initialized to the point it goes out of scope, as shown in Figure 4-4.

```
{  
    let r;  
    {  
        let x = 1;  
        ...  
        r = &x;  
        ...  
    }  
    assert_eq!(*r, 1);  
}
```

lifetime of anything stored in
r must cover at least this range

Figure 4-4. Permissible lifetimes for reference stored in *r*

If the reference can't live at least as long as the variable does, then at some point *r* will be a dangling pointer. We say that the reference's lifetime must contain or enclose the variable's.

The first kind of constraint limits how large a reference's lifetime can be, while the second kind limits how small it can be. Rust simply tries to find a lifetime for each reference that satisfies all these constraints. In our example, however, there is no such lifetime, as shown in [Figure 4-5](#).

```
{  
    let r;  
    {  
        let x = 1;  
        ...  
        r = &x;  
        ...  
    }  
    assert_eq!(*r, 1);  
}
```

There is no lifetime that lies entirely within this range...

...but also fully encloses this range.

Figure 4-5. A reference with contradictory constraints on its lifetime

Let's now consider a different example where things do work out. We have the same kinds of constraints: the reference's lifetime must be contained by `x`'s, but fully enclose `r`'s. But because `r`'s lifetime is smaller now, there is a lifetime that meets the constraints, as shown in [Figure 4-6](#).

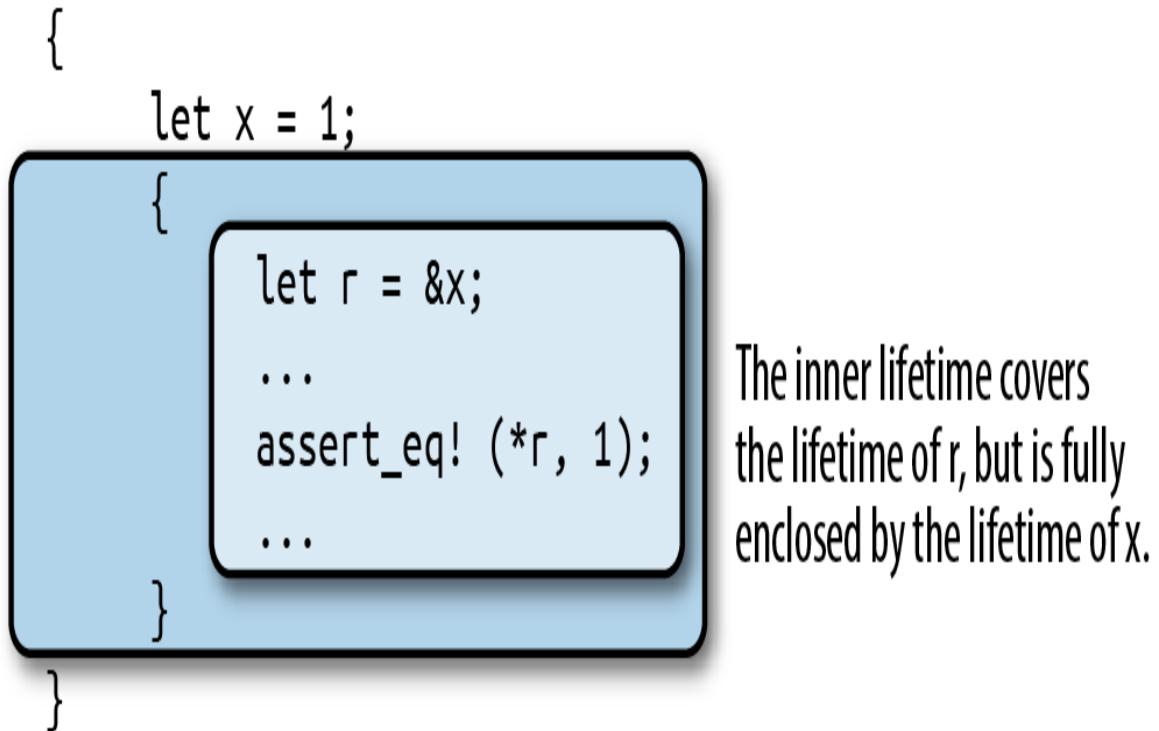


Figure 4-6. A reference with a lifetime enclosing r's scope, but within x's scope

These rules apply in a natural way when you borrow a reference to some part of some larger data structure, like an element of a vector:

```

let v = vec![1, 2, 3];
let r = &v[1];

```

Since `v` owns the vector, which owns its elements, the lifetime of `v` must enclose that of the reference type of `&v[1]`. Similarly, if you store a reference in some data structure, its lifetime must enclose that of the data structure. If you build a vector of references, say, all of them must have lifetimes enclosing that of the variable that owns the vector.

This is the essence of the process Rust uses for all code. Bringing more language features into the picture—data structures and function calls, say—introduces new sorts of constraints, but the principle remains the same: first, understand the constraints arising from the way the program uses references; then, find lifetimes that satisfy them. This is not so different

from the process C and C++ programmers impose on themselves; the difference is that Rust knows the rules, and enforces them.

Receiving References as 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 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 the code just shown doesn't follow:

- Every static must be initialized.
- Mutable statics are inherently not thread-safe (after all, any thread can access a static at any time), and even in single-threaded programs, they can fall prey to other sorts of reentrancy problems. For these reasons, you may access a mutable static only within an `unsafe` block. In this example we're not concerned with those particular problems, so we'll just throw in an `unsafe` block and move on.

With those revisions made, we now have the following:

```
static mut STASH: &i32 = &128;
fn f(p: &i32) { // still not good enough
    unsafe {
```

```
        STASH = p;  
    }  
}
```

We're almost done. To see the remaining problem, we need to write out a few things that Rust is helpfully letting us omit. The signature of `f` as written here is actually shorthand for the following:

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

Here, the lifetime '`a` (pronounced “tick A”) is a *lifetime parameter* of `f`. You can read `<'a>` as “for any lifetime '`a`” so when we write `fn f<'a>(p: &'a i32)`, we’re defining a function that takes a reference to an `i32` with any given lifetime '`a`.

Since we must allow '`a` to be any lifetime, things had better work out if it’s the smallest possible lifetime: one just enclosing the call to `f`. This assignment then becomes a point of contention:

```
STASH = p;
```

Since `STASH` lives for the program’s entire execution, the reference type it holds must have a lifetime of the same length; Rust calls this the *'static lifetime*. But the lifetime of `p`’s reference is some '`a`, which could be anything, as long as it encloses the call to `f`. So, Rust rejects our code:

```
error[E0312]: lifetime of reference outlives lifetime of borrowed content...  
--> references_static.rs:6:17  
|  
6 |     STASH = p;  
|     ^  
|  
= note: ...the reference is valid for the static lifetime...  
note: ...but the borrowed content is only valid for the anonymous lifetime #1  
      defined on the function body at 4:0  
--> references_static.rs:4:1  
|
```

```
4 | / fn f(p: &i32) { // still not good enough
5 | |     unsafe {
6 | |         STASH = p;
7 | |     }
8 | | }
```

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: &i32 = &10;

fn f(p: &'static i32) {
    unsafe {
        STASH = p;
    }
}
```

This time, `f`'s signature spells out that `p` must be a reference with lifetime '`'static`', so there's no longer any problem storing that in `STASH`. We can only apply `f` to references to other statics, but that's the only thing that's certain not to leave `STASH` dangling anyway. So we can write:

```
static WORTH_POINTING_AT: i32 = 1000;
f(&WORTH_POINTING_AT);
```

Since `WORTH_POINTING_AT` is a static, the type of `&WORTH_POINTING_AT` is `&'static i32`, which is safe to pass to `f`.

Take a step back, though, and notice what happened to `f`'s signature as we amended our way to correctness: the original `f(p: &i32)` ended up as `f(p: &'static i32)`. In other words, we were unable to write a function that stashed a reference in a global variable without reflecting that intention in the function's signature. In Rust, a function's signature always exposes the body's behavior.

Conversely, if we do see a function with a signature like `g(p: &i32)` (or with the lifetimes written out, `g<'a>(p: &'a i32)`), we can tell that it *does not* stash its argument `p` anywhere that will outlive the call. There's no need to look into `g`'s definition; the signature alone tells us what `g` can and can't do with its argument. This fact ends up being very useful when you're trying to establish the safety of a call to the function.

Passing References 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 encloses the call must work for `'a`. So Rust chooses the smallest possible lifetime for `&x`: that of the call to `g`. This meets all constraints: it doesn't outlive `x`, and encloses the entire call to `g`. So this code passes muster.

Note that although `g` takes a lifetime parameter `'a`, we didn't need to mention it when calling `g`. You only need to worry about lifetime parameters when defining functions and types; when using them, Rust infers the lifetimes for you.

What if we tried to pass `&x` to our function `f` from earlier that stores its argument in a static?

```
fn f(p: &'static i32) { ... }
```

```
let x = 10;
f(&x);
```

This fails to compile: the reference `&x` must not outlive `x`, but by passing it to `f`, we constrain it to live at least as long as `'static`. There's no way to satisfy everyone here, so Rust rejects the code.

Returning References

It's common for a function to take a reference to some data structure, and then return a reference into some part of that structure. For example, here's a function that returns a reference to the smallest element of a slice:

```
// v should have at least one element.
fn smallest(v: &[i32]) -> &i32 {
    let mut s = &v[0];
    for r in &v[1..] {
        if *r < *s { s = r; }
    }
    s
}
```

We've omitted lifetimes from that function's signature in the usual way. When a function takes a single reference as an argument, and returns a single reference, Rust assumes that the two must have the same lifetime. Writing this out explicitly would give us:

```
fn smallest<'a>(v: &'a [i32]) -> &'a i32 { ... }
```

Suppose we call `smallest` like this:

```
let s;
{
    let parabola = [9, 4, 1, 0, 1, 4, 9];
    s = smallest(&parabola);
}
assert_eq!(*s, 0); // bad: points to element of dropped array
```

From `smallest`'s signature, we can see that its argument and return value must have the same lifetime, '`a`'. In our call, the argument `¶bola` 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
--> references_lifetimes_propagated.rs:12:5
|
11 |         s = smallest(&parabola);
|                         ----- borrow occurs here
12 |
13 |     }
|     ^ `parabola` dropped here while still borrowed
14 |     assert_eq!(*s, 0); // bad: points to element of dropped array
|   }
| - borrowed value needs to live until here
```

Moving `s` so that its lifetime is clearly contained within `parabola`'s fixes the problem:

```
{
    let parabola = [9, 4, 1, 0, 1, 4, 9];
    let s = smallest(&parabola);
    assert_eq!(*s, 0); // fine: parabola still alive
}
```

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

Structs Containing References

How does Rust handle references stored in data structures? Here's the same erroneous program we looked at earlier, except that we've put the reference inside a structure:

```
// This does not compile.
```

```

struct S {
    r: &i32
}

let s;
{
    let x = 10;
    s = S { r: &x };
}
assert_eq!(*s.r, 10); // bad: reads from dropped `x`

```

The safety constraints Rust places on references can't magically disappear just because we hid the reference inside a struct. Somehow, those constraints must end up applying to `S` as well. Indeed, Rust is skeptical:

```

error[E0106]: missing lifetime specifier
--> references_in_struct.rs:7:12
 |
7 |     r: &i32
 |     ^ expected lifetime parameter

```

Whenever a reference type appears inside another type's definition, you must write out its lifetime. You can write this:

```

struct S {
    r: &'static i32
}

```

This says that `r` can only refer to `i32` values that will last for the lifetime of the program, which is rather limiting. The alternative is to give the type a lifetime parameter '`a`', and use that for `r`:

```

struct S<'a> {
    r: &'a i32
}

```

Now the `S` type has a lifetime, just as reference types do. Each value you create of type `S` gets a fresh lifetime '`a`', which becomes constrained by how

you use the value. The lifetime of any reference you store in `r` had better enclose '`a`', and '`a`' must outlast the lifetime of wherever you store the `S`.

Turning back to the preceding code, the expression `S { r: &x }` creates a fresh `S` value with some lifetime '`a`'. When you store `&x` in the `r` field, you constrain '`a`' to lie entirely within `x`'s lifetime.

The assignment `s = S { ... }` stores this `S` in a variable whose lifetime extends to the end of the example, constraining '`a`' to outlast the lifetime of `s`. And now Rust has arrived at the same contradictory constraints as before: '`a`' must not outlive `x`, yet must live at least as long as `s`. No satisfactory lifetime exists, and Rust rejects the code. Disaster averted!

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

```
struct 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[E0106]: missing lifetime specifier
--> references_in_nested_struct.rs:8:8
 |
8 |     s: S // not adequate
|           ^ expected lifetime parameter
```

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 parse_record<'i>(input: &'i [u8]) -> Record<'i> { ... }
```

Without looking into the definition of the `Record` type at all, we can tell that, if we receive a `Record` from `parse_record`, whatever references it contains must point into the input buffer we passed in, and nowhere else (except perhaps at `'static` values).

In fact, this exposure of internal behavior is the reason Rust requires types that contain references to take explicit lifetime parameters. There's no reason Rust couldn't simply make up a distinct lifetime for each reference in the struct, and save you the trouble of writing them out. Early versions of

Rust actually behaved this way, but developers found it confusing: it is helpful to know when one value borrows something from another value, especially when working through errors.

It's not just references and types like `S` that have lifetimes. Every type in Rust has a lifetime, including `i32` and `String`. Most are simply `'static`, meaning that values of those types can live for as long as you like; for example, a `Vec<i32>` is self-contained, and needn't be dropped before any particular variable goes out of scope. But a type like `Vec<&'a i32>` has a lifetime that must be enclosed by `'a`: it must be dropped while its referents are still alive.

Distinct Lifetime Parameters

Suppose you've defined a structure containing two references like this:

```
struct S<'a> {
    x: &'a i32,
    y: &'a i32
}
```

Both references use the same lifetime `'a`. This could be a problem if your code wants to do something like this:

```
let x = 10;
let r;
{
    let y = 20;
    {
        let s = S { x: &x, y: &y };
        r = s.x;
    }
}
```

This code doesn't create any dangling pointers. The reference to `y` stays in `s`, which goes out of scope before `y` does. The reference to `x` ends up in `r`, which doesn't outlive `x`.

If you try to compile this, however, Rust will complain that `y` does not live long enough, even though it clearly does. Why is Rust worried? If you work through the code carefully, you can follow its reasoning:

- Both fields of `S` are references with the same lifetime '`a`, so Rust must find a single lifetime that works for both `s.x` and `s.y`.
- We assign `r = s.x`, requiring '`a` to enclose `r`'s lifetime.
- We initialized `s.y` with `&y`, requiring '`a` to be no longer than `y`'s lifetime.

These constraints are impossible to satisfy: no lifetime is shorter than `y`'s scope, but longer than `r`'s. Rust balks.

The problem arises because both references in `S` have the same lifetime '`a`. Changing the definition of `S` to let each reference have a distinct lifetime fixes everything:

```
struct S<'a, 'b> {
    x: &'a i32,
    y: &'b i32
}
```

With this definition, `s.x` and `s.y` have independent lifetimes. What we do with `s.x` has no effect on what we store in `s.y`, so it's easy to satisfy the constraints now: '`a` can simply be `r`'s lifetime, and '`b` can be `s`'s. (`y`'s lifetime would work too for '`b`, but Rust tries to choose the smallest lifetime that works.) Everything ends up fine.

Function signatures can have similar effects. Suppose we have a function like this:

```
fn f<'a>(r: &'a i32, s: &'a i32) -> &'a i32 { r } // perhaps too tight
```

Here, both reference parameters use the same lifetime '`a`, which can unnecessarily constrain the caller in the same way we've shown previously.

If this is a problem, you can let parameters' lifetimes vary independently:

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

The downside to this is that adding lifetimes can make types and function signatures harder to read. Your authors tend to try the simplest possible definition first, and then loosen restrictions until the code compiles. Since Rust won't permit the code to run unless it's safe, simply waiting to be told when there's a problem is a perfectly acceptable tactic.

Omitting Lifetime Parameters

We've shown plenty of functions so far in this book that return references or take them as parameters, but we've usually not needed to spell out which lifetime is which. The lifetimes are there; Rust is just letting us omit them when it's reasonably obvious what they should be.

In the simplest cases, you may never need to write out lifetimes for your parameters. Rust just assigns a distinct lifetime to each spot that needs one. For example:

```
struct S<'a, 'b> {
    x: &'a i32,
    y: &'b i32
}

fn sum_r_xy(r: &i32, s: S) -> i32 {
    r + s.x + s.y
}
```

This function's signature is shorthand for:

```
fn sum_r_xy<'a, 'b, 'c>(r: &'a i32, s: S<'b, 'c>) -> i32
```

If you do return references or other types with lifetime parameters, Rust still tries to make the unambiguous cases easy. If there's only a single lifetime

that appears among your function's parameters, then Rust assumes any lifetimes in your return value must be that one:

```
fn first_third(point: &[i32; 3]) -> (&i32, &i32) {
    (&point[0], &point[2])
}
```

With all the lifetimes written out, the equivalent would be:

```
fn first_third<'a>(point: &'a [i32; 3]) -> (&'a i32, &'a i32)
```

If there are multiple lifetimes among your parameters, then there's no natural reason to prefer one over the other for the return value, and Rust makes you spell out what's going on.

If your function is a method on some type and takes its `self` parameter by reference, then that breaks the tie: Rust assumes that `self`'s lifetime is the one to give everything in your return value. (A `self` parameter refers to the value the method is being called on, Rust's equivalent of `this` in C++, Java, or JavaScript, or `self` in Python. We'll cover methods in “[Defining Methods with impl](#)”.)

For example, you can write the following:

```
struct StringTable {
    elements: Vec<String>,
}

impl StringTable {
    fn find_by_prefix(&self, prefix: &str) -> Option<&String> {
        for i in 0 .. self.elements.len() {
            if self.elements[i].starts_with(prefix) {
                return Some(&self.elements[i]);
            }
        }
        None
    }
}
```

The `find_by_prefix` method's signature is shorthand for:

```
fn find_by_prefix<'a, 'b>(&'a self, prefix: &'b str) -> Option<&'a String>
```

Rust assumes that whatever you're borrowing, you're borrowing from `self`.

Again, these are just abbreviations, meant to be helpful without introducing surprises. When they're not what you want, you can always write the lifetimes out explicitly.

Sharing Versus Mutation

So far, we've discussed how Rust ensures no reference will ever point to a variable that has gone out of scope. But there are other ways to introduce dangling pointers. Here's an easy case:

```
let v = vec![4, 8, 19, 27, 34, 10];
let r = &v;
let aside = v; // move vector to aside
r[0];          // bad: uses `v`, which is now uninitialized
```

The assignment to `aside` moves the vector, leaving `v` uninitialized, turning `r` into a dangling pointer, as shown in [Figure 4-7](#).

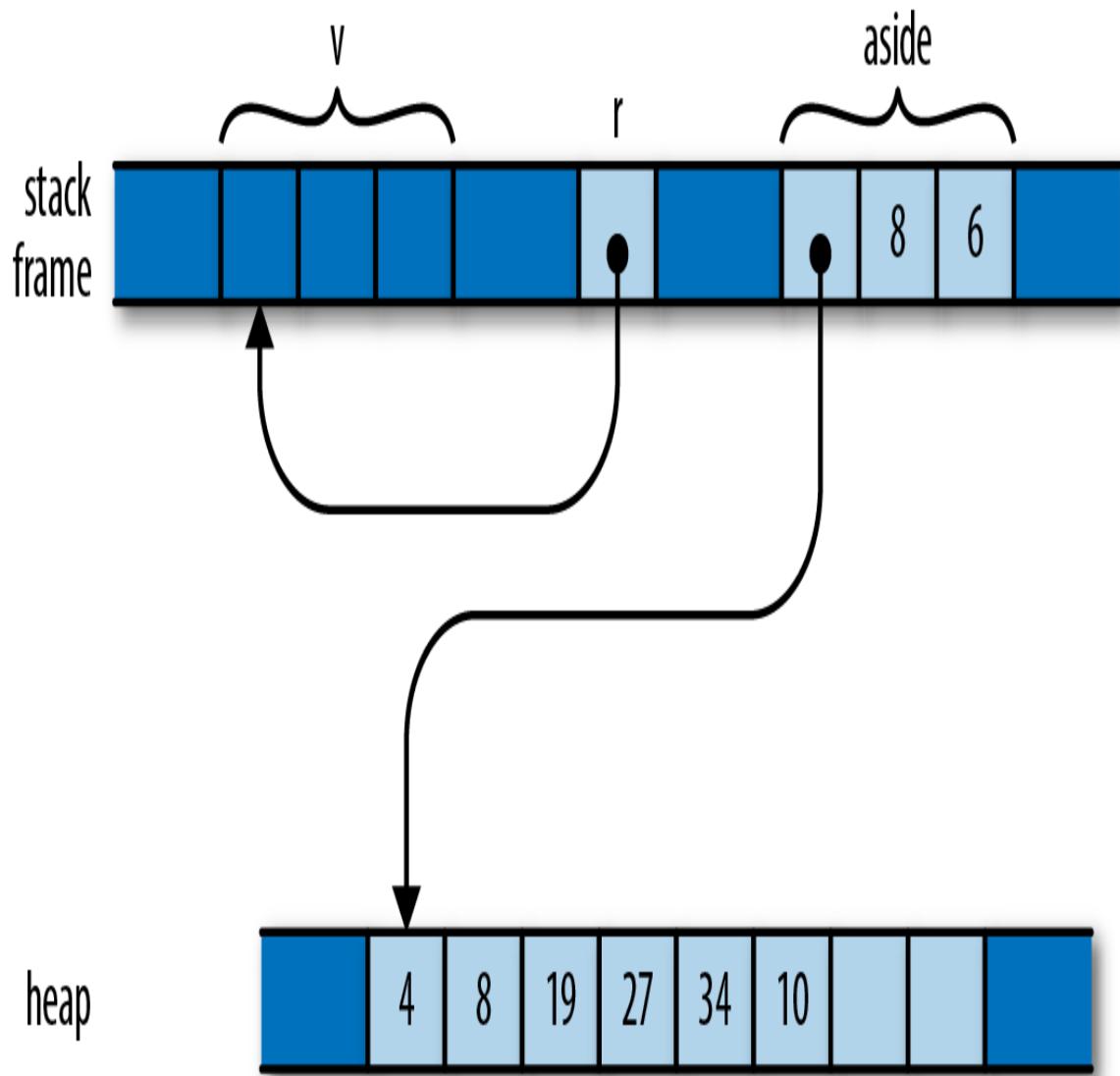


Figure 4-7. A reference to a vector that has been moved away

Although `v` stays in scope for `r`'s entire lifetime, the problem here is that `v`'s value gets moved elsewhere, leaving `v` uninitialized while `r` still refers to it. Naturally, Rust catches the error:

```
error[E0505]: cannot move out of `v` because it is borrowed
--> references_sharing_vs_mutation_1.rs:10:9
|
9 |     let r = &v;
|         - borrow of `v` occurs here
10 |     let aside = v; // move vector to aside
|             ^^^^^ move out of `v` occurs here
```

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

```
let v = vec![4, 8, 19, 27, 34, 10];
{
    let r = &v;
    r[0];      // ok: vector is still there
}
let aside = v;
```

In this version, `r` goes out of scope earlier, the reference's lifetime ends before `v` is moved aside, and all is well.

Here's a different way to wreak havoc. Suppose we have a handy function to extend a vector with the elements of a slice:

```
fn extend(vec: &mut Vec<f64>, slice: &[f64]) {
    for elt in slice {
        vec.push(*elt);
    }
}
```

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

```
let mut wave = Vec::new();
let head = vec![0.0, 1.0];
let tail = [0.0, -1.0];

extend(&mut wave, &head);    // extend wave with another vector
extend(&mut wave, &tail);    // extend wave with an array

assert_eq!(wave, vec![0.0, 1.0, 0.0, -1.0]);
```

So we've built up one period of a sine wave here. If we want to add another undulation, can we append the vector to itself?

```
extend(&mut wave, &wave);
assert_eq!(wave, vec![0.0, 1.0, 0.0, -1.0,
                     0.0, 1.0, 0.0, -1.0]);
```

This may look fine on casual inspection. But remember that when we add an element to a vector, if its buffer is full, it must allocate a new buffer with more space. Suppose `wave` starts with space for four elements, and so must allocate a larger buffer when `extend` tries to add a fifth. Memory ends up looking like [Figure 4-8](#).

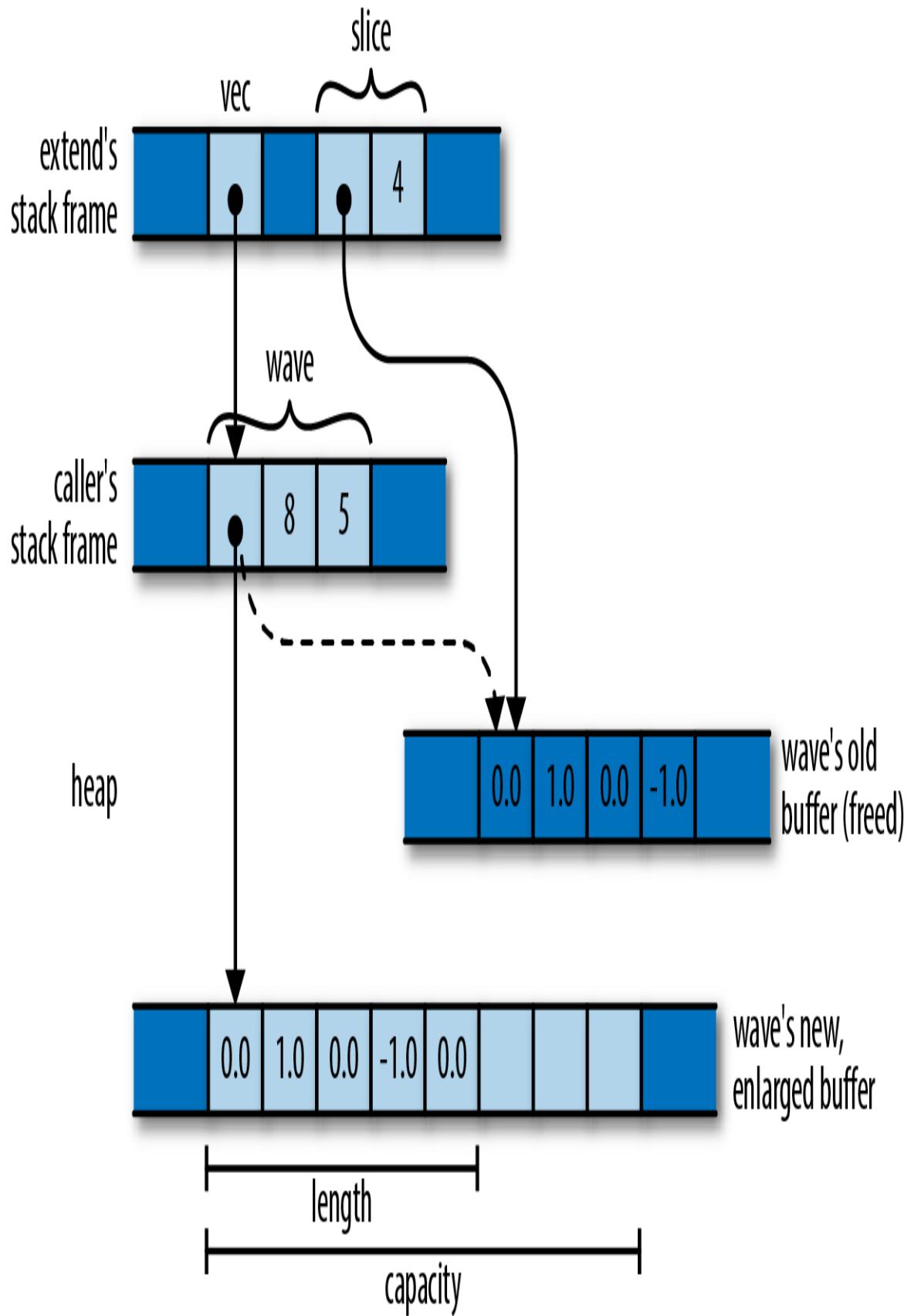


Figure 4-8. A slice turned into a dangling pointer by a vector reallocation

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 `std::vector` specification cautions you that “reallocation [of the vector’s buffer] invalidates all the references, pointers, and iterators referring to the elements in the sequence.” Similarly, Java says, of modifying a `java.util.Hashtable` object:

[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.

What’s especially difficult about this sort of bug is that it doesn’t happen all the time. In testing, your vector might always happen to have enough space, the buffer might never be reallocated, and the problem might never come to light.

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

```
error[E0502]: cannot borrow `wave` as immutable because it is also
borrowed as mutable
--> references_sharing_vs_mutation_2.rs:9:24
 |
9 |     extend(&mut wave, &wave);
|           ----- ^^^^- mutable borrow ends here
|           |
|           |           immutable borrow occurs here
|           mutable borrow occurs here
```

In other words, we may borrow a mutable reference to the vector, and we may borrow a shared reference to its elements, but those two references’ lifetimes may not overlap. In our case, both references’ lifetimes contain the call to `extend`, so Rust rejects the code.

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

- *Shared access is read-only access.* Values borrowed by shared references are read-only. Across the lifetime of a shared reference, neither its referent, nor anything reachable from that referent, can be changed *by anything*. There exist no live mutable references to anything in that structure; its owner is held read-only; and so on. It's really frozen.
- *Mutable access is exclusive access.* A value borrowed by a mutable reference is reachable exclusively via that reference. Across the lifetime of a mutable reference, there is no other usable path to its referent, or to any value reachable from there. The only references whose lifetimes may overlap with a mutable reference are those you borrow from the mutable reference itself.

Rust reported the `extend` example as a violation of the second rule: since we've borrowed a mutable reference to `wave`, that mutable reference must be the only way to reach the vector or its elements. The shared reference to the slice is itself another way to reach the elements, violating the second rule.

But Rust could also have treated our bug as a violation of the first rule: since we've borrowed a shared reference to `wave`'s elements, the elements and the `Vec` itself are all read-only. You can't borrow a mutable reference to a read-only value.

Each kind of reference affects what we can do with the values along the owning path to the referent, and the values reachable from the referent ([Figure 4-9](#)).

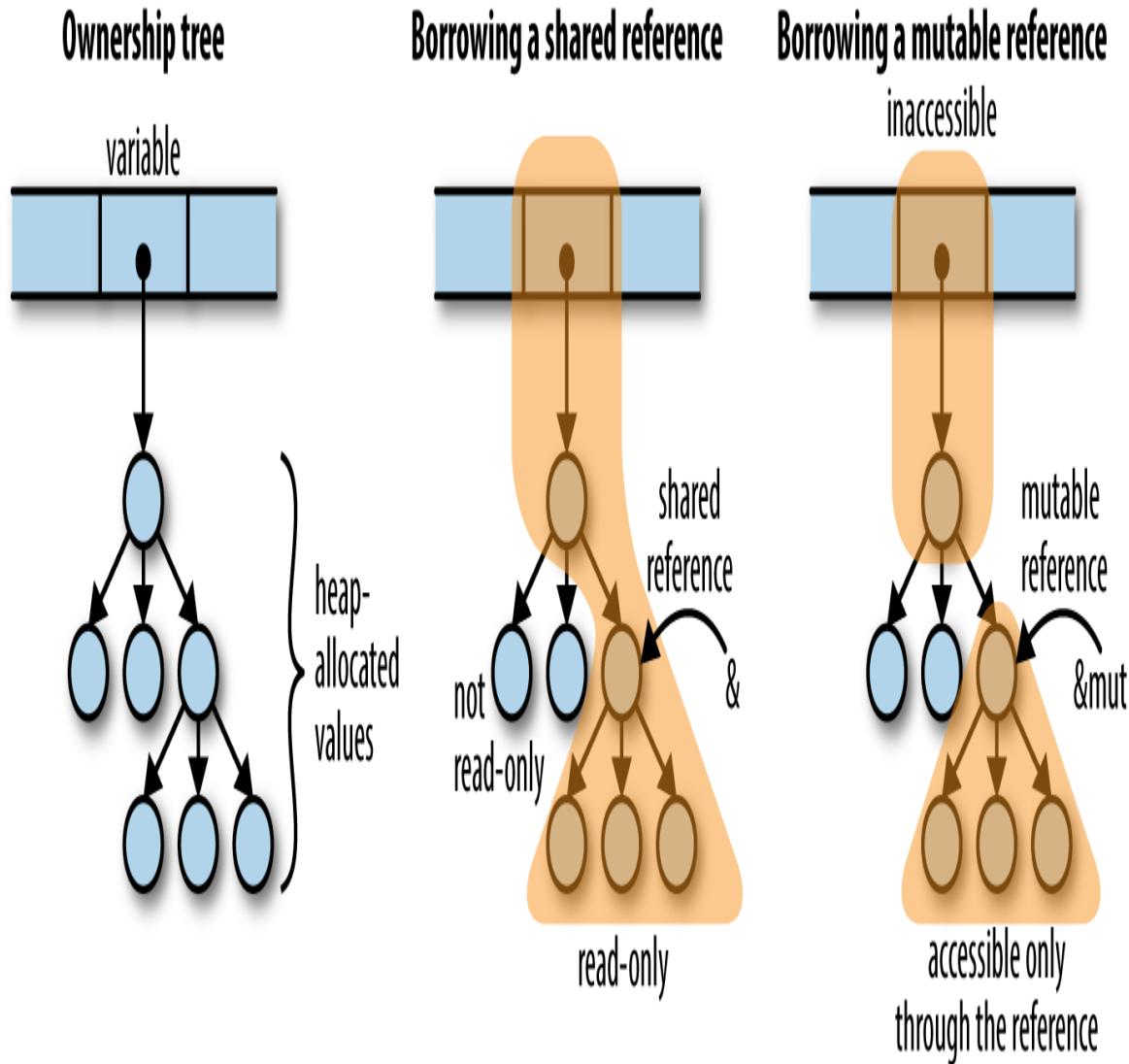


Figure 4-9. Borrowing a reference affects what you can do with other values in the same ownership tree

Note that in both cases, the path of ownership leading to the referent cannot be changed for the reference's lifetime. For a shared borrow, the path is read-only; for a mutable borrow, it's completely inaccessible. So there's no way for the program to do anything that will invalidate the reference.

Paring these principles down to the simplest possible examples:

```
let mut x = 10;
let r1 = &x;
let r2 = &x;      // ok: multiple shared borrows permitted
x += 10;         // error: cannot assign to `x` because it is borrowed
```

```

let m = &mut x; // error: cannot borrow `x` as mutable because it is
                // also borrowed as immutable
println!("{}, {}, {}", r1, r2, m); // the references are used here,
                                // so their lifetimes must last
                                // at least this long

let mut y = 20;
let m1 = &mut y;
let m2 = &mut y; // error: cannot borrow as mutable more than once
let z = y;        // error: cannot use `y` because it was mutably borrowed
println!("{}, {}, {}", m1, m2, z); // both references must be valid here

```

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

```

let mut w = (107, 109);
let r = &w;
let r0 = &r.0;           // ok: reborrowing shared as shared
let m1 = &mut r.1;       // error: can't reborrow shared as mutable
println!("{}", r0);      // r0 must be live at least this long

```

You can reborrow from a mutable reference:

```

let mut v = (136, 139);
let m = &mut v;
let m0 = &mut m.0;       // ok: reborrowing mutable from mutable
*m0 = 137;
let r1 = &m.1;           // ok: reborrowing shared from mutable,
                        // and doesn't overlap with m0
v.1;                   // error: access through other paths still forbidden
println!("{}", r1);      // r1 must live at least this long

```

These restrictions are pretty tight. Turning back to our attempted call `extend(&mut wave, &wave)`, there's no quick and easy way to fix up the code to work the way we'd like. And Rust applies these rules everywhere: if we borrow, say, a shared reference to a key in a `HashMap`, we can't borrow a mutable reference to the `HashMap` until the shared reference's lifetime ends.

But there's good justification for this: designing collections to support unrestricted, simultaneous iteration and modification is difficult, and often

precludes simpler, more efficient implementations. Java's `Hashtable` and C++'s `vector` don't bother, and neither Python dictionaries nor JavaScript objects define exactly how such access behaves. Other collection types in JavaScript do, but require heavier implementations as a result. C++'s `std::map` promises that inserting new entries doesn't invalidate pointers to other entries in the map, but by making that promise, the standard precludes more cache-efficient designs like Rust's `BTreeMap`, which stores multiple entries in each node of the tree.

Here's another example of the kind of bug these rules catch. Consider the following C++ code, meant to manage a file descriptor. To keep things simple, we're only going to show a constructor and a copying assignment operator, and we're going to omit error handling:

```
struct File {
    int descriptor;

    File(int d) : descriptor(d) { }

    File& operator=(const File &rhs) {
        close(descriptor);
        descriptor = dup(rhs.descriptor);
        return *this;
    }
};
```

The assignment operator is simple enough, but fails badly in a situation like this:

```
File f(open("foo.txt", ...));
...
f = f;
```

If we assign a `File` to itself, both `rhs` and `*this` are the same object, so `operator=` closes the very file descriptor it's about to pass to `dup`. We destroy the same resource we were meant to copy.

In Rust, the analogous code would be:

```

struct File {
    descriptor: i32
}

fn new_file(d: i32) -> File {
    File { descriptor: d }
}

fn clone_from(this: &mut File, rhs: &File) {
    close(this.descriptor);
    this.descriptor = dup(rhs.descriptor);
}

```

(This is not idiomatic Rust. There are excellent ways to give Rust types their own constructor functions and methods, which we describe in [Chapter 8](#), but the preceding definitions work for this example.)

If we write the Rust code corresponding to the use of `File`, we get:

```

let mut f = new_file(open("foo.txt", ...));
...
clone_from(&mut f, &f);

```

Rust, of course, refuses to even compile this code:

```

error[E0502]: cannot borrow `f` as immutable because it is also
borrowed as mutable
--> references_self_assignment.rs:18:25
 |
18 |     clone_from(&mut f, &f);
|             ^- mutable borrow ends here
|             |
|             |     immutable borrow occurs here
|     mutable borrow occurs here

```

This should look familiar. It turns out that two classic C++ bugs—failure to cope with self-assignment, and using invalidated iterators—are the same underlying kind of bug! In both cases, code assumes it is modifying one value while consulting another, when in fact they’re both the same value. If

you've ever accidentally let the source and destination of a call to `memcpy` or `strcpy` call overlap in C or C++, that's yet another form the bug can take. By requiring mutable access to be exclusive, Rust has fended off a wide class of everyday mistakes.

The immiscibility of shared and mutable references really demonstrates its value when writing concurrent code. A data race is possible only when some value is both mutable and shared between threads—which is exactly what Rust's reference rules eliminate. A concurrent Rust program that avoids `unsafe` code is free of data races *by construction*. We'll cover this aspect in more detail when we talk about concurrency in XREF HERE, but in summary, concurrency is much easier to use in Rust than in most other languages.

RUST'S SHARED REFERENCES VERSUS C'S POINTERS TO CONST

On first inspection, Rust's shared references seem to closely resemble C and C++'s pointers to `const` values. However, Rust's rules for shared references are much stricter. For example, consider the following C code:

```
int x = 42;           // int variable, not const
const int *p = &x;    // pointer to const int
assert(*p == 42);
x++;                // change variable directly
assert(*p == 43);    // "constant" referent's value has changed
```

The fact that `p` is a `const int *` means that you can't modify its referent via `p` itself: `(*p)++` is forbidden. But you can also get at the referent directly as `x`, which is not `const`, and change its value that way. The C family's `const` keyword has its uses, but constant it is not.

In Rust, a shared reference forbids all modifications to its referent, until its lifetime ends:

```
let mut x = 42;        // non-const i32 variable
let p = &x;            // shared reference to i32
assert_eq!(*p, 42);
x += 1;               // error: cannot assign to x because it is
                      // borrowed
assert_eq!(*p, 42);    // if you take out the assignment, this is true
```

To ensure a value is constant, we need to keep track of all possible paths to that value, and make sure that they either don't permit modification or cannot be used at all. C and C++ pointers are too unrestricted for the compiler to check this. Rust's references are always tied to a particular lifetime, making it feasible to check them at compile time.

Taking Arms Against a Sea of Objects

Since the rise of automatic memory management in the 1990s, the default architecture of all programs has been the *sea of objects*, shown in Figure 4-10.

This is what happens if you have garbage collection and you start writing a program without designing anything. We've all built systems that look like this.

This architecture has many advantages that don't show up in the diagram: initial progress is rapid, it's easy to hack stuff in, and a few years down the road, you'll have no difficulty justifying a complete rewrite. (Cue AC/DC's "Highway to Hell.")

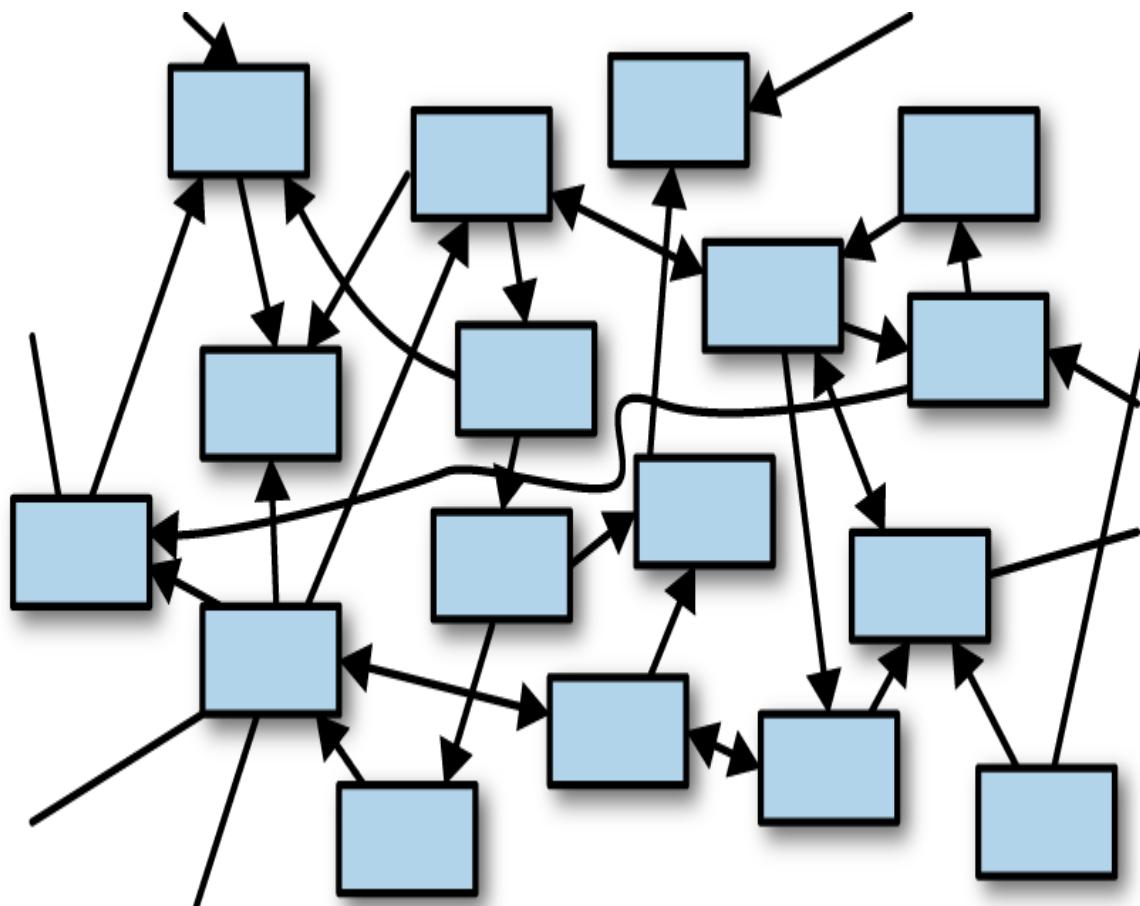


Figure 4-10. A sea of objects

Of course, there are disadvantages too. When everything depends on everything else like this, it's hard to test, evolve, or even think about any component in isolation.

One fascinating thing about Rust is that the ownership model puts a speed bump on the highway to hell. It takes a bit of effort to make a cycle in Rust —two values such that each one contains a reference pointing to the other. You have to use a smart pointer type, such as `Rc`, and **interior mutability**—a topic we haven't even covered yet. Rust prefers for pointers, ownership, and data flow to pass through the system in one direction, as shown in [Figure 4-11](#).

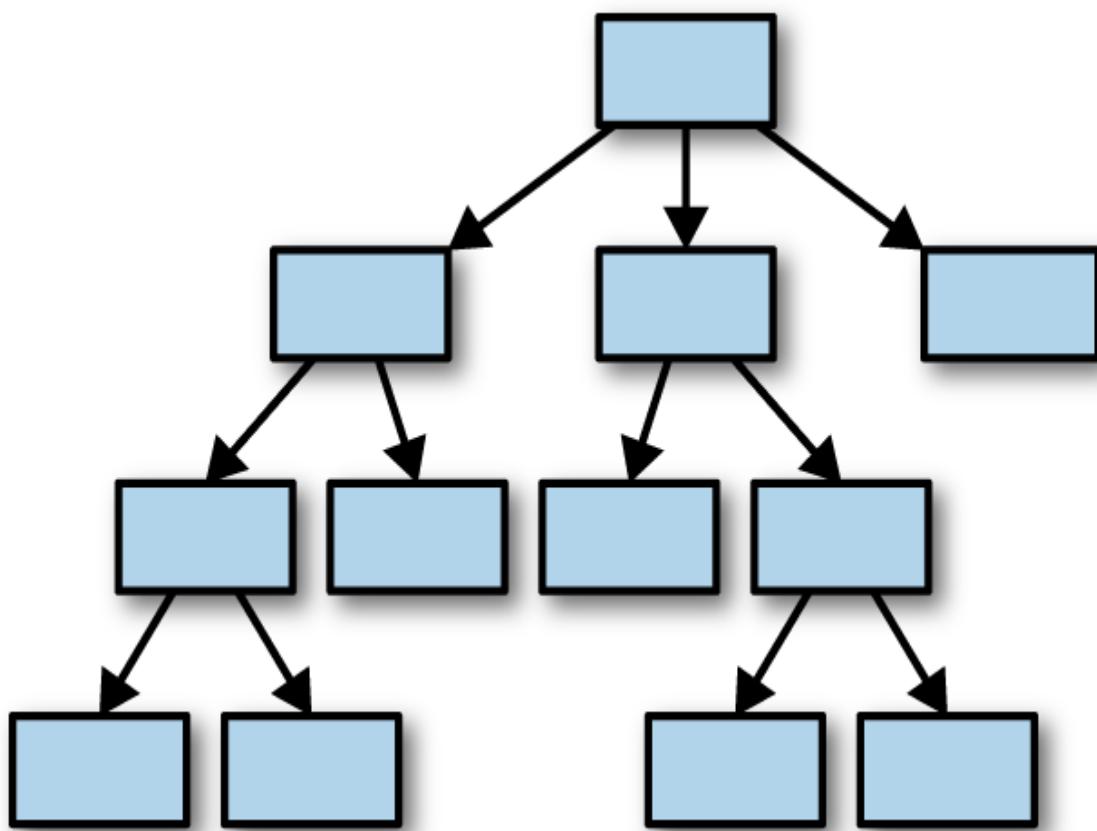


Figure 4-11. A tree of values

The reason we bring this up right now is that it would be natural, after reading this chapter, to want to run right out and create a “sea of structs,” all tied together with `Rc` smart pointers, and re-create all the object-oriented antipatterns you’re familiar with. This won’t work for you right away.

Rust's ownership model will give you some trouble. The cure is to do some up-front design and build a better program.

Rust is all about transferring the pain of understanding your program from the future to the present. It works unreasonably well: not only can Rust force you to understand why your program is thread-safe, it can even require some amount of high-level architectural design.

Chapter 5. Expressions

A NOTE FOR EARLY RELEASE READERS

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

This will be the 6th chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at jimb@red-bean.com.

LISP programmers know the value of everything, but the cost of nothing.

—Alan Perlis, epigram #55

In this chapter, we’ll cover the *expressions* of Rust, the building blocks that make up the body of Rust functions. 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 that look something like this:

5 * (fahr - 32) / 9

and *statements*, which look more like this:

```
for ( ; begin != end; ++begin) {
    if (*begin == target)
        break;
}
```

Expressions have values. Statements don't.

Rust is what is called an *expression language*. This means it follows an older tradition, dating back to Lisp, where expressions do all the work.

In C, `if` and `switch` are statements. They don't produce a value, and they can't be used in the middle of an expression. In Rust, `if` and `match` *can* produce values. We already saw a `match` expression that produces a numeric value in XREF HERE:

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

An `if` expression can be used to initialize a variable:

```
let status =
    if cpu.temperature <= MAX_TEMP {
        HttpStatus::Ok
    } else {
        HttpStatus::ServerError // server melted
    };
```

A `match` expression can be passed as an argument to a function or macro:

```
println!("Inside the vat, you see {}.",
    match vat.contents {
        Some(brain) => brain.desc(),
```

```
    None => "nothing of interest"
});
```

This explains why Rust does not have C's ternary operator (`expr1 ? expr2 : expr3`). In C, it is a handy expression-level analogue to the `if` statement. It would be redundant in Rust: the `if` expression handles both cases.

Most of the control flow tools in C are statements. In Rust, they are all expressions.

Blocks and Semicolons

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 = post.get_network_metadata()?;
        let ip = network_info.client_address();
        ip.to_string()
    }
};
```

The code after `Some(author) =>` is the simple expression `author.name()`. The code after `None =>` is a block expression. It makes no difference to Rust. The value of the block is the value of its last expression, `ip.to_string()`.

Note that there is no semicolon after that expression. Most lines of Rust code do end with either a semicolon or curly braces, just like C or Java. And if a block looks like C code, with semicolons in all the familiar places, then it will run just like a C block, and its value will be `()`. As we mentioned in XREF HERE, when you leave the semicolon off the last line of a block, you're making that block produce a value—the value of the final expression.

In some languages, particularly JavaScript, you’re allowed to omit semicolons, and the language simply fills them in for you—a minor convenience. This is different. In Rust, the semicolon actually means something.

```
let msg = {
    // let-declaration: semicolon is always required
    let dandelion_control = puffball.open();

    // expression + semicolon: method is called, return value dropped
    dandelion_control.release_all_seeds(launch_codes);

    // expression with no semicolon: method is called,
    // return value stored in `msg`
    dandelion_control.get_status()
};
```

This ability of blocks to contain declarations and also produce a value at the end is a neat feature, one that quickly comes to feel natural. The one drawback is that it leads to an odd error message when you leave out a semicolon by accident.

```
...
if preferences.changed() {
    page.compute_size() // oops, missing semicolon
}
...
```

If you made this mistake in a C or Java program, the compiler would simply point out that you’re missing a semicolon. Here’s what Rust says:

```
error[E0308]: mismatched types
22 |         page.compute_size() // oops, missing semicolon
|         ^^^^^^^^^^^^^^^^^^- help: try adding a semicolon: `;`  
|         |
|         expected (), found tuple
|
= note: expected type `()`  
       found type `(u32, u32)`
```

With the semicolon missing, the block's value would be whatever `page.compute_size()` returns, but an `if` without an `else` must always return `()`. Fortunately, Rust has seen this sort of thing before, and suggests adding the semicolon.

Declarations

In addition to expressions and semicolons, a block may contain any number of declarations. The most common are `let` declarations, which declare local variables:

```
let name: type = expr;
```

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

A `let` declaration can declare a variable without initializing it. The variable can then be initialized with a later assignment. This is occasionally useful, because sometimes a variable should be initialized from the middle of some sort of control flow construct:

```
let name;
if user.hasNickname() {
    name = user.nickname();
} else {
    name = generateUniqueName();
    user.register(&name);
}
```

Here there are two different ways the local variable `name` might be initialized, but either way it will be initialized exactly once, so `name` does not need to be declared `mut`.

It's an error to use a variable before it's initialized. (This is closely related to the error of using a value after it's been moved. Rust really wants you to use values only while they exist!)

You may occasionally see code that seems to redeclare an existing variable, like this:

```
for line in file.lines() {
    let line = line?;
    ...
}
```

This is equivalent to:

```
for line_result in file.lines() {
    let line = line_result?;
    ...
}
```

The `let` declaration creates a new, second variable, of a different type. The type of `line_result` is `Result<String, io::Error>`. The second variable, `line`, is a `String`. It's legal to give the second variable the same name as the first. In this book, we'll stick to using a `_result` suffix in such situations, so that all variables have distinct names.

A block can also contain *item declarations*. An item is simply any declaration that could appear globally in a program or module, such as a `fn`, `struct`, or `use`.

Later chapters will cover items in detail. For now, `fn` makes a sufficient example. Any block may contain a `fn`:

```
use std::io;
use std::cmp::Ordering;

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

    fn cmp_by_timestamp_then_name(a: &FileInfo, b: &FileInfo) -> Ordering {
        a.timestamp.cmp(&b.timestamp) // first, compare timestamps
            .reverse()                // newest file first
    }
}
```

```
        .then(a.path.cmp(&b.path)) // compare paths to break ties
    }

    v.sort_by(cmp_by_timestamp_then_name);
    ...
}
```

When a `fn` is declared inside a block, its scope is the entire block—that is, it can be *used* throughout the enclosing block. But a nested `fn` cannot access local variables or arguments that happen to be in scope. For example, the function `cmp_by_timestamp_then_name` could not use `v` directly. (Rust also has closures, which do see into enclosing scopes. See XREF HERE.)

A block can even contain a whole module. This may seem a bit much—do we really need to be able to nest *every* piece of the language inside every other piece?—but programmers (and particularly programmers using macros) have a way of finding uses for every scrap of orthogonality the language provides.

if and match

The form of an `if` expression is familiar:

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

Each `condition` must be an expression of type `bool`; true to form, Rust does not implicitly convert numbers or pointers to Boolean values.

Unlike C, parentheses are not required around conditions. In fact, `rustc` will emit a warning if unnecessary parentheses are present. The curly braces, however, are required.

The `else if` blocks, as well as the final `else`, are optional. An `if` expression with no `else` block behaves exactly as though it had an empty `else` block.

`match` expressions are something like the C `switch` statement, but more flexible. A simple example:

```
match code {  
    0 => println!("OK"),  
    1 => println!("Wires Tangled"),  
    2 => println!("User Asleep"),  
    _ => println!("Unrecognized Error {}", code)  
}
```

This is something a `switch` statement could do. Exactly one of the four arms of this `match` expression will execute, depending on the value of `code`. The wildcard pattern `_` matches everything, so it serves as the `default`: case.

The compiler can optimize this kind of `match` using a jump table, just like a `switch` statement in C++. A similar optimization is applied when each arm of a `match` produces a constant value. In that case, the compiler builds an array of those values, and the `match` is compiled into an array access. Apart from a bounds check, there is no branching at all in the compiled code.

The versatility of `match` stems from the variety of supported *patterns* that can be used to the left of `=>` in each arm. Above, each pattern is simply a constant integer. We've also shown `match` expressions that distinguish the two kinds of `Option` value:

```
match params.get("name") {  
    Some(name) => println!("Hello, {}!", name),  
    None => println!("Greetings, stranger.")  
}
```

This is only a hint of what patterns can do. A pattern can match a range of values. It can unpack tuples. It can match against individual fields of

structs. It can chase references, borrow parts of a value, and more. Rust's patterns are a mini-language of their own. We'll dedicate several pages to them in [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: nonexhaustive patterns
```

All blocks of an `if` expression must produce values of the same type:

```
let suggested_pet =  
    if with_wings { Pet::Buzzard } else { Pet::Hyena }; // ok  
  
let favorite_number =  
    if user.is_hobbit() { "eleventy-one" } else { 9 }; // error  
  
let best_sports_team =  
    if is_hockey_season() { "Predators" }; // error
```

(The last example is an error because in July, the result would be `()`.)

Similarly, all arms of a `match` expression must have the same type:

```
let suggested_pet =  
  match favorites.element {  
    Fire => Pet::RedPanda,  
    Air => Pet::Buffalo,  
    Water => Pet::Orca,  
    _ => None // error: incompatible types  
};
```

if let

There is one more `if` form, the `if let` expression:

```
if let pattern = expr {  
  block1  
} else {  
  block2  
}
```

The given `expr` either matches the `pattern`, in which case `block1` runs, or it doesn't, and `block2` runs. Sometimes this is a nice way to get data out of an `Option` or `Result`:

```
if let Some(cookie) = request.session_cookie {  
  return restore_session(cookie);  
}  
  
if let Err(err) = present_cheesy_anti_robot_task() {  
  log_robot_attempt(err);  
  politely_accuse_user_of_being_a_robot();  
} else {  
  session.mark_as_human();  
}
```

It's never strictly *necessary* to use `if let`, because `match` can do everything `if let` can do. An `if let` expression is shorthand for a `match` with just one pattern:

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

Loops

There are four looping expressions:

```
while condition {  
    block  
}  
  
while let pattern = expr {  
    block  
}  
  
loop {  
    block  
}  
  
for pattern in collection {  
    block  
}
```

Loops are expressions in Rust, but the value of a `while` or `for` loop is always `()`, so their value isn't very useful. A `loop` expression can produce a value if you specify one.

A `while` loop behaves exactly like the C equivalent, except that again, the `condition` must be of the exact type `bool`.

The `while let` loop is analogous to `if let`. At the beginning of each loop iteration, the value of `expr` either matches the given `pattern`, in which case the `block` runs, or 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 XREF HERE. 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 rs in &strings {
    println!("String {:?} is at address {:p}.", *rs, rs);
}
```

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

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

```
for rs in &mut strings { // the type of rs is &mut String
    rs.push('\n'); // add a newline to each string
}
```

XREF HERE 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.)

Within the body of a `loop`, you can give `break` an expression, whose value becomes that of the loop:

```
// Each call to `next_line` returns either `Some(line)`, where
// `line` is a line of input, or `None`, if we've reached the end of
// the input. Return the first line that starts with `answer: `
// otherwise, return "answer: nothing".
let answer = loop {
    if let Some(line) = next_line() {
        if line.starts_with("answer: ") {
            break line;
        }
    } else {
        break "answer: nothing";
    }
};
```

Naturally, all the `break` expressions within a `loop` must produce values with the same type, which becomes the type of the `loop` itself.

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

```
// Read some data, one line at a time.
for line in input_lines {
    let trimmed = trim_comments_and_whitespace(line);
    if trimmed.is_empty() {
        // Jump back to the top of the loop and
        // move on to the next line of input.
        continue;
    }
    ...
}
```

In a `for` loop, `continue` advances to the next value in the collection. If there are no more values, the loop exits. Similarly, in a `while` loop, `continue` rechecks the loop condition. If it's now false, the loop exits.

A loop can be *labeled* with a lifetime. In the following example, '`search:`' is a label for the outer `for` loop. Thus `break 'search'` exits that loop, not the inner loop.

```
'search:
for room in apartment {
    for spot in room.hiding_spots() {
        if spot.contains(keys) {
            println!("Your keys are {} in the {}.", spot, room);
            break 'search;
        }
    }
}
```

A `break` can have both a label and a value expression:

```
// Find the square root of the first perfect square
// in the series.
let sqrt = 'outer: loop {
    let n = next_number();
    for i in 1.. {
        let square = i * i;
        if square == n {
            // Found a square root.
            break 'outer i;
    }
}
```

```
    if square > n {  
        // `n` isn't a perfect square, try the next  
        break;  
    }  
};
```

Labels can also be used with `continue`.

return Expressions

A `return` expression exits the current function, returning a value to the caller.

`return` without a value is shorthand for `return ()`:

```
fn f() {    // return type omitted: defaults to ()  
    return; // return value omitted: defaults to ()  
}
```

Like a `break` expression, `return` can abandon work in progress. For example, back in XREF HERE, we used the `?` operator to check for errors after calling a function that can fail:

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

and we explained that this is shorthand for a `match` expression:

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

This code starts by calling `File::create(filename)`. If that returns `Ok(f)`, then the whole `match` expression evaluates to `f`, so `f` is stored in `output` and we continue with the next line of code following the `match`.

Otherwise, we'll match `Err(err)` and hit the `return` expression. When that happens, it doesn't matter that we're in the middle of evaluating a `match` expression to determine the value of the variable `output`. We abandon all of that and exit the enclosing function, returning whatever error we got from `File::create()`.

We'll cover the `?` operator more completely in “[Propagating Errors](#)”.

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: mismatched types: expected i32, found ()
```

The error here is bogus. This function only exits via the `return` statement, so the fact that the `while` loop doesn't produce an `i32` is irrelevant.

The `loop` expression is offered as a “say-what-you-mean” solution to this problem.

Rust's type system is affected by control flow, too. Earlier we said that all branches of an `if` expression must have the same type. But it would be silly to enforce this rule on blocks that end with a `break` or `return` expression, an infinite `loop`, or a call to `panic!()` or `std::process::exit()`. What all those expressions have in common is that they never finish in the usual way, producing a value. A `break` or `return` exits the current block abruptly; 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()`:

```
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.

This concludes the part of this chapter that focuses on control flow. The rest covers Rust functions, methods, and operators.

Function and Method Calls

The syntax for calling functions and methods is the same in Rust as in many other languages:

```
let x = gcd(1302, 462); // function call  
let room = player.location(); // method call
```

In the second example here, `player` is a variable of the made-up type `Player`, which has a made-up `.location()` method. (We'll show how to define your own methods when we start talking about user-defined types in [Chapter 8](#).)

Rust usually makes a sharp distinction between references and the values they refer to. If you pass a `&i32` to a function that expects an `i32`, that's a type error. You'll notice that the `.` operator relaxes those rules a bit. In the method call `player.location()`, `player` might be a `Player`, a reference of type `&Player`, or a smart pointer of type `Box<Player>` or `Rc<Player>`. The `.location()` method might take the player either by value or by reference. The same `.location()` syntax works in all cases, because Rust's `.` operator automatically dereferences `player` or borrows a reference to it as needed.

A third syntax is used for calling static methods, like `Vec::new()`.

```
let mut numbers = Vec::new(); // static method call
```

The difference between static and nonstatic methods is the same as in object-oriented languages: nonstatic methods are called on values (like `my_vec.len()`), and static methods are called on types (like `Vec::new()`).

Naturally, method calls can be chained:

```
// From the Actix-based web server in chapter 2:  
server  
    .bind("127.0.0.1:3000").expect("error binding server to address")  
    .run().expect("error running server");
```

One quirk of Rust syntax is that in a function call or method call, the usual syntax for generic types, `Vec<T>`, does not work:

```
return Vec<i32>::with_capacity(1000); // error: something about chained  
comparisons  
  
let ramp = (0 .. n).collect<Vec<i32>>(); // same error
```

The problem is that in expressions, `<` is the less-than operator. The Rust compiler helpfully suggests writing `::<T>` instead of `<T>` in this case, and that solves the problem:

```
return Vec::<i32>::with_capacity(1000); // ok, using ::<  
  
let ramp = (0 .. n).collect::<Vec<i32>>(); // ok, using ::<
```

The symbol `::<...>` is affectionately known in the Rust community as the *turbofish*.

Alternatively, it is often possible to drop the type parameters and let Rust infer them:

```
return Vec::with_capacity(10); // ok, if the fn return type is Vec<i32>
let ramp: Vec<i32> = (0 .. n).collect(); // ok, variable's type is given
```

It's considered good style to omit the types whenever they can be inferred.

Fields and Elements

The fields of a struct are accessed using familiar syntax. Tuples are the same except that their fields have numbers rather than names:

```
game.black_pawns // struct field
coords.1          // tuple element
```

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 these three are called *lvalues*, because they can appear on the left side of an assignment:

```
game.black_pawns = 0x00ff0000_00000000_u64;
coords.1 = 0;
pieces[2] = Some(Piece::new(Black, Knight, coords));
```

Of course, this is permitted only if `game`, `coords`, and `pieces` are declared as `mut` variables.

Extracting a slice from an array or vector is straightforward:

```
let second_half = &game_moves[midpoint .. end];
```

Here `game_moves` may be either an array, a slice, or a vector; the result, regardless, is a borrowed slice of length `end - midpoint`. `game_moves` is considered borrowed for the lifetime of `second_half`.

The `..` operator allows either operand to be omitted; it produces up to four different types of object depending on which operands are present:

```
..      // RangeFull
a ..    // RangeFrom { start: a }
.. b    // RangeTo { end: b }
a .. b // Range { start: a, end: b }
```

These ranges are *half-open*: they include the start value, if any, but not the end value. The range `0 .. 3` includes the numbers `0, 1, and 2`.

The `..=` operator produces *closed* ranges, that do include the end value:

```
..= b   // RangeToInclusive { end: b }
a ..= b // RangeInclusive::new(a, b)
```

For example, the range `0 ..= 3` includes the numbers `0, 1, 2, and 3`.

Only ranges that include a start value are iterable, since a loop must have somewhere to start. But in array slicing, all six forms are useful. If the start or end of the range is omitted, it defaults to the start or end of the data being sliced.

So an implementation of quicksort, the classic divide-and-conquer sorting algorithm, might look, in part, like this:

```
fn quicksort<T: Ord>(slice: &mut [T]) {
    if slice.len() <= 1 {
        return; // Nothing to sort.
    }
```

```

    // Partition the slice into two parts, front and back.
    let pivot_index = partition(slice);

    // Recursively sort the front half of `slice`.
    quicksort(&mut slice[.. pivot_index]);

    // And the back half.
    quicksort(&mut slice[pivot_index + 1 ..]);
}

```

Reference Operators

The address-of operators, `&` and `&mut`, are covered in [Chapter 4](#).

The unary `*` operator is used to access the value pointed to by a reference. As we've seen, Rust automatically follows references when you use the `.` operator to access a field or method, so the `*` operator is necessary only when we want to read or write the entire value that the reference points to.

For example, sometimes an iterator produces references, but the program needs the underlying values:

```

let padovan: Vec<u64> = compute_padovan_sequence(n);
for elem in &padovan {
    draw_triangle(turtle, *elem);
}

```

In this example, the type of `elem` is `&u64`, so `*elem` is a `u64`.

Arithmetic, Bitwise, Comparison, and Logical Operators

Rust's binary operators are like those in many other languages. To save time, we assume familiarity with one of those languages, and focus on the few points where Rust departs from tradition.

Rust has the usual arithmetic operators, `+`, `-`, `*`, `/`, and `%`. As mentioned in [Chapter 2](#), integer overflow is detected, and causes a panic, in debug builds.

The standard library provides methods like `a.wrapping_add(b)` for unchecked arithmetic.

Dividing an integer by zero triggers a panic even in release builds. Integers have a method `a.checked_div(b)` that returns an `Option` (`None` if `b` is zero) and never panics.

Unary - negates a number. It is supported for all the numeric types except unsigned integers. There is no unary + operator.

```
println!("{}", -100);      // -100
println!("{}", -100u32);   // error: can't apply unary '-' to type `u32`
println!("{}", +100);      // error: expected expression, found `+'
```

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 unsigned integers, it does not need Java's >>> operator.

Bitwise operations have higher precedence than comparisons, unlike C, so if you write `x & BIT != 0`, that means `(x & BIT) != 0`, as you probably

intended. This is much more useful than C's interpretation, `x & (BIT != 0)`, which tests the wrong bit!

Rust's comparison operators are `==`, `!=`, `<`, `<=`, `>`, and `>=`. The two values being compared must have the same type.

Rust also has the two short-circuiting logical operators `&&` and `||`. Both operands must have the exact type `bool`.

Assignment

The `=` operator can be used to assign to `mut` variables and their fields or elements. But assignment is not as common in Rust as in other languages, since variables are immutable by default.

As described in [Chapter 3](#), if the value has a non-Copy type, assignment *moves* it into the destination. Ownership of the value is transferred from the source to the destination. The destination's prior value, if any, is dropped.

Compound assignment is supported:

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

This is equivalent to `total = total + item.price;`. Other operators are supported too: `-=`, `*=`, and so forth. The full list is given in [Table 5-1](#), at the end of this chapter.

Unlike C, Rust doesn't support chaining assignment: you can't write `a = b = 3` to assign the value 3 to both `a` and `b`. Assignment is rare enough in Rust that you won't miss this shorthand.

Rust does not have C's increment and decrement operators `++` and `--`.

Type Casts

Converting a value from one type to another usually requires an explicit cast in Rust. Casts use the `as` keyword:

```
let x = 17;           // x is type i32
let index = x as usize; // convert to usize
```

Several kinds of casts are permitted:

- Numbers may be cast from any of the built-in numeric types to any other.

Casting an integer to another integer type is always well-defined. Converting to a narrower type results in truncation. A signed integer cast to a wider type is sign-extended; an unsigned integer is zero-extended; and so on. In short, there are no surprises.

Converting from a floating-point type to an integer type rounds towards zero: the value of `-1.99 as i32` is `-1`. If the value is too large to fit in the integer type, the cast produces the closest value that the integer type can represent: the value of `1e6 as u8` is `255`.

- Values of type `bool`, `char`, or of a C-like `enum` type, may be cast to any integer type. (We'll cover enums in [Chapter 9](#).)

Casting in the other direction is not allowed, as `bool`, `char`, and `enum` types all have restrictions on their values that would have to be enforced with run-time checks. For example, casting a `u16` to type `char` is banned because some `u16` values, like `0xd800`, correspond to Unicode surrogate code points and therefore would not make valid `char` values. There is a standard method, `std::char::from_u32()`, which performs the run-time check and returns an `Option<char>`; but more to the point, the need for this kind of conversion has grown rare. We typically convert whole strings or streams at once, and algorithms on Unicode text are often nontrivial and best left to libraries.

As an exception, a `u8` may be cast to type `char`, since all integers from 0 to 255 are valid Unicode code points for `char` to hold.

- Some casts involving unsafe pointer types are also allowed. See [XREF HERE](#).

We said that a conversion *usually* requires a cast. A few conversions involving reference types are so straightforward that the language performs them even without a cast. One trivial example is converting a `mut` reference to a non-`mut` reference.

Several more significant automatic conversions can happen, though:

- Values of type `&String` auto-convert to type `&str` without a cast.
- Values of type `&Vec<i32>` auto-convert to `&[i32]`.
- Values of type `&Box<Chessboard>` auto-convert to `&Chessboard`.

These are called *deref coercions*, because they apply to types that implement the `Deref` built-in trait. The purpose of Deref coercion is to make smart pointer types, like `Box`, behave as much like the underlying value as possible. Using a `Box<Chessboard>` is mostly just like using a plain `Chessboard`, thanks to `Deref`.

User-defined types can implement the `Deref` trait, too. When you need to write your own smart pointer type, see XREF HERE.

Closures

Rust has *closures*, lightweight function-like values. A closure usually consists of an argument list, given between vertical bars, followed by an expression:

```
let is_even = |x| x % 2 == 0;
```

Rust infers the argument types and return type. You can also write them out explicitly, as you would for a function. If you do specify a return type, then the body of the closure must be a block, for the sake of syntactic sanity:

```
let is_even = |x: u64| -> bool x % 2 == 0; // error
```

```
let is_even = |x: u64| -> bool { x % 2 == 0 }; // ok
```

Calling a closure uses the same syntax as calling a function:

```
assert_eq!(is_even(14), true);
```

Closures are one of Rust's most delightful features, and there is a great deal more to be said about them. We shall say it in XREF HERE.

Precedence and Associativity

Table 5-1 gives a summary of Rust expression syntax. Operators are listed in order of precedence, from highest to lowest. (Like most programming languages, Rust has *operator precedence* to determine the order of operations when an expression contains multiple adjacent operators. For example, in `limit < 2 * broom.size + 1`, the `.` operator has the highest precedence, so the field access happens first.)

Table 5-1. Expressions

Expression type	Example	Related traits
Array literal	[1, 2, 3]	
Repeat array literal	[0; 50]	
Tuple	(6, "crullers")	
Grouping	(2 + 2)	
Block	{ f(); g() }	
Control flow expressions	<pre>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</pre>	
Macro invocation	println!("ok")	
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
Function call	stdin()	Fn(Arg0, ...) -> T, FnMut(Arg0, ...) -> T, FnOnce(Arg0, ...) -> T

Expression type	Example	Related traits
Index	<code>arr[0]</code>	Index, IndexMut <code>Deref, DerefMut</code>
Error check	<code>create_dir("tmp")?</code>	
Logical/bitwise NOT	<code>!ok</code>	Not
Negation	<code>-num</code>	Neg
Dereference	<code>*ptr</code>	Deref, DerefMut
Borrow	<code>&val</code>	
Type cast	<code>x as u32</code>	
Multiplication	<code>n * 2</code>	Mul
Division	<code>n / 2</code>	Div
Remainder (modulus)	<code>n % 2</code>	Rem
Addition	<code>n + 1</code>	Add
Subtraction	<code>n - 1</code>	Sub
Left shift	<code>n << 1</code>	Shl
Right shift	<code>n >> 1</code>	Shr
Bitwise AND	<code>n & 1</code>	BitAnd
Bitwise exclusive OR	<code>n ^ 1</code>	BitXor
Bitwise OR	<code>n 1</code>	BitOr
Less than	<code>n < 1</code>	<code>std::cmp::PartialOrd</code>
Less than or equal	<code>n <= 1</code>	<code>std::cmp::PartialOrd</code>
Greater than	<code>n > 1</code>	<code>std::cmp::PartialOrd</code>
Greater than or equal	<code>n >= 1</code>	<code>std::cmp::PartialOrd</code>
Equal	<code>n == 1</code>	<code>std::cmp::PartialEq</code>
Not equal	<code>n != 1</code>	<code>std::cmp::PartialEq</code>
Logical AND	<code>x.ok && y.ok</code>	
Logical OR	<code>x.ok backup.ok</code>	
Range	<code>start .. stop</code>	

Expression type	Example	Related traits
Assignment	<code>x = val</code>	
Compound assignment	<code>x *= 1</code>	<code>MulAssign</code>
	<code>x /= 1</code>	<code>DivAssign</code>
	<code>x %= 1</code>	<code>RemAssign</code>
	<code>x += 1</code>	<code>AddAssign</code>
	<code>x -= 1</code>	<code>SubAssign</code>
	<code>x <= 1</code>	<code>ShlAssign</code>
	<code>x >= 1</code>	<code>ShrAssign</code>
	<code>x &= 1</code>	<code>BitAndAssign</code>
	<code>x ^= 1</code>	<code>BitXorAssign</code>
	<code>x = 1</code>	<code>BitOrAssign</code>
Closure	<code> x, y x + y</code>	

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

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

The comparison operators, the assignment operators, and the range 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.

Coming up, we'll spend several chapters talking about features in this area: modules and crates, which give your program structure, and then structs and enums, which do the same for your data.

First, we'll dedicate a few pages to the important topic of what to do when things go wrong.

Chapter 6. Error Handling

A NOTE FOR EARLY RELEASE READERS

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

This will be the 7th chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at jimb@red-bean.com.

I knew if I stayed around long enough, something like this would happen.

—George Bernard Shaw on dying

Error handling in Rust is just different enough to warrant its own short chapter. There aren’t any difficult ideas here, just ideas that might be new to you. This chapter covers the two different kinds of error-handling in Rust: `panic` and `Results`.

Ordinary errors are handled using `Results`. These are typically caused by things outside the program, like erroneous input, a network outage, or a permissions problem. That such situations occur is not up to us; even a bug-free program will encounter them from time to time. Most of this chapter is dedicated to that kind of error. We’ll cover `panic` first, though, because it’s the simpler of the two.

`Panic` is for the other kind of error, the kind that *should never happen*.

Panic

A program panics when it encounters something so messed up that there must be a bug in the program itself. Something like:

- Out-of-bounds array access
- Integer division by zero
- Calling `.expect()` on a `Result` that happens to be `Err`
- Assertion failure

(There's also the macro `panic!()`, for cases where your own code discovers that it has gone wrong, and you therefore need to trigger a panic directly. `panic!()` accepts optional `println!()`-style arguments, for building an error message.)

What these conditions have in common is that they are all—not to put too fine a point on it—the programmer's fault. A good rule of thumb is: “Don't panic”.

But we all make mistakes. When these errors that shouldn't happen, do happen—what then? Remarkably, Rust gives you a choice. Rust can either unwind the stack when a panic happens, or abort the process. Unwinding is the default.

Unwinding

When pirates divvy up the booty from a raid, the captain gets half of the loot. Ordinary crew members earn equal shares of the other half. (Pirates hate fractions, so if either division does not come out even, the result is rounded down, with the remainder going to the ship's parrot.)

```
fn pirate_share(total: u64, crew_size: usize) -> u64 {
    let half = total / 2;
    half / crew_size as u64
}
```

This may work fine for centuries until one day it transpires that the captain is the sole survivor of a raid. If we pass a `crew_size` of zero to this function, it will divide by zero. In C++, this would be undefined behavior. In Rust, it triggers a panic, which typically proceeds as follows:

- An error message is printed to the terminal:

```
thread 'main' panicked at 'attempt to divide by zero',
pirates.rs:3780
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

If you set the `RUST_BACKTRACE` environment variable, as the messages suggests, Rust will also dump the stack at this point.

- The stack is unwound. This is a lot like C++ exception handling. Any temporary values, local variables, or arguments that the current function was using are dropped, in the reverse of the order they were created. Dropping a value simply means cleaning up after it: any `Strings` or `Vecs` the program was using are freed, any open `Files` are closed, and so on. User-defined `drop` methods are called too; see XREF HERE. In the particular case of `pirate_share()`, there's nothing to clean up.
Once the current function call is cleaned up, we move on to its caller, dropping its variables and arguments the same way. Then *that* function's caller, and so on up the stack.
- Finally, the thread exits. If the panicking thread was the main thread, then the whole process exits (with a nonzero exit code).

Perhaps *panic* is a misleading name for this orderly process. A panic is not a crash. It's not undefined behavior. It's more like a `RuntimeException` in Java or a `std::logic_error` in C++. The behavior is well-defined; it just shouldn't be happening.

Panic is safe. It doesn't violate any of Rust's safety rules; even if you manage to panic in the middle of a standard library method, it will never leave a dangling pointer or a half-initialized value in memory. The idea is that Rust catches the invalid array access, or whatever it is, *before* anything bad happens. It would be unsafe to proceed, so Rust unwinds the stack. But the rest of the process can continue running.

Panic is per thread. One thread can be panicking while other threads are going on about their normal business. In XREF HERE, we'll show how a parent thread can find out when a child thread panics and handle the error gracefully.

There is also a way to *catch* stack unwinding, allowing the thread to survive and continue running. The standard library function

`std::panic::catch_unwind()` does this. We won't cover how to use it, but this is the mechanism used by Rust's test harness to recover when an assertion fails in a test. (It can also be necessary when writing Rust code that can be called from C or C++, because unwinding across non-Rust code is undefined behavior; see XREF HERE.)

Ideally, we would all have bug-free code that never panics. But nobody's perfect. You can use threads and `catch_unwind()` to handle panic, making your program more robust. One important caveat is that these tools only catch panics that unwind the stack. Not every panic proceeds this way.

Aborting

Stack unwinding is the default panic behavior, but there are two circumstances in which Rust does not try to unwind the stack.

If a `.drop()` method triggers a second panic while Rust is still trying to clean up after the first, this is considered fatal. Rust stops unwinding and aborts the whole process.

Also, Rust's panic behavior is customizable. If you compile with `-C panic=abort`, the *first* panic in your program immediately aborts the

process. (With this option, Rust does not need to know how to unwind the stack, so this can reduce the size of your compiled code.)

This concludes our discussion of panic in Rust. There is not much to say, because ordinary Rust code has no obligation to handle panic. Even if you do use threads or `catch_unwind()`, all your panic-handling code will likely be concentrated in a few places. It's unreasonable to expect every function in a program to anticipate and cope with bugs in its own code. Errors caused by other factors are another kettle of fish.

Result

Rust doesn't have exceptions. Instead, functions that can fail have a return type that says so:

```
fn get_weather(location: LatLng) -> Result<WeatherReport, io::Error>
```

The `Result` type indicates possible failure. When we call the `get_weather()` function, it will return either a *success result* `Ok(weather)`, where `weather` is a new `WeatherReport` value, or an *error result* `Err(error_value)`, where `error_value` is an `io::Error` explaining what went wrong.

Rust requires us to write some kind of error handling whenever we call this function. We can't get at the `WeatherReport` without doing *something* to the `Result`, and you'll get a compiler warning if a `Result` value isn't used.

In [Chapter 9](#), we'll see how the standard library defines `Result` and how you can define your own similar types. For now, we'll take a "cookbook" approach and focus on how to use `Results` to get the error-handling behavior you want.

Catching Errors

The most thorough way of dealing with a `Result` is the way we showed in XREF HERE: use a `match` expression.

```
match get_weather(hometown) {
    Ok(report) => {
        display_weather(hometown, &report);
    }
    Err(err) => {
        println!("error querying the weather: {}", err);
        schedule_weather_retry();
    }
}
```

This is Rust's equivalent of `try/catch` in other languages. It's what you use when you want to handle errors head-on, not pass them on to your caller.

`match` is a bit verbose, so `Result<T, E>` offers a variety of methods that are useful in particular common cases. Each of these methods has a `match` expression in its implementation. (For the full list of `Result` methods, consult the online documentation. The methods listed here are the ones we use the most.)

- `result.is_ok()` and `result.is_err()` return a `bool` telling if `result` is a success result or an error result.
- `result.ok()` returns the success value, if any, as an `Option<T>`. If `result` is a success result, this returns `Some(success_value)`; otherwise, it returns `None`, discarding the error value.
- `result.err()` returns the error value, if any, as an `Option<E>`.
- `result.unwrap_or(fallback)` returns the success value, if `result` is a success result. Otherwise, it returns `fallback`, discarding the error value.

// A fairly safe prediction for Southern California.

```

const THE_USUAL: WeatherReport = WeatherReport::Sunny(72);

// Get a real weather report, if possible.
// If not, fall back on the usual.
let report = get_weather(los_angeles).unwrap_or(THE_USUAL);
display_weather(los_angeles, &report);

```

This is a nice alternative to `.ok()` because the return type is `T`, not `Option<T>`. Of course, it only works when there's an appropriate fallback value.

- `result.unwrap_or_else(fallback_fn)` is the same, but instead of passing a fallback value directly, you pass a function or closure. This is for cases where it would be wasteful to compute a fallback value if you're not going to use it. The `fallback_fn` is called only if we have an error result.

```

let report =
    get_weather(hometown)
    .unwrap_or_else(|err| vague_prediction(hometown));

```

(XREF HERE covers closures in detail.)

- `result.unwrap()` also returns the success value, if `result` is a success result. However, if `result` is an error result, this method panics. This method has its uses; we'll talk more about it later.
- `result.expect(message)` is the same as `.unwrap()`, but lets you provide a message that it prints in case of panic.

Lastly, two methods for borrowing references to the value in a `Result`:

- `result.as_ref()` converts a `Result<T, E>` to a `Result<&T, &E>`, borrowing a reference to the success or error value in the existing `result`.

- `result.as_mut()` is the same, but borrows a mutable reference. The return type is `Result<&mut T, &mut E>`.

One reason these last two methods are useful is that all of the other methods listed here, except `.is_ok()` and `.is_err()`, *consume* the `result` they operate on. That is, they take the `self` argument by value. Sometimes it's quite handy to access data inside a result without destroying it, and this is what `.as_ref()` and `.as_mut()` do for us. For example, suppose you'd like to call `result.ok()`, but you need `result` to be left intact. You can write `result.as_ref().ok()`, which merely borrows `result`, returning an `Option<&T>` rather than an `Option<T>`.

Result Type Aliases

Sometimes you'll see Rust documentation that seems to omit the error type of a `Result`:

```
fn remove_file(path: &Path) -> Result<()>
```

This means that a `Result` type alias is being used.

A type alias is a kind of shorthand for type names. Modules often define a `Result` type alias to avoid having to repeat an error type that's used consistently by almost every function in the module. For example, the standard library's `std::io` module includes this line of code:

```
pub type Result<T> = result::Result<T, Error>;
```

This defines a public type `std::io::Result<T>`. It's an alias for `Result<T, E>`, but hardcoding `std::io::Error` as the error type. In practical terms, this means that if you write `use std::io;` then Rust will understand `io::Result<String>` as shorthand for `Result<String, io::Error>`.

When something like `Result<()>` appears in the online documentation, you can click on the identifier `Result` to see which type alias is being used and learn the error type. In practice, it's usually obvious from context.

Printing Errors

Sometimes the only way to handle an error is by dumping it to the terminal and moving on. We already showed one way to do this:

```
println!("error querying the weather: {}", err);
```

The standard library defines several error types with boring names:

`std::io::Error`, `std::fmt::Error`, `std::str::Utf8Error`, and so on. All of them implement a common interface, the `std::error::Error` trait, which means they share the following features:

- They're all printable using `println!()`. Printing an error with the `{}` format specifier typically displays only a brief error message. Alternatively, you can print with the `{:?}` format specifier, to get a `Debug` view of the error. This is less user-friendly, but includes extra technical information.

```
// result of `println!("error: {}", err);`  
error: failed to lookup address information: No address associated  
with  
hostname  
  
// result of `println!("error: {:?}", err);`  
error: Error { repr: Custom(Custom { kind: Other, error: StringError(  
"failed to lookup address information: No address associated with  
hostname") }) }
```

- `err.to_string()` returns an error message as a `String`.

- `err.source()` returns an `Option` of the underlying error, if any, that caused `err`.

For example, a networking error might cause a banking transaction to fail, which could in turn cause your boat to be repossessed. If `err.to_string()` is "boat was repossessed", then `err.source()` might return an error about the failed transaction. That error's `.to_string()` might be "failed to transfer \$300 to United Yacht Supply", and its `.source()` might be an `io::Error` with details about the specific network outage that caused all the fuss. This third error is the root cause, so its `.source()` method would return `None`.

Since the standard library only includes rather low-level features, this is usually `None` for standard library errors.

Printing an error value does not also print out its source. If you want to be sure to print all the available information, use this function:

```
use std::error::Error;
use std::io::{Write, stderr};

/// Dump an error message to `stderr`.
///
/// If another error happens while building the error message or
/// writing to `stderr`, it is ignored.
fn print_error(mut err: &dyn Error) {
    let _ = writeln!(stderr(), "error: {}", err);
    while let Some(source) = err.source() {
        let _ = writeln!(stderr(), "caused by: {}", source);
        err = source;
    }
}
```

The `writeln!` macro works like `println!`, except that it writes the data to a stream of your choice. Here, we write the error messages to the standard error stream, `std::io::stderr`. We could use the `eprintln!` macro to do the same thing, but `eprintln!` panics if an error occurs. In `print_error`,

we want to ignore errors that arise while writing the message; we explain why in “[Ignoring Errors](#)”, later in the chapter.

The standard library’s error types do not include a stack trace, but the `error-chain` crate makes it easy to define your own custom error type that supports grabbing a stack trace when it’s created. It uses the `backtrace` crate to capture the stack.

Propagating Errors

In most places where we try something that could fail, we don’t want to catch and handle the error immediately. It is simply too much code to use a 10-line `match` statement every place where something could go wrong.

Instead, if an error occurs, we usually want to let our caller deal with it. We want errors to *propagate* up the call stack.

Rust has a `?` operator that does this. You can add a `?` to any expression that produces a `Result`, such as the result of a function call:

```
let weather = get_weather(hometown)?;
```

The behavior of `?` depends on whether this function returns a success result or an error result:

- On success, it unwraps the `Result` to get the success value inside. The type of `weather` here is not `Result<WeatherReport, io::Error>` but simply `WeatherReport`.
- On error, it immediately returns from the enclosing function, passing the error result up the call chain. To ensure that this works, `?` can only be used in functions that have a `Result` return type.

There’s nothing magical about the `?` operator. You can express the same thing using a `match` expression, although it’s much wordier:

```
let weather = match get_weather(hometown) {
```

```
Ok(success_value) => success_value,  
Err(err) => return Err(err)  
};
```

The only differences between this and the `?` operator are some fine points involving types and conversions. We'll cover those details in the next section.

In older code, you may see the `try!()` macro, which was the usual way to propagate errors until the `?` operator was introduced in Rust 1.13.

```
let weather = try!(get_weather(hometown));
```

The macro expands to a `match` expression, like the one above.

It's easy to forget just how pervasive the possibility of errors is in a program, particularly in code that interfaces with the operating system. The `?` operator sometimes shows up on almost every line of a function:

```
use std::fs;  
use std::io;  
use std::path::Path;  
  
fn move_all(src: &Path, dst: &Path) -> io::Result<()> {  
    for entry_result in src.read_dir()? { // opening dir could fail  
        let entry = entry_result?; // reading dir could fail  
        let dst_file = dst.join(entry.file_name());  
        fs::rename(entry.path(), dst_file)?; // renaming could fail  
    }  
    Ok(()) // phew!  
}
```

`?` also works similarly with the `Option` type. In a function that returns `Option`, you can use `?` to unwrap a value and return early in the case of `None`.

```
let weather = get_weather(hometown).ok();
```

Working with Multiple Error Types

Often, more than one thing could go wrong. Suppose we are simply reading numbers from a text file.

```
use std::io::{self, BufRead};

/// Read integers from a text file.
/// The file should have one number on each line.
fn read_numbers(file: &mut dyn BufRead) -> Result<Vec<i64>, io::Error> {
    let mut numbers = vec![];
    for line_result in file.lines() {
        let line = line_result?;           // reading lines can fail
        numbers.push(line.parse()?);      // parsing integers can fail
    }
    Ok(numbers)
}
```

Rust gives us a compiler error:

```
error: `?` couldn't convert the error to `std::io::Error`

numbers.push(line.parse()?);      // parsing integers can fail
                                ^
the trait `std::convert::From<std::num::ParseIntError>`
is not implemented for `std::io::Error`

note: the question mark operation (`?`) implicitly performs a conversion
on the error value using the `From` trait
```

The terms in this error message will make more sense when we reach [Chapter 10](#), which covers traits. For now, just note that Rust is complaining that the ? operator can't convert a `std::num::ParseIntError` value to the type `std::io::Error`.

The problem here is that reading a line from a file and parsing an integer produce two different potential error types. The type of `line_result` is `Result<String, std::io::Error>`. The type of `line.parse()` is `Result<i64, std::num::ParseIntError>`. The return type of our `read_numbers()` function only accommodates `io::Errors`. Rust tries to

cope with the `ParseIntError` by converting it to a `io::Error`, but there's no such conversion, so we get a type error.

There are several ways of dealing with this. For example, the `image` crate that we used in XREF HERE to create image files of the Mandelbrot set defines its own error type, `ImageError`, and implements conversions from `io::Error` and several other error types to `ImageError`. If you'd like to go this route, try the aforementioned `error-chain` crate, which is designed to help you define good error types with just a few lines of code.

A simpler approach is to use what's built into Rust. All of the standard library error types can be converted to the type `Box<dyn std::error::Error + Send + Sync + 'static>`. This is a bit of a mouthful, but `dyn std::error::Error` represents "any error", and `Send + Sync + 'static` makes it safe to pass between threads, which you'll often want. For convenience, you can define type aliases:

```
type GenericError = Box<dyn std::error::Error + Send + Sync + 'static>;
type GenericResult<T> = Result<T, GenericError>;
```

Then, change the return type of `read_numbers()` to `GenericResult<Vec<i64>>`. With this change, the function compiles. The `?` operator automatically converts either type of error into a `GenericError` as needed.

Incidentally, the `?` operator does this automatic conversion using a standard method that you can use yourself. To convert any error to the `GenericError` type, call `GenericError::from()`:

```
let io_error = io::Error::new(          // make our own io::Error
    io::ErrorKind::Other, "timed out");
return Err(GenericError::from(io_error)); // manually convert to GenericError
```

We'll cover the `From` trait and its `from()` method fully in XREF HERE.

The downside of the `GenericError` approach is that the return type no longer communicates precisely what kinds of errors the caller can expect. The caller must be ready for anything.

If you’re calling a function that returns a `GenericResult`, and you want to handle one particular kind of error, but let all others propagate out, use the generic method `error.downcast_ref::<ErrorType>()`. It borrows a reference to the error, *if* it happens to be the particular type of error you’re looking for:

```
loop {
    match compile_project() {
        Ok(_) => return Ok(),
        Err(err) => {
            if let Some(mse) = err.downcast_ref::<MissingSemicolonError>() {
                insert_semicolon_in_source_code(mse.file(), mse.line())?;
                continue; // try again!
            }
            return Err(err);
        }
    }
}
```

Many languages have built-in syntax to do this, but it turns out to be rarely needed. Rust has a method for it instead.

Dealing with Errors That “Can’t Happen”

Sometimes we just *know* that an error can’t happen. For example, suppose we’re writing code to parse a configuration file, and at one point we find that the next thing in the file is a string of digits:

```
if next_char.is_digit(10) {
    let start = current_index;
    current_index = skip_digits(&line, current_index);
    let digits = &line[start..current_index];
    ...
}
```

We want to convert this string of digits to an actual number. There's a standard method that does this:

```
let num = digits.parse::<u64>();
```

Now the problem: the `str.parse::<u64>()` method doesn't return a `u64`. It returns a `Result`. It can fail, because some strings aren't numeric.

```
"bleen".parse::<u64>() // ParseIntError: invalid digit
```

But we happen to know that in this case, `digits` consists entirely of digits. What should we do?

If the code we're writing already returns a `GenericResult`, we can tack on a `?` and forget about it. Otherwise, we face the irritating prospect of having to write error-handling code for an error that can't happen. The best choice then would be to use `.unwrap()`, a `Result` method that panics if the result is an `Error`, but simply returns the the success value of an `Ok`:

```
let num = digits.parse::<u64>().unwrap();
```

This is just like `?` except that if we're wrong about this error, if it *can* happen, then in that case we would panic.

In fact, we are wrong about this particular case. If the input contains a long enough string of digits, the number will be too big to fit in a `u64`.

```
"999999999999999999999999".parse::<u64>() // overflow error
```

Using `.unwrap()` in this particular case would therefore be a bug. Bogus input shouldn't cause a panic.

That said, situations do come up where a `Result` value truly can't be an error. For example, in XREF HERE, you'll see that the `Write` trait defines a

common set of methods (`.write()` and others) for text and binary output. All of those methods return `io::Result`, but if you happen to be writing to a `Vec<u8>`, they can't fail. In such cases, it's acceptable to use `.unwrap()` or `.expect(message)` to dispense with the `Results`.

These methods are also useful when an error would indicate a condition so severe or bizarre that panic is exactly how you want to handle it.

```
fn print_file_age(filename: &Path, last_modified: SystemTime) {
    let age = last_modified.elapsed().expect("system clock drift");
    ...
}
```

Here, the `.elapsed()` method can fail only if the system time is *earlier* than when the file was created. This can happen if the file was created recently, and the system clock was adjusted backward while our program was running. Depending on how this code is used, it's a reasonable judgment call to panic in that case, rather than handle the error or propagate it to the caller.

Ignoring Errors

Occasionally we just want to ignore an error altogether. For example, in our `print_error()` function, we had to handle the unlikely situation where printing the error triggers another error. This could happen, for example, if `stderr` is piped to another process, and that process is killed. The original error we were trying to report is probably more important to propagate, so we just want to ignore the troubles with `stderr`, but the Rust compiler warns about unused `Result` values:

```
writeln!(stderr(), "error: {}", err); // warning: unused result
```

The idiom `let _ = ...` is used to silence this warning:

```
let _ = writeln!(stderr(), "error: {}", err); // ok, ignore result
```

Handling Errors in main()

In most places where a `Result` is produced, letting the error bubble up to the caller is the right behavior. This is why `?` is a single character in Rust. As we've seen, in some programs it's used on many lines of code in a row.

But if you propagate an error long enough, eventually it reaches `main()`, and something has to be done with it. Normally, `main()` can't use `?` because its return type is not `Result`.

```
fn main() {
    calculate_tides()?;
    // error: can't pass the buck any further
}
```

The simplest way to handle errors in `main()` is to use `.expect()`.

```
fn main() {
    calculate_tides().expect("error");
    // the buck stops here
}
```

If `calculate_tides()` returns an error result, the `.expect()` method panics. Panicking in the main thread prints an error message, then exits with a nonzero exit code, which is roughly the desired behavior. We use this all the time for tiny programs. It's a start.

The error message is a little intimidating, though:

```
$ tidecalc --planet mercury
thread 'main' panicked at 'error: "moon not found"', /buildslave/rust-
buildbot/s
lave/nightly-dist-rustc-linux/build/src/libcore/result.rs:837
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

The error message is lost in the noise. Also, `RUST_BACKTRACE=1` is bad advice in this particular case.

However, you can also change the type signature of `main()` to return a `Result` type, so you can use `?`.

```
fn main() -> Result<(), TideCalcError> {
    let tides = calculate_tides()?;
    print_tides(tides);
    Ok(())
}
```

This works for any error type that can be printed with the `{ :? }` formatter, which all standard error types, like `std::io::Error`, can be. This technique is easy to use and gives a somewhat nicer error message, but it's not ideal.

```
$ tidecalc --planet mercury
Error: TideCalcError { error_type: NoMoon, message: "moon not found" }
```

If you have more complex error types, or want to include more details in your message, it pays to print the error message yourself:

```
fn main() {
    if let Err(err) = calculate_tides() {
        print_error(&err);
        std::process::exit(1);
    }
}
```

This code uses an `if let` expression to print the error message only if the call to `calculate_tides()` returns an error result. For details about `if let` expressions, see [Chapter 9](#). The `print_error` function is listed in “[Printing Errors](#)”.

Now the output is nice and tidy:

```
$ tidecalc --planet mercury
error: moon not found
```

Declaring a Custom Error Type

Suppose you are writing a new JSON parser, and you want it to have its own error type. (We haven't covered user-defined types yet; that's coming up in a few chapters. But error types are handy, so we'll include a bit of a sneak preview here.)

Approximately the minimum code you would write is:

```
// json/src/error.rs

#[derive(Debug, Clone)]
pub struct JsonError {
    pub message: String,
    pub line: usize,
    pub column: usize,
}
```

This struct will be called `json::error::JsonError`, and when you want to raise an error of this type, you can write:

```
return Err(JsonError {
    message: "expected ']' at end of array".to_string(),
    line: current_line,
    column: current_column
});
```

This will work fine. However, if you want your error type to work like the standard error types, as your library's users will expect, then you have a bit more work to do:

```
use std;
use std::fmt;

// Errors should be printable.
impl fmt::Display for JsonError {
    fn fmt(&self, f: &mut fmt::Formatter) -> Result<(), fmt::Error> {
        write!(f, "{} ({}:{})", self.message, self.line, self.column)
    }
}
```

```
}
```

```
// Errors should implement the std::error::Error trait,
// but the default definitions for the Error methods are fine.
impl std::error::Error for JsonError { }
```

Again, the meaning of the `impl` keyword, `self`, and all the rest will be explained in the next few chapters.

Why Results?

Now we know enough to understand what Rust is getting at by choosing **Results** over exceptions. Here are the key points of the design:

- Rust requires the programmer to make some sort of decision, and record it in the code, at every point where an error could occur. This is good because otherwise, it's easy to get error handling wrong through neglect.
- The most common decision is to allow errors to propagate, and that's written with a single character, '?'. Thus error plumbing does not clutter up your code the way it does in C and Go. Yet it's still visible: you can look at a chunk of code and see at a glance all places where errors are propagated.
- Since the possibility of errors is part of every function's return type, it's clear which functions can fail and which can't. If you change a function to be fallible, you're changing its return type, so the compiler will make you update that function's downstream users.
- Rust checks that **Result** values are used, so you can't accidentally let an error pass silently (a common mistake in C).
- Since **Result** is a data type like any other, it's easy to store success and error results in the same collection. This makes it easy to model partial success. For example, if you're writing a program that loads millions of records from a text file, and you need a way

to cope with the likely outcome that most will succeed, but some will fail, you can represent that situation in memory using a vector of `Results`.

The cost is that you'll find yourself thinking about and engineering error handling more in Rust than you would in other languages. As in many other areas, Rust's take on error handling is wound just a little tighter than what you're used to. For systems programming, it's worth it.

Chapter 7. Crates and Modules

A NOTE FOR EARLY RELEASE READERS

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

This will be the 8th chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at jimb@red-bean.com.

This is one note in a Rust theme: systems programmers can have nice things.

—Robert O’Callahan, “[Random Thoughts on Rust: Crates.io and IDEs](#)”

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 cover a wide range of topics that come up naturally as your project grows, including how to document and test Rust code, how to silence unwanted compiler warnings, how to use Cargo to manage project dependencies and versioning, how to publish open

source libraries on Rust's public crate repository, crates.io, language editions and how Rust evolves, and more.

Crates

Rust programs are made of *crates*. Each crate is a complete, cohesive unit: all the source code for a single library or executable, plus any associated tests, examples, tools, configuration, and other junk. 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 (see [Figure 7-1](#)).

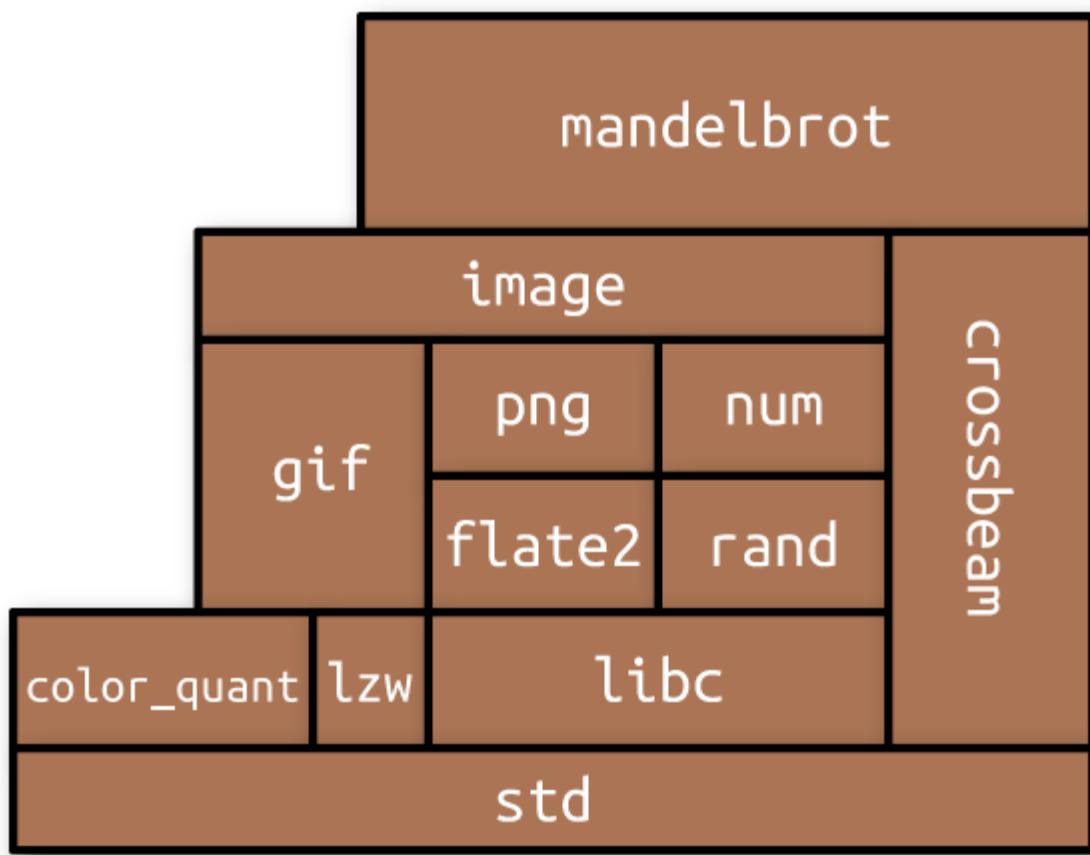


Figure 7-1. A crate and its dependencies

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 XREF HERE as our example. The results are shown here:

```
$ cd mandelbrot
$ cargo clean      # delete previously compiled code
$ cargo build --verbose
    Updating registry `https://github.com/rust-lang/crates.io-index`
    Downloading autocfg v1.0.0
    Downloading semver-parser v0.7.0
    Downloading gif v0.9.0
    Downloading png v0.7.0

... (downloading and compiling many more crates)

Compiling jpeg-decoder v0.1.18
    Running `rustc
        --crate-name jpeg_decoder
        --crate-type lib
        ...
        --extern byteorder=.../libbyteorder-29efdd0b59c6f920.rmeta
        ...
Compiling image v0.13.0
    Running `rustc
        --crate-name image
        --crate-type lib
        ...
        --extern byteorder=.../libbyteorder-29efdd0b59c6f920.rmeta
        --extern gif=.../libgif-a7006d35f1b58927.rmeta
        --extern jpeg_decoder=.../libjpeg_decoder-5c10558d0d57d300.rmeta
Compiling mandelbrot v0.1.0 (/tmp/rustbook-test-files/mandelbrot)
    Running `rustc
        --edition=2018
        --crate-name mandelbrot
        --crate-type bin
        ...
        --extern crossbeam=.../libcrossbeam-f87b4b3d3284acc2.rlib
        --extern image=.../libimage-b5737c12bd641c43.rlib
        --extern num=.../libnum-1974e9a1dc582ba7.rlib -C link-arg=-fuse-
ld=lld`  
    Finished dev [unoptimized + debuginfo] target(s) in 16.94s
$
```

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

with an ellipsis (...).

You might recall that by the time we were done, the Mandelbrot program's `main.rs` contained several `use` declarations for items from other crates:

```
use num::Complex;
// ...
use image::ColorType;
use image::png::PNGEncoder;
```

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

```
[dependencies]
num = "0.1.34"
image = "0.13.0"
crossbeam = "0.2.9"
```

The word *dependencies* here just means other crates this project uses: code we're depending on. We found these crates on [crates.io](#), the Rust community's site for open source crates. For example, we found out about the `image` library by going to [crates.io](#) and searching for an image library. Each crate's page on [crates.io](#) provides links to documentation and source code, as well as a line of configuration like `image = "0.13.0"` that you can copy and add to your `Cargo.toml`. The version numbers shown here are simply the latest versions of these three packages at the time we wrote the program.

The Cargo transcript tells the story of how this information is used. When we run `cargo build`, Cargo starts by downloading 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 0.13.0 of the `image` crate contains a `Cargo.toml` file that includes this:

```
[dependencies]
```

```
byteorder = "1.0.0"
num-iter = "0.1.32"
num-rational = "0.1.32"
num-traits = "0.1.32"
enum_primitive = "0.1.0"
```

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.

Since `mandelbrot` depends on these crates indirectly, through its use of the `image` crate, we call them *transitive* dependencies of `mandelbrot`. The collection of all these dependency relationships, which tells Cargo everything it needs to know about what crates to build and in what order, is known as the *dependency graph* of the crate. Cargo's automatic handling of the dependency graph and transitive dependencies is a huge win in terms of programmer time and effort.

Once it has the source code, Cargo compiles all the crates. It runs `rustc`, the Rust compiler, once for each 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 that can be used to create binaries and other `.rlib` files.

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 `rustc` command, Cargo passes `--extern` options giving the filename of each library the crate will use. That way, when `rustc` sees a line of code like `use image::png::PNGEncoder`, it can figure out that `image` is the name of another crate, and thanks to Cargo, it knows where to find that compiled crate on disk. The Rust compiler needs access to these `.rlib` files because they contain the compiled code of the library. Rust will statically link that code into the final executable. The `.rlib` also contains type information, so Rust can check that the library features we're using in our code actually exist in the crate, and that we're using them correctly. It

also contains a copy of the crate’s public inline functions, generics, and macros, features that can’t be fully compiled to machine code until Rust sees how we use them.

`cargo build` supports all sorts of options, most of which are beyond the scope of this book, but we will mention one here: `cargo build --release` produces an optimized build. Release builds run faster, but they take longer to compile, they don’t check for integer overflow, they skip `debug_assert!()` assertions, and the stack traces they generate on panic are generally less reliable.

Build Profiles

There are several configuration settings you can put in your *Cargo.toml* file that affect the `rustc` command lines that `cargo` generates.

Command line	Cargo.toml section used
<code>cargo build</code>	<code>[profile.dev]</code>
<code>cargo build --release</code>	<code>[profile.release]</code>
<code>cargo test</code>	<code>[profile.test]</code>

The defaults are usually fine, but one exception we’ve found is when you want to use a profiler—a tool that measures where your program is spending its CPU time. To get the best data from a profiler, you need both optimizations (usually enabled only in release builds) and debug symbols (usually enabled only in debug builds). To enable both, add this to your *Cargo.toml*:

```
[profile.release]
debug = true # enable debug symbols in release builds
```

The `debug` setting controls the `-g` option to `rustc`. With this configuration, when you type `cargo build --release`, you’ll get a binary with debug symbols. The optimization settings are unaffected.

The [Cargo documentation](#) lists many other settings you can adjust.

Editions

Rust has extremely strong compatibility guarantees. Any code that compiled on Rust 1.0 must compile just as well on Rust 1.40 or, if it's ever released, Rust 1.900.

But sometimes the community comes across compelling proposals for extensions to the language that would cause older code to no longer compile. For example, after much discussion, Rust settled on a syntax for asynchronous programming support that repurposes the identifiers `async` and `await` as keywords (see XREF HERE). But this language change would break any existing code that uses `async` or `await` as the name of a variable.

To evolve without breaking existing code, Rust uses *editions*. The 2015 edition of Rust is compatible with Rust 1.0. The 2018 edition changed `async` and `await` into keywords, streamlined the module system, and introduced various other language changes that are incompatible with the 2015 edition. Each crate indicates which edition of Rust it is written in with a line like this in the `[package]` section atop its `Cargo.toml` file:

```
edition = "2018"
```

If that keyword is absent, the 2015 edition is assumed, so old crates don't have to change at all. But if you want to use asynchronous functions or the new module system, you'll need `edition = "2018"` in your `Cargo.toml` file (or perhaps something even newer).

Rust promises that the compiler will always accept all extant editions of the language, and programs can freely mix crates written in different editions. It's even fine for a 2015 edition crate to depend on a 2018 edition crate. In other words, a crate's edition only affects how its source code is construed; edition distinctions are gone by the time the code has been compiled. This means there's no pressure to update old crates just to continue to participate

in the modern Rust ecosystem. Similarly, there's no pressure to keep your crate on an older edition to avoid inconveniencing its users. You only need to change editions when you want to use new language features in your own code.

Editions don't come out every year, only when the Rust project decides one is necessary. For example, there's no 2020 edition. Setting `edition` to "2020" causes an error. The [Rust Edition Guide](#) covers the changes introduced in each edition, and provides good background on the edition system.

It's almost always a good idea to use the latest edition, especially for new code. `cargo new` creates new projects on the latest edition by default. This book uses the 2018 edition throughout.

If you have a crate written in an older edition of Rust, the `cargo fix` command may be able to help you automatically upgrade your code to the newer edition. The Rust Edition Guide explains the `cargo fix` command in detail.

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 between projects, modules are about code organization *within* a project. They look like this:

```
mod spores {
    use crate::cells::{Cell, Gene};

    /// 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 the zygote
    /// that grows into 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 {
        ...
    }

    /// Extract the genes in a particular spore.
    pub(crate) fn genes(&self) -> Vec<Gene> {
        ...
    }

    /// 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 in this example. The `pub` keyword makes an item public, so it can be accessed from outside the module.

One function is marked `pub(crate)`, meaning that it is available anywhere inside this crate, but isn't exposed as part of the external interface. It can't be used by other crates, and it won't show up in this crate's documentation.

Anything that isn't marked `pub` is private and can only be used in the same module in which it is defined.

```

let s = spores::produce_spore(&mut factory); // ok

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

```

Marking an item as `pub` is often known as “exporting” that item.

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 {
        ...
    }
}

```

It's also possible to specify `pub(super)`, making an item visible to the parent module only, and `pub(in <path>)`, which makes it visible in a specific parent module. This is especially useful with deeply nested modules.

```

mod plant_structures {
    pub mod roots {
        pub mod products {
            pub mod cytokinin {
                pub(in crate::plant_structures::roots) struct Cytokinin {
                    ...
                }
            }
            use products::cytokinin::Cytokinin; // error: `Cytokinin` is
private
            ...
        }
        use products::cytokinin::Cytokinin; // ok
    }
    ...
}

```

In this way, we could write out a whole program, with a huge amount of code and a whole hierarchy of modules, related in whatever ways we wanted, 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, called `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(factory: &mut Sporangium) -> Spore {
    ...
}

/// Extract the genes in a particular spore.
pub(crate) fn genes(&self) -> Vec<Gene> {
    ...
}

/// 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
    └── spores.rs
        └── plant_structures/
            ├── mod.rs
            ├── leaves.rs
            ├── roots.rs
            └── stems.rs
```

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

```
pub mod plant_structures;
```

This causes 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.

It's also possible to use a file and directory with the same name to make up a module. For instance, if `stems` needed to include modules called `xylem` and `phloem`, we could choose to keep `stems` in `plant_structures/stems.rs` and add a `stems` directory.

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

Then, in `stems.rs`, we declare the two new submodules:

```
// in plant_structures/stems.rs
pub mod xylem;
pub mod phloem;
```

These three options - modules in their own file, modules in their own directory with a `mod.rs`, and modules in their own file with a supplementary directory containing submodules - give the module system enough flexibility to support almost any project structure you might desire.

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 path:

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

`std` is the name of the standard library. The path `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, but it would be tedious to type and hard to read. The alternative is to *import* features into the modules where they're used:

```
use std::mem;

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

The `use` declaration causes the name `mem` to be a local alias for `std::mem` throughout the enclosing block or module.

We could write `use std::mem::swap;` to import the `swap` function itself instead of the `mem` module. However, what we did above is generally considered the best style: import types, traits, and modules (like `std::mem`), then use relative paths to access the functions, constants, and other members within.

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 items in 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 *proteins/mod.rs*:

```
// proteins/mod.rs
pub enum AminoAcid { ... }
pub mod synthesis;
```

Then the code in *synthesis.rs* does not automatically see the type *AminoAcid*:

```
// proteins/synthesis.rs
pub fn synthesize(seq: &[AminoAcid]) // error: can't find type `AminoAcid`
    ...
    ...
```

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

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

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

By default, paths are relative to the current module.

```
// in proteins/mod.rs

// import from a submodule
use synthesis::synthesize;
```

`self` is also a synonym for the current module, so we could write either:

```
// in proteins/mod.rs

// import names from an enum,
```

```
// so we can write `Lys` for lysine, rather than `AminoAcid::Lys`  
use self::AminoAcid::*;


```

or simply:

```
// in proteins/mod.rs  
  
use AminoAcid::*;


```

(The `AminoAcid` example here is, of course, a departure from the style rule we mentioned earlier about only importing types, traits, and modules. If our program includes long amino acid sequences, this is justified under Orwell's Sixth Rule: "Break any of these rules sooner than say anything outright barbarous.")

The keywords `super` and `crate` have a special meaning in paths: `super` refers to the parent module, and `crate` refers to the crate containing the current module.

Using paths relative to the crate root rather than the current module makes it easier to move code around the project, since all the imports won't break if the path of the current module changes. For example, we could write `synthesis.rs` using `crate`:

```
// proteins/synthesis.rs  
use crate::protiens::AminoAcid; // explicitly import relative to crate root  
  
pub fn synthesize(seq: &[AminoAcid]) // ok  
...  
...
```

Submodules can access private items in their parent modules, but they have to import each one by name. `use super::*;` only imports items that are marked `pub`.

Modules aren't the same thing as files, but 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.

If your crate has a module with the same name as another crate that you are using, you'll have to tell the compiler which one you mean. For example, a module named `image` would conflict with the use of the crate `image`.

```
mod image { mod png { ... } }
```

```
use image::png; // Does this mean our png module, or the image crate's png module?
```

To avoid ambiguity, Rust has a special kind of path called *absolute paths*. By default, the `use image::png;` line above refers to the module in the current crate. To refer to the `png` module in the `image` crate, we would have to write

```
mod image { mod png { ... } }
```

```
use ::image::png; // Unambiguously refers to the png module in the image crate.
```

Long crate or module names, especially names that are used often, can be abbreviated using `as`:

```
use crossbeam as c;
```

```
c::thread::spawn( ... )
```

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 linked with every project. This means you can always `use std::whatever` or refer to `std` items by name, like `std::mem::swap()` inline in your code. Furthermore, a few particularly handy names, like `Vec` and `Result`, are included in the

standard prelude and automatically imported. 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.

In XREF HERE, 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*. 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 Fern {
    pub roots: RootSet,
    pub stems: StemSet
}
```

A struct's fields, even private fields, are accessible throughout the module where the struct is declared. Outside the module, only public

fields are accessible.

It turns out that enforcing access control by module, rather than by class as in Java or C++, is surprisingly helpful for software design. It cuts down on boilerplate “getter” and “setter” methods, and it largely eliminates the need for anything like C++ `friend` declarations. A single module can define several types that work closely together, such as perhaps `frond::LeafMap` and `frond::LeafMapIter`, accessing each other’s private fields as needed, while still hiding those implementation details from the rest of your program.

Type aliases

As we’ve seen, the `type` keyword can be used like `typedef` in C++, to declare a new name for an existing type:

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

The type `Table` that we’re declaring here is shorthand for this particular kind of `HashMap`.

```
fn show(table: &Table) {  
    ...  
}
```

impl blocks

Methods are attached to types using `impl` blocks:

```
impl Cell {
```

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

The syntax is explained in [Chapter 8](#). An `impl` block can't be marked `pub`. Instead, individual methods are marked `pub` to make them visible outside the current module.

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, `UPPERCASE_NAMES` are conventional for constants:

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

The `static` keyword introduces a static item, which is 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, or any time you'll need to borrow a reference to the constant value.

There are no `mut` constants. Statics can be marked `mut`, but as discussed in [Chapter 4](#), Rust has no way to enforce its rules about exclusive access on `mut` statics. They are, therefore, inherently non-thread-safe, and safe code can't use them at all:

```
static mut PACKETS_SERVED: usize = 0;

println!("{} served", PACKETS_SERVED); // error: use of mutable static
```

Rust discourages global mutable state. For a discussion of the alternatives, see [XREF HERE](#).

Modules

We've already talked about these quite a bit. As we've seen, a module can contain submodules, which can be public or private, like any other named item.

Imports

`use` declarations are items too. Even though they're just aliases, they can be public:

```
// in plant_structures/mod.rs

...
pub use self::leaves::Leaf;
pub use self::roots::Root;
```

This means that `Leaf` and `Root` are public items 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.

extern blocks

These declare a collection of functions written in some other language (typically C or C++), so that your Rust code can call them. We'll cover `extern` blocks in XREF HERE.

Rust warns about items that are declared, but never used:

```
warning: function is never used: `is_square`  
--> src/crates_unused_items.rs:23:9  
|  
23 | /      pub fn is_square(root: &Root) -> bool {  
24 | |          root.cross_section_shape().is_square()  
25 | |      }  
| |_____ ^  
|
```

This warning can be puzzling, because there are two very different possible causes. Perhaps this function really is dead code at the moment. Or, maybe you meant to use it in other crates. In that case, you need to mark it *and all enclosing modules* as public.

Turning a Program into a Library

As your fern simulator starts to take off, you decide you need more than a single program. Suppose you've got one command-line program that runs the simulation and saves results in a file. Now, you want to write other programs for performing scientific analysis of the saved results, displaying 3D renderings of the growing plants in real time, rendering photorealistic pictures, and so on. All these programs need to share the basic fern simulation code. You need to make a library.

The first step is to factor your existing project into two parts: a library crate, which contains all the shared code, and an executable, which contains the code that's only needed for your existing command-line program.

To show how you can do this, let's use a grossly simplified example program:

```
struct Fern {
    size: f64,
    growth_rate: f64
}

impl Fern {
    /// Simulate a fern growing for one day.
    fn grow(&mut self) {
        self.size *= 1.0 + self.growth_rate;
    }
}

/// Run a fern simulation for some number of days.
fn run_simulation(fern: &mut Fern, days: usize) {
    for _ in 0 .. days {
        fern.grow();
    }
}

fn main() {
    let mut fern = Fern {
        size: 1.0,
        growth_rate: 0.001
    };
    run_simulation(&mut fern, 1000);
    println!("final fern size: {}", fern.size);
}
```

We'll assume that this program has a trivial *Cargo.toml* file:

```
[package]
name = "fern_sim"
version = "0.1.0"
authors = ["You <you@example.com>"]
edition = "2018"
```

Turning this program into a library is easy. Here are the steps:

1. Rename the file `src/main.rs` to `src/lib.rs`.
2. Add the `pub` keyword to items in `src/lib.rs` that will be public features of our library.
3. Move the `main` function to a temporary file somewhere. We'll come back to it in a minute.

The resulting `src/lib.rs` file looks like this:

```
pub struct Fern {
    pub size: f64,
    pub growth_rate: f64
}

impl Fern {
    /// Simulate a fern growing for one day.
    pub fn grow(&mut self) {
        self.size *= 1.0 + self.growth_rate;
    }
}

/// Run a fern simulation for some number of days.
pub fn run_simulation(fern: &mut Fern, days: usize) {
    for _ in 0 .. days {
        fern.grow();
    }
}
```

Note that we didn't need to change anything in `Cargo.toml`. This is because our minimal `Cargo.toml` file leaves Cargo to its default behavior. By default, `cargo build` looks at the files in our source directory and figures out what to build. When it sees the file `src/lib.rs`, it knows to build a library.

The code in `src/lib.rs` forms the *root module* of the library. Other crates that use our library can only access the public items of this root module.

The `src/bin` Directory

Getting the original command-line `fern_sim` program working again is also straightforward: Cargo has some built-in support for small programs that live in the same crate as a library.

In fact, Cargo itself is written this way. The bulk of the code is in a Rust library. The `cargo` command-line program that we've been using throughout this book is a thin wrapper program that calls out to the library for all the heavy lifting. Both the library and the command-line program **live in the same source repository**.

We can keep our program and our library in the same crate, too. Put this code into a file named `src/bin/efern.rs`:

```
use fern_sim::{Fern, run_simulation};

fn main() {
    let mut fern = Fern {
        size: 1.0,
        growth_rate: 0.001
    };
    run_simulation(&mut fern, 1000);
    println!("final fern size: {}", fern.size);
}
```

The `main` function is the one we set aside earlier. We've added a `use` declaration for some items from the `fern_sim` crate, `Fern` and `run_simulation`. In other words, we're using that crate as a library.

Because we've put this file into `src/bin`, Cargo will compile both the `fern_sim` library and this program the next time we run `cargo build`. We can run the `efern` program using `cargo run --bin efern`. Here's what it looks like, using `--verbose` to show the commands Cargo is running:

```
$ cargo build --verbose
Compiling fern_sim v0.1.0 (file:///.../fern_sim)
  Running `rustc src/lib.rs --crate-name fern_sim --crate-type lib ...`
  Running `rustc src/bin/efern.rs --crate-name efern --crate-type bin ...`
$ cargo run --bin efern --verbose
  Fresh fern_sim v0.1.0 (file:///.../fern_sim)
```

```
Running `target/debug/efern`
final fern size: 2.7169239322355985
```

We still didn't have to make any changes to *Cargo.toml*, because again, Cargo's default is to look at your source files and figure things out. It automatically treats *.rs* files in *src/bin* as extra programs to build.

We can also build larger programs in the *src/bin* directory using subdirectories. Suppose we wanted to provide a second program the drew a fern on the screen, but the drawing code is large and modular, so it belongs in its own file. We can give the second program its own subdirectory:

```
fern_sim/
├── Cargo.toml
└── src/
    └── bin/
        ├── efern.rs
        └── draw_fern/
            ├── main.rs
            └── draw.rs
```

This has the advantage of letting larger binaries have their own sub-modules without cluttering up either the library code or the *src/bin* directory.

Of course, now that *fern_sim* is a library, we also have another option. We could have put this program in its own isolated project, in a completely separate directory, with its own *Cargo.toml* listing *fern_sim* as a dependency:

```
[dependencies]
fern_sim = { path = "../fern_sim" }
```

Perhaps that is what you'll do for other fern-simulating programs down the road. The *src/bin* directory is just right for simple programs like *efern* and *draw_fern*.

Attributes

Any item in a Rust program can be decorated with *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 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;
```

The full syntax of `#[cfg]` is specified in the [Rust Reference](#); the most commonly used options are listed here:

#[cfg (...)]	option	Enabled when
	test	Tests are enabled (compiling with <code>cargo test</code> or <code>rustc --test</code>).
	debug_assertions	Debug assertions are enabled (typically in nonoptimized builds).
	unix	Compiling for Unix, including macOS.
	windows	Compiling for Windows.
	target_pointer_width = "64"	Targeting a 64-bit platform. The other possible value is "32".
	target_arch = "x86_64"	Targeting x86-64 in particular. Other values: "x86", "arm", "aarch64", "powerpc", "powerpc64", "mips".
	target_os = "macos"	Compiling for macOS. Other values: "windows", "ios", "android", "linux", "openbsd", "netbsd", "dragonfly", "bitrig".
	feature = "robots"	The user-defined feature named "robots" is enabled (compiling with <code>cargo build --feature robots</code> or <code>rustc --cfg feature='robots'</code>). Features are declared in the [features] section of <code>Cargo.toml</code> .
	not(A)	A is not satisfied. To provide two different implementations of a function, mark one with <code>#![cfg(X)]</code> and the other with <code>#![cfg(not(X))]</code> .
	all(A,B)	Both A and B are satisfied (the equivalent of <code>&&</code>).
	any(A,B)	Either A or B is satisfied (the equivalent of <code> </code>).

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) {
    ...
}
```

There's one situation where inlining *won't* happen without `#![inline]`. When a function or method defined in one crate is called in another crate,

Rust won't inline it unless it's generic (it has type parameters) or it's explicitly marked `#[inline]`.

Otherwise, the compiler 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 `#[test]` and `#[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 `#`, like this:

```
// 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 just `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. Some attributes always use the `#!` syntax because they can only be applied to a whole crate.

For example, the `#![feature]` attribute is used to turn on *unstable* features of the Rust language and libraries, features that are experimental, and therefore might have bugs or might be changed or removed in the future. For instance, as we're writing this, Rust has experimental support for tracing the expansion of macros like `assert!`, but since this support is experimental, you can only use it by (1) installing the Nightly version of Rust and (2) explicitly declaring that your crate uses macro tracing:

```
#![feature(trace_macros)]  
  
fn main() {  
    // I wonder what actual Rust code this use of assert_eq!  
    // gets replaced with!  
    trace_macros!(true);  
    assert_eq!(10*10*10 + 9*9*9, 12*12*12 + 1*1*1);  
    trace_macros!(false);  
}
```

Over time, the Rust team sometimes *stabilizes* an experimental feature, so that it becomes a standard part of the language. The `#![feature]` attribute then becomes superfluous, and Rust generates a warning advising you to remove it.

Tests and Documentation

As we saw in XREF HERE, 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; 0 filtered out
```

(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 panics, 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!
}
```

You can also return a `Result` from your tests. As long as the error variant is `Debug`, which is usually the case, you can simply return a `Result` without `?,`

.unwrap(), or .expect(). The test will succeed if the return value is Ok and fail if it's an Err.

```
use std::num::ParseIntError;

// This test will pass if "1024" is a valid number, which it is. Note the
lack
// of any handling of the `Result` in the test itself.
#[test]
fn test_int_parsing() -> Result<i32, ParseIntError> {
    i32::from_str_radix("1024", 10)
}
```

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 run-time 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() {
    use std::f64::consts::PI;
    assert!(roughly_equal(PI.sin(), 0.0));
}
```

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

```
$ cargo build
Compiling math_test v0.1.0 (file:///.../math_test)
warning: function is never used: `roughly_equal`
--> src/crates_unused_testing_function.rs:7:1
```

```
|  
7 | / fn roughly_equal(a: f64, b: f64) -> bool {  
8 | |     (a - b).abs() < 1e-6  
9 | | }  
| |_  
|= note: #[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() {  
        use std::f64::consts::PI;  
        assert!(roughly_equal(PI.sin(), 0.0));  
    }  
}
```

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. (To disable this, either run a single test, `cargo test testname`; or set the environment variable `RUST_TEST_THREADS` to 1.) This means that, technically, the Mandelbrot program we showed in XREF HERE was not the second multithreaded program in that chapter, but the third! The `cargo test` run in XREF HERE 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

use fern_sim::Terrarium;
use std::time::Duration;

#[test]
fn test_fiddlehead_unfurling() {
    let mut world = Terrarium::load("tests/unfurl_files/fiddlehead.tm");
    assert!(world.fern(0).is_furled());
    let one_hour = Duration::from_secs(60 * 60);
    world.apply_sunlight(one_hour);
    assert!(world.fern(0).is_fully_unfurled());
}
```

Integration tests are valuable in part because they see your crate from the outside, just as a user would. They test the crate's public API.

`cargo test` runs both unit tests and integration tests. To run only the integration tests in a particular file—for example, `tests/unfurl.rs`—use the command `cargo test --test unfurl`.

Documentation

The command `cargo doc` creates HTML documentation for your library:

```
$ cargo doc --no-deps --open
Documenting fern_sim v0.1.0 (file:///.../fern_sim)
```

The `--no-deps` option tells Cargo to generate documentation only for `fern_sim` itself, and not for all the crates it depends on.

The `--open` option tells Cargo to open the documentation in your browser afterward.

You can see the result in [Figure 7-2](#). Cargo saves the new documentation files in `target/doc`. The starting page is `target/doc/fern_sim/index.html`.

Click or press 'S' to search, '?' for more options...

Crate fern_sim

[**-**] [**src**]

[**-**] Simulate the growth of ferns, from the level of individual cells on up.

Reexports

pub use plant_structures::Fern;

pub use simulation::Terrarium;

Modules

`cells` The simulation of biological cells, which is as low-level as we go.

`plant_structures` Higher-level biological structures.

`simulation` Overall simulation control.

`spores` Fern reproduction.

Figure 7-2. Example of documentation generated by rustdoc

The documentation is generated from the `pub` features of your library, plus any *doc comments* you've attached to them. We've seen a few doc comments in this chapter already. They look like comments:

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

But when Rust sees comments that start with three slashes, it treats them as a `#[doc]` attribute instead. Rust treats the preceding example exactly the same as this:

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

When you compile or test a library, these attributes are ignored. When you generate documentation, doc comments on public features are included in the output.

Likewise, comments starting with `//!` are treated as `#![doc]` attributes, and are attached to the enclosing feature, typically a module or crate. For example, your `fern_sim/src/lib.rs` file might begin like this:

```
//! Simulate the growth of ferns, from the level of  
//! individual cells on up.
```

The content of a doc comment is treated as Markdown, a shorthand notation for simple HTML formatting. Asterisks are used for ***italics*** and ****bold type****, a blank line is treated as a paragraph break, and so on. However, you can also fall back on HTML; any HTML tags in your doc comments are copied through verbatim into the documentation.

You can use `backticks` to set off bits of code in the middle of running text. In the output, these snippets will be formatted in a fixed-width font. Larger code samples can be added by indenting four spaces.

```
// A block of code in a doc comment:  
///  
///     if everything().works() {  
///         println!("ok");  
///     }
```

You can also use Markdown fenced code blocks. This has exactly the same effect.

```
// Another snippet, the same code, but written differently:  
///  
/// ...  
/// if everything().works() {  
///     println!("ok");  
/// }  
/// ...
```

Whichever format you use, an interesting thing happens when you include a block of code in a doc comment. Rust automatically turns it into a test.

Doc-Tests

When you run tests in a Rust library crate, Rust checks that all the code that appears in your documentation actually runs and works. It does this by taking each block of code that appears in a doc comment, compiling it as a separate executable crate, linking it with your library, and running it.

Here is a standalone example of a doc-test. Create a new project by running `cargo new --lib ranges` (the `--lib` flag tells Cargo we're creating a library crate, not an executable crate) and put the following code in `ranges/src/lib.rs`:

```
use std::ops::Range;

/// Return true if two ranges overlap.
///
///     assert_eq!(ranges::overlap(0..7, 3..10), true);
///     assert_eq!(ranges::overlap(1..5, 101..105), false);
///
/// If either range is empty, they don't count as overlapping.
///
///     assert_eq!(ranges::overlap(0..0, 0..10), false);
///
pub fn overlap(r1: Range<usize>, r2: Range<usize>) -> bool {
    r1.start < r1.end && r2.start < r2.end &&
        r1.start < r2.end && r2.start < r1.end
}
```

The two small blocks of code in the doc comment appear in the documentation generated by `cargo doc`, as shown in [Figure 7-3](#).

Function ranges::overlap

[[-\]](#) [[src](#)]

```
pub fn overlap(r1: Range<usize>, r2: Range<usize>) -> bool
```

[[-](#)] Return true if two ranges overlap.

```
assert_eq!(ranges::overlap(0..7, 3..10), true);  
assert_eq!(ranges::overlap(1..5, 101..105), false);
```

If either range is empty, they don't count as overlapping.

```
assert_eq!(ranges::overlap(0..0, 0..10), false);
```

Figure 7-3. Documentation showing some doc-tests

They also become two separate tests:

```
$ cargo test  
Compiling ranges v0.1.0 (file:///.../ranges)  
...  
Doc-tests ranges
```

```
running 2 tests
test overlap_0 ... ok
test overlap_1 ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

If you pass the `--verbose` flag to Cargo, you'll see that it's using `rustdoc --test` to run these two tests. Rustdoc stores each code sample in a separate file, adding a few lines of boilerplate code, to produce two programs. Here's the first:

```
use ranges;
fn main() {
    assert_eq!(ranges::overlap(0..7, 3..10), true);
    assert_eq!(ranges::overlap(1..5, 101..105), false);
}
```

And here's the second:

```
use ranges;
fn main() {
    assert_eq!(ranges::overlap(0..0, 0..10), false);
}
```

The tests pass if these programs compile and run successfully.

These two code samples contain assertions, but that's just because in this case, the assertions make decent documentation. The idea behind doc-tests is not to put all your tests into comments. Rather, you write the best possible documentation, and Rust makes sure the code samples in your documentation actually compile and run.

Very often a minimal working example includes some details, such as imports or setup code, that are necessary to make the code compile, but just aren't important enough to show in the documentation. To hide a line of a code sample, put a # followed by a space at the beginning of that line:

```
/// Let the sun shine in and run the simulation for a given
/// amount of time.
///
///     # use fern_sim::Terrarium;
///     # use std::time::Duration;
///     # let mut tm = Terrarium::new();
///     tm.apply_sunlight(Duration::from_secs(60));
///
pub fn apply_sunlight(&mut self, time: Duration) {
    ...
}
```

Sometimes it's helpful to show a complete sample program in documentation, including a `main` function. Obviously, if those pieces of code appear in your code sample, you do not also want Rustdoc to add them automatically. The result wouldn't compile. Rustdoc therefore treats any code block containing the exact string `fn main` as a complete program, and doesn't add anything to it.

Testing can be disabled for specific blocks of code. To tell Rust to compile your example, but stop short of actually running it, use a fenced code block with the `no_run` annotation:

```
/// Upload all local terrariums to the online gallery.
///
/// ````no_run
/// let mut session = fern_sim::connect();
/// session.upload_all();
/// ```
pub fn upload_all(&mut self) {
    ...
}
```

If the code isn't even expected to compile, use `ignore` instead of `no_run`. If the code block isn't Rust code at all, use the name of the language, like `c++` or `sh`, or `text` for plain text. `rustdoc` doesn't know the names of hundreds of programming languages; rather, it treats any annotation it doesn't recognize as indicating that the code block isn't Rust. This disables code highlighting as well as doc-testing.

Specifying Dependencies

We've seen one way of telling Cargo where to get source code for crates your project depends on: by version number.

```
image = "0.6.1"
```

There are several ways to specify dependencies, and some rather nuanced things you might want to say about which versions to use, so it's worth spending a few pages on this.

First of all, you may want to use dependencies that aren't published on crates.io at all. One way to do this is by specifying a Git repository URL and revision:

```
image = { git = "https://github.com/Piston IMAGE.git", rev = "528f19c" }
```

This particular crate is open source, hosted on GitHub, but you could just as easily point to a private Git repository hosted on your corporate network. As shown here, you can specify the particular `rev`, `tag`, or `branch` to use. (These are all ways of telling Git which revision of the source code to check out.)

Another alternative is to specify a directory that contains the crate's source code:

```
image = { path = "vendor/image" }
```

This is convenient when your team has a single version control repository that contains source code for several crates, or perhaps the entire dependency graph. Each crate can specify its dependencies using relative paths.

Having this level of control over your dependencies is powerful. If you ever decide that any of the open source crates you use isn't exactly to your

liking, you can trivially fork it: just hit the “Fork” button on GitHub and change one line in your *Cargo.toml* file. Your next `cargo build` will seamlessly use your fork of the crate instead of the official version.

Versions

When you write something like `image = "0.13.0"` in your *Cargo.toml* file, Cargo interprets this rather loosely. It uses the most recent version of `image` that is considered compatible with version 0.13.0.

The compatibility rules are adapted from [Semantic Versioning](#).

- A version number that starts with 0.0 is so raw that Cargo never assumes it’s compatible with any other version.
- A version number that starts with 0.*x*, where *x* is nonzero, is considered compatible with other point releases in the 0.*x* series. We specified `image` version 0.6.1, but Cargo would use 0.6.3 if available. (This is not what the Semantic Versioning standard says about 0.*x* version numbers, but the rule proved too useful to leave out.)
- Once a project reaches 1.0, only new major versions break compatibility. So if you ask for version 2.0.1, Cargo might use 2.17.99 instead, but not 3.0.

Version numbers are flexible by default because otherwise the problem of which version to use would quickly become overconstrained. Suppose one library, `libA`, used `num = "0.1.31"` while another, `libB`, used `num = "0.1.29"`. If version numbers required exact matches, no project would be able to use those two libraries together. Allowing Cargo to use any compatible version is a much more practical default.

Still, different projects have different needs when it comes to dependencies and versioning. You can specify an exact version or range of versions by using operators:

Cargo.toml line	Meaning
<code>image = "=0.10.0"</code>	Use only the exact version 0.10.0
<code>image = ">=1.0.5"</code>	Use 1.0.5 or <i>any</i> higher version (even 2.9, if it's available)
<code>image = ">1.0.5 <1.1.9"</code>	Use a version that's higher than 1.0.5, but lower than 1.1.9
<code>image = "<=2.7.10"</code>	Use any version up to 2.7.10

Another version specification you'll occasionally see is the wildcard `*`. This tells Cargo that any version will do. Unless some other *Cargo.toml* file contains a more specific constraint, Cargo will use the latest available version. [The Cargo documentation at doc.crates.io](#) covers version specifications in even more detail.

Note that the compatibility rules mean that version numbers can't be chosen purely for marketing reasons. They actually mean something. They're a contract between a crate's maintainers and its users. If you maintain a crate that's at version 1.7, and you decide to remove a function or make any other change that isn't fully backward compatible, you must bump your version number to 2.0. If you were to call it 1.8, you'd be claiming that the new version is compatible with 1.7, and your users might find themselves with broken builds.

Cargo.lock

The version numbers in *Cargo.toml* are deliberately flexible, yet we don't want Cargo to upgrade us to the latest library versions every time we build. Imagine being in the middle of an intense debugging session when suddenly `cargo build` upgrades you to a new version of a library. This could be incredibly disruptive. Anything changing in the middle of debugging is bad. In fact, when it comes to libraries, there's never a good time for an unexpected change.

Cargo therefore has a built-in mechanism to prevent this. The first time you build a project, Cargo outputs a *Cargo.lock* file that records the exact version of every crate it used. Later builds will consult this file and continue to use the same versions. Cargo upgrades to newer versions only when you

tell it to, either by manually bumping up the version number in your *Cargo.toml* file, or by running `cargo update`:

```
$ cargo update
  Updating registry `https://github.com/rust-lang/crates.io-index'
  Updating libc v0.2.7 -> v0.2.11
  Updating png v0.4.2 -> v0.4.3
```

`cargo update` only upgrades to the latest versions that are compatible with what you've specified in *Cargo.toml*. If you've specified `image = "0.6.1"`, and you want to upgrade to version 0.10.0, you'll have to change that in *Cargo.toml*. The next time you build, Cargo will update to the new version of the `image` library and store the new version number in *Cargo.lock*.

The preceding example shows Cargo updating two crates that are hosted on crates.io. Something very similar happens for dependencies that are stored in Git. Suppose our *Cargo.toml* file contains this:

```
image = { git = "https://github.com/Piston IMAGE.git", branch = "master" }
```

`cargo build` will not pull new changes from the Git repository if it sees that we've got a *Cargo.lock* file. Instead, it reads *Cargo.lock* and uses the same revision as last time. But `cargo update` will pull from `master`, so that our next build uses the latest revision.

Cargo.lock is automatically generated for you, and you normally won't edit it by hand. Nonetheless, if your project is an executable, you should commit *Cargo.lock* to version control. That way, everyone who builds your project will consistently get the same versions. The history of your *Cargo.lock* file will record your dependency updates.

If your project is an ordinary Rust library, don't bother committing *Cargo.lock*. Your library's downstream users will have *Cargo.lock* files that contain version information for their entire dependency graph; they will ignore your library's *Cargo.lock* file. In the rare case that your project is a

shared library (i.e., the output is a `.dll`, `.dylib`, or `.so` file), there is no such downstream `cargo` user, and you should therefore commit `Cargo.lock`.

`Cargo.toml`'s flexible version specifiers make it easy to use Rust libraries in your project and maximize compatibility among libraries. `Cargo.lock`'s bookkeeping supports consistent, reproducible builds across machines. Together, they go a long way toward helping you avoid dependency hell.

Publishing Crates to crates.io

You've decided to publish your fern-simulating library as open source software. Congratulations! This part is easy.

First, make sure Cargo can pack the crate for you.

```
$ cargo package
warning: manifest has no description, license, license-file, documentation,
homepage or repository. See http://doc.crates.io/manifest.html#package-
metadata
for more info.

    Packaging fern_sim v0.1.0 (file:///.../fern_sim)
    Verifying fern_sim v0.1.0 (file:///.../fern_sim)
    Compiling fern_sim v0.1.0 (file:///.../fern_sim/target/package/fern_sim-
0.1.0)
```

The `cargo package` command creates a file (in this case, `target/package/fern_sim-0.1.0.crate`) containing all your library's source files, including `Cargo.toml`. This is the file that you'll upload to crates.io to share with the world. (You can use `cargo package --list` to see which files are included.) Cargo then double-checks its work by building your library from the `.crate` file, just as your eventual users will.

Cargo warns that the `[package]` section of `Cargo.toml` is missing some information that will be important to downstream users, such as the license under which you're distributing the code. The URL in the warning is an excellent resource, so we won't explain all the fields in detail here. In short, you can fix the warning by adding a few lines to `Cargo.toml`:

```
[package]
name = "fern_sim"
version = "0.1.0"
edition = "2018"
authors = [ "You <you@example.com>" ]
license = "MIT"
homepage = "https://fernsm.example.com/"
repository = "https://gitlair.com/sporeador/fern_sim"
documentation = "http://fernsm.example.com/docs"
description = """
Fern simulation, from the cellular level up.
"""

```

NOTE

Once you publish this crate on crates.io, anyone who downloads your crate can see the *Cargo.toml* file. So if the `authors` field contains an email address that you'd rather keep private, now's the time to change it.

Another problem that sometimes arises at this stage is that your *Cargo.toml* file might be specifying the location of other crates by `path`, as shown in “[Specifying Dependencies](#)”:

```
image = { path = "vendor/image" }
```

For you and your team, this might work fine. But naturally, when other people download the `fern_sim` library, they will not have the same files and directories on their computer that you have. Cargo therefore *ignores* the `path` key in automatically downloaded libraries, and this can cause build errors. The fix, however, is straightforward: if your library is going to be published on crates.io, its dependencies should be on crates.io too. Specify a version number instead of a `path`:

```
image = "0.13.0"
```

If you prefer, you can specify both a `path`, which takes precedence for your own local builds, and a `version` for all other users:

```
image = { path = "vendor/image", version = "0.13.0" }
```

Of course, in that case it's your responsibility to make sure that the two stay in sync.

Lastly, before publishing a crate, you'll need to log in to crates.io and get an API key. This step is straightforward: once you have an account on crates.io, your “Account Settings” page will show a `cargo login` command, like this one:

```
$ cargo login 5j0dV54BjlXBpUUbfIj7G9DvNl1vsWW1
```

Cargo saves the key in a configuration file, and the API key should be kept secret, like a password. So run this command only on a computer you control.

That done, the final step is to run `cargo publish`:

```
$ cargo publish
Updating registry `https://github.com/rust-lang/crates.io-index'
Uploading fern_sim v0.1.0 (file:///.../fern_sim)
```

With this, your library joins thousands of others on crates.io.

Workspaces

As your project continues to grow, you end up writing many crates. They live side by side in a single source repository:

```
fernsoft/
├─ .git/...
```

```
└── fern_sim/
    ├── Cargo.toml
    ├── Cargo.lock
    └── src/...
    └── target/...
└── fern_img/
    ├── Cargo.toml
    ├── Cargo.lock
    └── src/...
    └── target/...
└── fern_video/
    ├── Cargo.toml
    ├── Cargo.lock
    └── src/...
    └── target/...
```

The way Cargo works, each crate has its own build directory, `target`, which contains a separate build of all that crate's dependencies. These build directories are completely independent. Even if two crates have a common dependency, they can't share any compiled code. This is wasteful.

You can save compilation time and disk space by using a *Cargo workspace*, a collection of crates that share a common build directory and *Cargo.lock* file.

All you need to do is create a *Cargo.toml* file in your repository's root directory and put these lines in it:

```
[workspace]
members = ["fern_sim", "fern_img", "fern_video"]
```

where `fern_sim` etc. are the names of the subdirectories containing your crates. Delete any leftover *Cargo.lock* files and *target* directories that exist in those subdirectories.

Once you've done this, `cargo build` in any crate will automatically create and use a shared build directory under the root directory (in this case, *fernsoft/target*). The command `cargo build --all` builds all crates in the current workspace. `cargo test` and `cargo doc` accept the `--all` option as well.

More Nice Things

In case you're not delighted yet, the Rust community has a few more odds and ends for you:

- When you publish an open source crate on [crates.io](#), your documentation is automatically rendered and hosted on *docs.rs* thanks to Onur Aslan.
- If your project is on GitHub, Travis CI can build and test your code on every push. It's surprisingly easy to set up; see [travis-ci.org](#) for details. If you're already familiar with Travis, this *.travis.yml* file will get you started:

```
language: rust
```

```
rust:
```

```
  - stable
```

- You can generate a *README.md* file from your crate's top-level doc-comment. This feature is offered as a third-party Cargo plugin by Livio Ribeiro. Run `cargo install cargo-readme` to install the plugin, then `cargo readme --help` to learn how to use it.

We could go on.

Rust is new, but it's designed to support large, ambitious projects. It has great tools and an active community. System programmers *can* have nice things.

Chapter 8. Structs

A NOTE FOR EARLY RELEASE READERS

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

This will be the 9th chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at jimb@red-bean.com.

Long ago, when shepherds wanted to see if two herds of sheep were isomorphic, they would look for an explicit isomorphism.

—John C. Baez and James Dolan, “[Categorification](#)”

Rust structs, sometimes called *structures*, resemble `struct` types in C and C++, classes in Python, and objects in JavaScript. A struct assembles several values of assorted types together into a single value, so you can deal with them as a unit. Given a struct, you can read and modify its individual components. And a struct can have methods associated with it that operate on its components.

Rust has three kinds of struct types, *named-field*, *tuple-like*, and *unit-like*, which differ in how you refer to their components: a named-field struct gives a name to each component, whereas a tuple-like struct identifies them by the order in which they appear. Unit-like structs have no components at all; these are not common, but more useful than you might think.

In this chapter, we’ll explain each kind in detail, and show what they look like in memory. We’ll cover how to add methods to them, how to define

generic struct types that work with many different component types, and how to ask Rust to generate implementations of common handy traits for your structs.

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. The convention in Rust is for all types, structs included, to have names that capitalize the first letter of each word, like `GrayscaleMap`, a convention called *CamelCase*. Fields and methods are lowercase, with words separated by underscores. This is called *snake_case*.

You can construct a value of this type with a *struct expression*, like this:

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

A struct expression starts with the type name (`GrayscaleMap`), and lists the name and value of each field, all enclosed in curly braces. There's also shorthand for populating fields from local variables or arguments with the same name:

```
fn new_map(size: (usize, usize), pixels: Vec<u8>) -> GrayscaleMap {
    assert_eq!(pixels.len(), size.0 * size.1);
```

```
    GrayscaleMap { pixels, size }  
}
```

The struct expression `GrayscaleMap { pixels, size }` is short for `GrayscaleMap { pixels: pixels, size: size }`. You can use `key: value` syntax for some fields and shorthand for others in the same struct expression.

To access a struct's fields, use the familiar `.` operator:

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

Like all other items, structs are private by default, visible only in the module where they're declared. You can make a struct visible outside its module by prefixing its definition with `pub`. The same goes for each of its fields, which are also private by default:

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

Even if a struct is declared `pub`, its fields can be private:

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

Other modules can use this struct and any public methods it might have, but can't access the private fields by name or use struct expressions to create new `GrayscaleMap` values. That is, creating a struct value requires all the struct's fields to be visible. This is why you can't write a struct expression

to create a new `String` or `Vec`. These standard types are structs, but all their fields are private. To create one, you must use public methods like `Vec::new()`.

When creating a named-field struct value, you can use another struct of the same type to supply values for fields you omit. In a struct expression, if the named fields are followed by `.. EXPR`, then any fields not mentioned take their values from EXPR, which must be another value of the same struct type. Suppose we have a struct representing a monster in a game:

```
// In this game, brooms are monsters. You'll see.
struct Broom {
    name: String,
    height: f32,
    health: f32,
    position: (f32, f32, f32),
    intent: BroomIntent
}

/// Two possible alternatives for what a `Broom` could be working on.
#[derive(Copy, Clone)]
enum BroomIntent { FetchWater, DumpWater }
```

The best fairy tale for programmers is *The Sorcerer's Apprentice*: a novice magician enchants a broom to do his work for him, but doesn't know how to stop it when the job is done. Chopping the broom in half with an axe just produces two brooms, each of half the size, but continuing the task with the same blind dedication as the original:

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

    // Initialize `broom2` mostly from `broom1`. Since `String` is not
    // `Copy`, we must clone `name` explicitly.
    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);
```

Individual elements of a tuple-like struct may be public or not:

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

The expression `Bounds(1024, 768)` looks like a function call, and in fact it is: defining the type also implicitly defines a function:

```
fn Bounds(elem0: usize, elem1: usize) -> Bounds { ... }
```

At the most fundamental level, named-field and tuple-like structs are very similar. The choice of which to use comes down to questions of legibility, ambiguity, and brevity. If you will use the `.` operator to get at a value's components much at all, identifying fields by name provides the reader more information, and is probably more robust against typos. If you will usually use pattern matching to find the elements, tuple-like structs can work nicely.

Tuple-like structs are good for *newtypes*, structs with a single component that you define to get stricter type checking. For example, if you are working with ASCII-only text, you might define a newtype like this:

```
struct Ascii(Vec<u8>);
```

Using this type for your ASCII strings is much better than simply passing around `Vec<u8>` buffers and explaining what they are in the comments. The newtype helps Rust catch mistakes where some other byte buffer is passed to a function expecting ASCII text. We'll give an example of using newtypes for efficient type conversions in XREF HERE.

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 the `..` range operator in “[Fields and Elements](#)”. 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, earlier in the chapter we defined this struct:

```
struct GrayscaleMap {  
    pixels: Vec<u8>,
```

```
    size: (u8, u8)  
}
```

A `GrayscaleMap` value is laid out in memory as diagrammed in Figure 8-1.

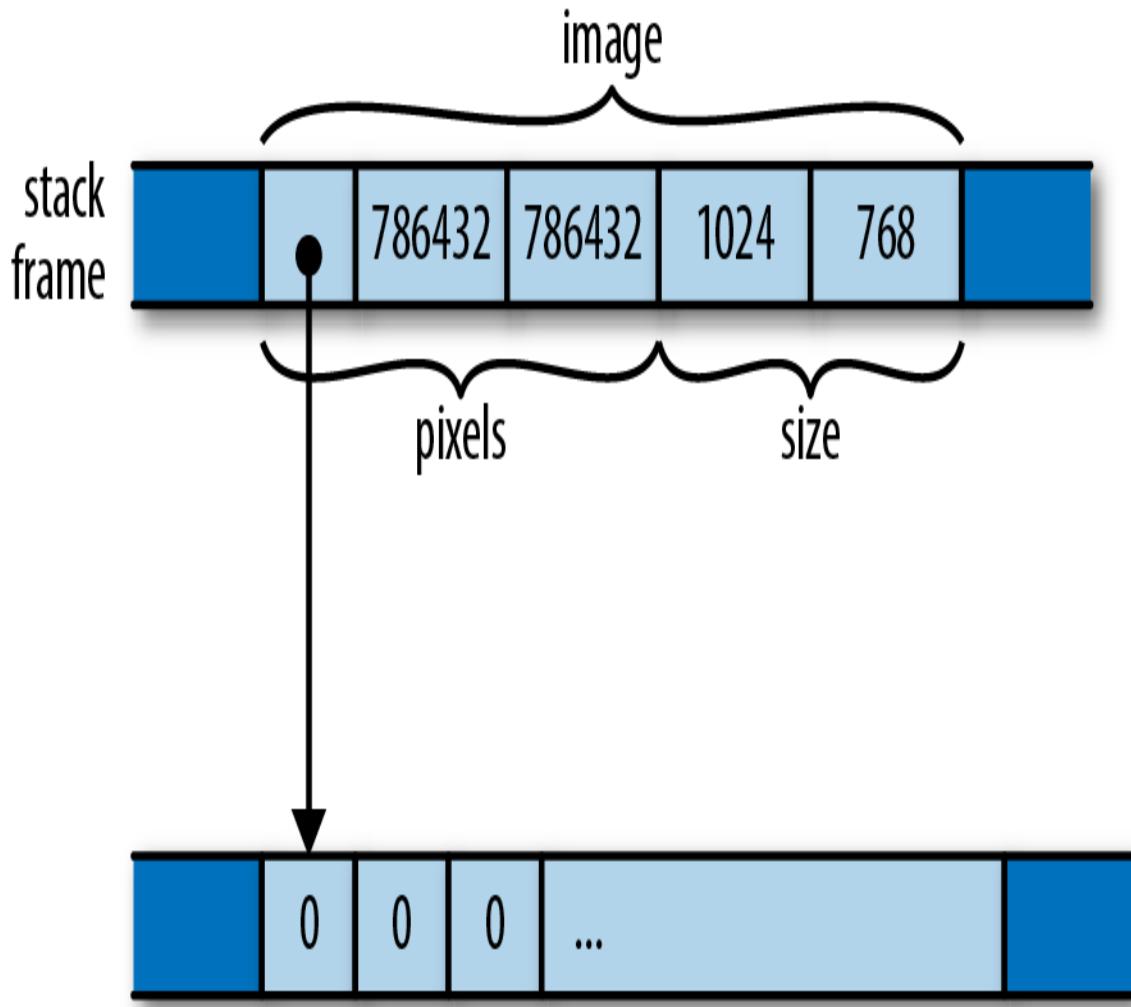


Figure 8-1. A `GrayscaleMap` structure in memory

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 put the `pixels` and `size` values each in their own heap-allocated blocks and have `GrayscaleMap`'s fields point at them, Rust

embeds `pixels` and `size` directly in the `GrayscaleMap` value. Only the heap-allocated buffer owned by the `pixels` vector remains in its own block.

You can ask Rust to lay out structures in a way compatible with C and C++, using the `#[repr(C)]` attribute. We'll cover this in detail in XREF HERE.

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.

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 first-in, first-out queue of characters.
pub 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.
    pub fn push(&mut self, 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.
    pub fn pop(&mut self) -> 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 struct type named at the top of the block. Here we've defined a public struct `Queue`, and then given it two public methods, `push` and `pop`.

Methods are also known as *associated functions*, since they're associated with a specific type. The opposite of an associated function is a *free function*, one that is not defined as an `impl` block's item.

Rust passes a method the value it's being called on as its first argument, which must have the special name `self`. Since `self`'s type is obviously the one named at the top of the `impl` block, or a reference to that, Rust lets you omit the type, and write `self`, `&self` or `&mut self` as shorthand for `self: Queue`, `self: &Queue` or `self: &mut Queue`. You can use the longhand forms if you like, but almost all Rust code uses the shorthand, as shown before.

In our example, the `push` and `pop` methods refer to the `Queue`'s 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 explicitly 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 take `&mut self`. However, when you call a method, you don't need to borrow the 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`, as if you had written `(&mut q).push(...)`, since that's what the `push` method's `self` requires.

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 {
    pub fn is_empty(&self) -> 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 {
    pub fn split(self) -> (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.

Passing Self as a Box, Rc, or Arc

A method's `self` argument can also be a `Box<Self>`, `Rc<Self>`, or `Arc<Self>`. Such a method can only be called on a value of the given pointer type. Calling the method passes ownership of the pointer to it.

You won't usually need to do this. A method that expects `self` by reference works fine when called on any of those pointer types:

```
let mut bq = Box::new(Queue::new());

// `Queue::push` expects a `&mut Queue`, but `bq` is a `Box<Queue>`.
// This is fine: Rust borrows a `&mut Queue` from the `Box` for the
// duration of the call.
bq.push('■');
```

For method calls and field access, Rust automatically borrows a reference from pointer types like `Box`, `Rc`, and `Arc`, so `&self` and `&mut self` are almost always the right thing in a method signature, along with the occasional `self`.

But what if the method's purpose involves managing ownership of the pointer? Suppose we have a tree of nodes like this, some sort of drastically simplified XML:

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

struct Node {
    tag: String,
    children: Vec<Rc<Node>>
}

impl Node {
    fn new(tag: &str) -> Node {
        Node {
            tag: tag.to_string(),
            children: vec![],

        }
    }
}
```

Each node has a tag, to indicate what sort of node it is, and a vector of children, held by reference-counted pointers to permit sharing and make their lifetimes a bit more flexible.

Usually, markup nodes have a method that appends a child to its own list, but for the moment, let's reverse the roles and give `Node` a method that appends it to some other `Node`'s children. We could write:

```
impl Node {
    fn append_to(self, parent: &mut Node) {
        parent.children.push(Rc::new(self));
    }
}
```

But this is unsatisfying. This method calls `Rc::new` to allocate a fresh heap location and copy `self` into it, but if the caller already has an `Rc<Node>`, all that is unnecessary: we should just increment the reference count and push the pointer onto the vector. Wasn't the whole point of `Rc` to enable sharing?

Instead, we can write this:

```
impl Node {
    fn append_to(self: Rc<Self>, parent: &mut Node) {
        parent.children.push(self);
    }
}
```

If the caller has an `Rc<Node>` at hand, it can call `append_to` directly, passing the `Rc` by value:

```
shared_node.append_to(&mut parent);
```

This passes ownership of `shared_node` to the method: no reference counts are adjusted, and there's certainly no new allocation.

If the caller needs to retain a pointer to the node for later use, then it can clone the `Rc` first:

```
shared_node.clone().append_to(&mut parent);
```

Cloning an `Rc` just bumps its reference count: there's still no heap allocation or copying. But when the call returns, both `shared_node` and `parent`'s vector of children are pointing to the same `Node`.

Finally, if the caller actually owns the `Node` outright, then it must create the `Rc` itself before passing it:

```
let owned = Node::new("owned directly");
Rc::new(owned).append_to(&mut parent);
```

Putting `Rc<Self>` into the signature of the `append_to` method makes the caller aware of `Node`'s requirements. The caller is then able to minimize allocation and reference-counting activity given its own needs:

- If it can pass ownership of the `Rc`, it simply hands over the pointer.
- If it needs to retain ownership of an `Rc`, it just bumps the reference count.
- Only if it owns the `Node` itself must it call `Rc::new` to allocate heap space and move the `Node` into it. Since `parent` will insist on referring to its children via `Rc<Node>` pointers, this was going to be necessary eventually.

Again, for most methods, `&self`, `&mut self` and `self` (by value) are all you need. But if a method's purpose is to affect the ownership of the value, using other pointer types for `self` can be just the right thing.

Static Methods

You can also define methods that don't take `self` as an argument at all. These become functions associated with the struct 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 {
    pub 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, you might need to skim hundreds of lines of member function definitions to be sure you haven't missed any of the class's data members; in Rust, they're all in one place.
- Although one can imagine fitting methods into the syntax for named-field structs, it's not so neat for tuple-like and unit-like structs. Pulling methods out into an `impl` block allows a single syntax for all three. In fact, Rust uses this same syntax for defining methods on types that are not structs at all, such as `enum` types and primitive types like `i32`. (The fact that any type can have methods is one reason Rust doesn't use the term *object* much, preferring to call everything a *value*.)
- The same `impl` syntax also serves neatly for implementing traits, which we'll go into in [Chapter 10](#).

Static Values

Another feature of languages like C# and Java that Rust adopts in its type system is the idea of values associated with a type, rather than a specific instance of that type. In Rust, these are known as *associated consts*.

As the name implies, associated consts are constant values. They're often used to specify commonly used values of a type. For instance, you could define a 2-dimensional vector for use in linear algebra with an associated unit vector:

```
pub struct Vector2 {
    x: f32,
    y: f32,
}

impl Vector2 {
    const ZERO: Vector2 = Vector2 { x: 0.0, y: 0.0 };
    const UNIT: Vector2 = Vector2 { x: 1.0, y: 0.0 };
}
```

These values are associated with the type itself, and you can use them without referring to another instance of `Vector2`. Much like static methods, they are accessed by naming the type with which they're associated, followed by their name:

```
let scaled = Vector2::UNIT.scaled_by(2.0);
```

Nor does an associated const have to be of the same type as the type it's associated with; we could use this feature to add IDs or names to types. For example, if there were several types similar to `Vector2` that needed to be written to a file and then loaded into memory later, an associated const could be used to add names or numeric ID which could be written next to the data to identify its type.

```
impl Vector2 {
    const NAME: &'static str = "Vector2";
    const ID: u32 = 18;
}
```

Generic Structs

Our earlier definition of `Queue` 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 struct 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 structs 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:

```
pub 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 fields of type `Vec<T>`.`”` For example, in `Queue<String>`, `T` is `String`, so `older` and `younger` have type `Vec<String>`. In `Queue<char>`, `T` is `char`, and we get a struct identical to the `char`-specific definition we started with. In fact, `Vec` itself is a generic struct, defined in just this way.

In generic struct definitions, the type names used in <angle brackets> are called *type parameters*. An `impl` block for a generic struct looks like this:

```
impl<T> Queue<T> {
    pub fn new() -> Queue<T> {
        Queue { older: Vec::new(), younger: Vec::new() }
    }

    pub fn push(&mut self, t: T) {
        self.younger.push(t);
    }

    pub 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 in the preceding code; 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 preceding code, `Self` would be `Queue<T>`, so we can abbreviate `Queue::new`’s definition a bit further:

```
pub 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; simply writing `Queue { ... }` was good enough. This is Rust’s type inference at work: since there’s only one type that works for that function’s return value—namely, `Queue<T>`—Rust supplies the parameter 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 explicit types as the basis from which it infers types within function bodies.

For static method calls, you can supply the type parameter explicitly using the turbofish `::<>` notation:

```
let mut q = Queue::<char>::new();
```

But in practice, you can usually just let Rust figure it out for you:

```
let mut q = Queue::new();
```

```
let mut r = Queue::new();

q.push("CAD"); // apparently a Queue<&'static str>
r.push(0.74); // apparently a Queue<f64>

q.push("BTC"); // Bitcoins per USD, 2019-6
r.push(13764.0); // Rust fails to detect irrational exuberance
```

In fact, this is exactly what we've been doing with `Vec`, another generic struct type, throughout the book.

It's not just structs that can be generic. Enums can take type parameters as well, with a very similar syntax. We'll show that in detail in “[Enums](#)”.

Structs with Lifetime Parameters

As we discussed in “[Structs Containing References](#)”, 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, 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. Rust always infers lifetime parameters for calls, so calls to `find_extrema` needn't mention them:

```

let a = [0, -3, 0, 15, 48];
let e = find_extrema(&a);
assert_eq!(*e.least, -3);
assert_eq!(*e.greatest, 48);

```

Because it's so common for the return type to use the same lifetime as an argument, Rust lets us omit the lifetimes when there's one obvious candidate. 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

Structs can be very easy to write:

```
struct Point {
```

```
    x: f64,  
    y: f64  
}
```

However, if you were to start using this `Point` type, you would quickly notice that it's a bit of a pain. As written, `Point` is not copyable or cloneable. You can't print it with `println!("{}:{}?", point);` and it does not support the `==` and `!=` operators.

Each of these features has a name in Rust—`Copy`, `Clone`, `Debug`, and `PartialEq`. They are called *traits*. In [Chapter 10](#), we'll show how to implement traits by hand for your own structs. But in the case of these standard traits, and several others, you don't need to implement them by hand unless you want some kind of custom behavior. Rust can automatically implement them for you, with mechanical accuracy. Just add a `#[derive]` attribute to the struct:

```
#[derive(Copy, Clone, Debug, PartialEq)]  
struct Point {  
    x: f64,  
    y: f64  
}
```

Each of these traits can be implemented automatically for a struct, provided that each of its fields implements the trait. We can ask Rust to derive `PartialEq` for `Point` because its two fields are both of type `f64`, which already implements `PartialEq`.

Rust can also derive `PartialOrd`, which would add support for the comparison operators `<`, `>`, `<=`, and `>=`. We haven't done so here, because comparing two points to see if one is "less than" the other is actually a pretty weird thing to do. There's no one conventional order on points. So we choose not to support those operators for `Point` values. Cases like this are one reason that Rust makes us write the `#[derive]` attribute rather than automatically deriving every trait it can. Another reason is that implementing a trait is automatically a public feature, so copyability,

cloneability, and so forth are all part of your struct's public API and should be chosen deliberately.

We'll describe Rust's standard traits in detail, and tell which ones are `#[derive]`able, in XREF HERE.

Interior Mutability

Mutability is like anything else: in excess, it causes problems, but you often want just a little bit of it. For example, say your spider robot control system has a central struct, `SpiderRobot`, that contains settings and I/O handles. It's set up when the robot boots, and the values never change:

```
pub struct SpiderRobot {
    species: String,
    web_enabled: bool,
    leg_devices: [fd::FileDesc; 8],
    ...
}
```

Every major system of the robot is handled by a different struct, and each one has a pointer back to the `SpiderRobot`:

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

pub struct SpiderSenses {
    robot: Rc<SpiderRobot>, // <-- pointer to settings and I/O
    eyes: [Camera; 32],
    motion: Accelerometer,
    ...
}
```

The structs for web construction, predation, venom flow control, and so forth also each have an `Rc<SpiderRobot>` smart pointer. Recall that `Rc` stands for [reference counting](#), and a value in an `Rc` box is always shared and therefore always immutable.

Now suppose you want to add a little logging to the `SpiderRobot` struct, using the standard `File` type. There's a problem: a `File` has to be `mut`. All the methods for writing to it require a `mut` reference.

This sort of situation comes up fairly often. What we need is a little bit of mutable data (a `File`) inside an otherwise immutable value (the `SpiderRobot` struct). This is called *interior mutability*. Rust offers several flavors of it; in this section, we'll discuss the two most straightforward types: `Cell<T>` and `RefCell<T>`, both in the `std::cell` module.

A `Cell<T>` is a struct that contains a single private value of type `T`. The only special thing about a `Cell` is that you can get and set the field even if you don't have `mut` access to the `Cell` itself:

- `Cell::new(value)` creates a new `Cell`, moving the given `value` into it.
- `cell.get()` returns a copy of the value in the `cell`.
- `cell.set(value)` stores the given `value` in the `cell`, dropping the previously stored value.

This method takes `self` as a non-`mut` reference:

```
fn set(&self, value: T)    // note: not `&mut self`
```

This is, of course, unusual for methods named `set`. By now, Rust has trained us to expect that we need `mut` access if we want to make changes to data. But by the same token, this one unusual detail is the whole point of `Cells`. They're simply a safe way of bending the rules on immutability—no more, no less.

Cells also have a few other methods, which you can read about [in the documentation](#).

A `Cell` would be handy if you were adding a simple counter to your `SpiderRobot`. You could write:

```

use std::cell::Cell;

pub struct SpiderRobot {
    ...
    hardware_error_count: Cell<u32>,
    ...
}

```

and then even non-mut methods of `SpiderRobot` can access that `u32`, using the `.get()` and `.set()` methods:

```

impl SpiderRobot {
    /// Increase the error count by 1.
    pub fn add_hardware_error(&self) {
        let n = self.hardware_error_count.get();
        self.hardware_error_count.set(n + 1);
    }

    /// True if any hardware errors have been reported.
    pub fn has_hardware_errors(&self) -> bool {
        self.hardware_error_count.get() > 0
    }
}

```

This is easy enough, but it doesn't solve our logging problem. `Cell` does *not* let you call `mut` methods on a shared value. The `.get()` method returns a copy of the value in the cell, so it works only if `T` implements [the Copy trait](#). For logging, we need a mutable `File`, and `File` isn't copyable.

The right tool in this case is a `RefCell`. Like `Cell<T>`, `RefCell<T>` is a generic type that contains a single value of type `T`. Unlike `Cell`, `RefCell` supports borrowing references to its `T` value:

- `RefCell::new(value)` creates a new `RefCell`, moving `value` into it.
- `ref_cell.borrow()` returns a `Ref<T>`, which is essentially just a shared reference to the value stored in `ref_cell`.

This method panics if the value is already mutably borrowed; see details to follow.

- `ref_cell.borrow_mut()` returns a `RefMut<T>`, essentially a mutable reference to the value in `ref_cell`.

This method panics if the value is already borrowed; see details to follow.

Again, `RefCell` has a few other methods, which you can find [in the documentation](#).

The two `borrow` methods panic only if you try to break the Rust rule that `mut` references are exclusive references. For example, this would panic:

```
let ref_cell: RefCell<String> = RefCell::new("hello".to_string());

let r = ref_cell.borrow();          // ok, returns a Ref<String>
let count = r.len();               // ok, returns "hello".len()
assert_eq!(count, 5);

let mut w = ref_cell.borrow_mut();  // panic: already borrowed
w.push_str(" world");
```

To avoid panicking, you could put these two borrows into separate blocks. That way, `r` would be dropped before you try to borrow `w`.

This is a lot like how normal references work. The only difference is that normally, when you borrow a reference to a variable, Rust checks *at compile time* to ensure that you're using the reference safely. If the checks fail, you get a compiler error. `RefCell` enforces the same rule using run-time checks. So if you're breaking the rules, you get a panic.

Now we're ready to put `RefCell` to work in our `SpiderRobot` type:

```
pub struct SpiderRobot {
    ...
    log_file: RefCell<File>,
    ...
}
```

```

impl SpiderRobot {
    /// Write a line to the log file.
    pub fn log(&self, message: &str) {
        let mut file = self.log_file.borrow_mut();
        // `writeln!` is like `println!`, but sends
        // output to the given file.
        writeln!(file, "{}", message).unwrap();
    }
}

```

The variable `file` has type `RefMut<File>`. It can be used just like a mutable reference to a `File`. For details about writing to files, see XREF HERE.

Cells are easy to use. Having to call `.get()` and `.set()` or `.borrow()` and `.borrow_mut()` is slightly awkward, but that's just the price we pay for bending the rules. The other drawback is less obvious and more serious: cells—and any types that contain them—are not thread-safe. Rust therefore will not allow multiple threads to access them at once. We'll describe thread-safe flavors of interior mutability in XREF HERE, when we discuss mutex in Rust, atomics, and global variables.

Whether a struct has named fields or is tuple-like, it is an aggregation of other values: if I have a `SpiderSenses` struct, then I have an `Rc` pointer to a shared `SpiderRobot` struct, and I have eyes, and I have an accelerometer, and so on. So the essence of a struct is the word “and”: I have an `X and a Y`. But what if there were another kind of type built around the word “or”? That is, when you have a value of such a type, you'd have *either* an `X or a Y`? Such types turn out to be so useful that they're ubiquitous in Rust, and they are the subject of the next chapter.

Chapter 9. Enums and Patterns

A NOTE FOR EARLY RELEASE READERS

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

This will be the 10th chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at jimb@red-bean.com.

Surprising how much computer stuff makes sense viewed as tragic deprivation of sum types (cf. deprivation of lambdas)

—Graydon Hoare

The first topic of this chapter 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. But it’s not the devil. It’s a kind of user-defined data type, long known to ML and Haskell hackers as sum types, discriminated unions, or algebraic data types. In Rust, they are called *enumerations*, or simply *enums*. Unlike the devil, they are quite safe, and the price they ask is no great privation.

C++ and C# have enums; you can use them to define your own type whose values are a set of named constants. For example, you might define a type named `Color` with values `Red`, `Orange`, `Yellow`, and so on. This kind of enum works in Rust, too. But Rust takes enums much further. A Rust enum can also contain data, even data of varying types. For example, Rust’s `Result<String, io::Error>` type is an enum; such a value is either an `Ok` value containing a `String`, or an `Err` value containing an `io::Error`. This

is beyond what C++ and C# enums can do. It's more like a C `union`—but unlike unions, Rust enums are type-safe.

Enums are useful whenever a value might be either one thing or another. The “price” of using them is that you must access the data safely, using 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 further. 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 it, either by itself:

```
use std::cmp::Ordering;

fn compare(n: i32, m: i32) -> Ordering {
    if n < m {
        Ordering::Less
    } else if n > m {
        Ordering::Greater
    } else {
        Ordering::Equal
    }
}
```

```
    } else {
        Ordering::Equal
    }
}
```

or with all its constructors:

```
use std::cmp::Ordering;
use std::cmp::Ordering::*;

fn compare(n: i32, m: i32) -> Ordering {
    if n < m {
        Less
    } else if n > m {
        Greater
    } else {
        Equal
    }
}
```

After importing the constructors, we can write `Less` instead of `Ordering::Less`, and so on, but because this is less explicit, it's generally considered better style *not* to import them except when it makes your code much more readable.

To import the constructors of an enum declared in the current module, use a `self` import:

```
enum Pet {
    Orca,
    Giraffe,
    ...
}

use self::Pet::*;


```

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 accommodate 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 in-memory representation by adding a `# [repr]` attribute to the enum. For details, see XREF HERE.

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. You can either write your own checked conversion:

```
fn http_status_from_u32(n: u32) -> Option<HttpStatus> {
    match n {
        200 => Some(HttpStatus::Ok),
        304 => Some(HttpStatus::NotModified),
        404 => Some(HttpStatus::NotFound),
        ...
        _ => None,
    }
}
```

or use [the enum_primitive crate](#). It contains a macro that autogenerateds this kind of conversion code for you.

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,
}
```

Enums can have methods, just like structs:

```
impl TimeUnit {
    /// Return the plural noun for this time unit.
    fn plural(self) -> &'static str {
        match self {
            TimeUnit::Seconds => "seconds",
            TimeUnit::Minutes => "minutes",
            TimeUnit::Hours => "hours",
            TimeUnit::Days => "days",
            TimeUnit::Months => "months",
            TimeUnit::Years => "years",
        }
    }

    /// Return the singular noun for this time unit.
    fn singular(self) -> &'static str {
        self.plural().trim_end_matches('s')
    }
}
```

So much for C-style enums. The more interesting sort of Rust enum is the kind that contains data.

Enums with Data

Some programs always need to display full dates and times down to the millisecond, but for most applications, it's more user-friendly to use a rough

approximation, like “two months ago.” We can write an enum to help with that:

```
/// 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, `InThePast` and `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 },
    Cuboid { corner1: Point3d, corner2: 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,  
    },  
}
```

All constructors and fields of a public enum are automatically public.

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 1.17, `RoughTime` fits in 8 bytes, as shown in [Figure 9-1](#).

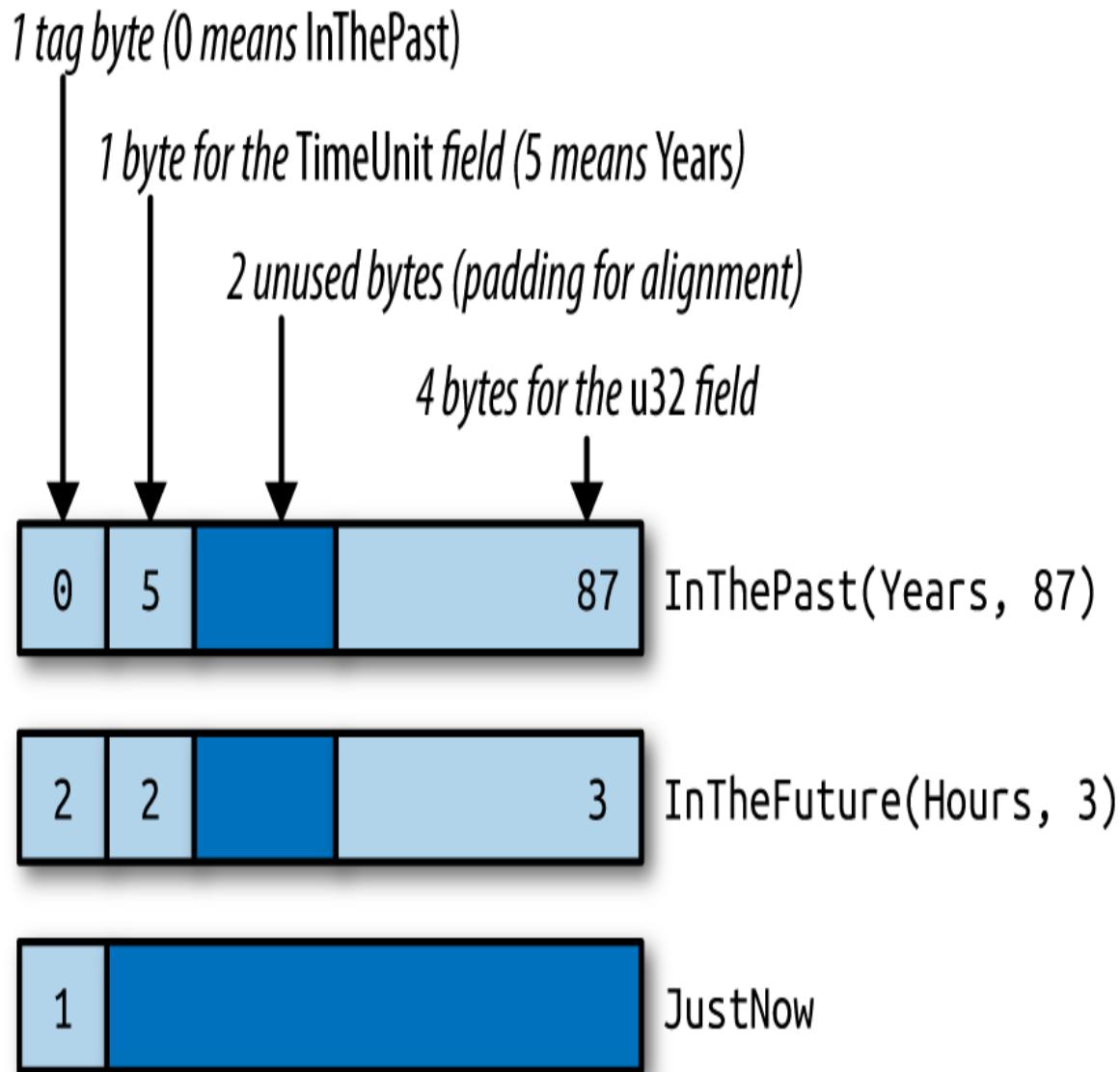


Figure 9-1. RoughTime values in memory

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 the figure 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 9-2](#) 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.

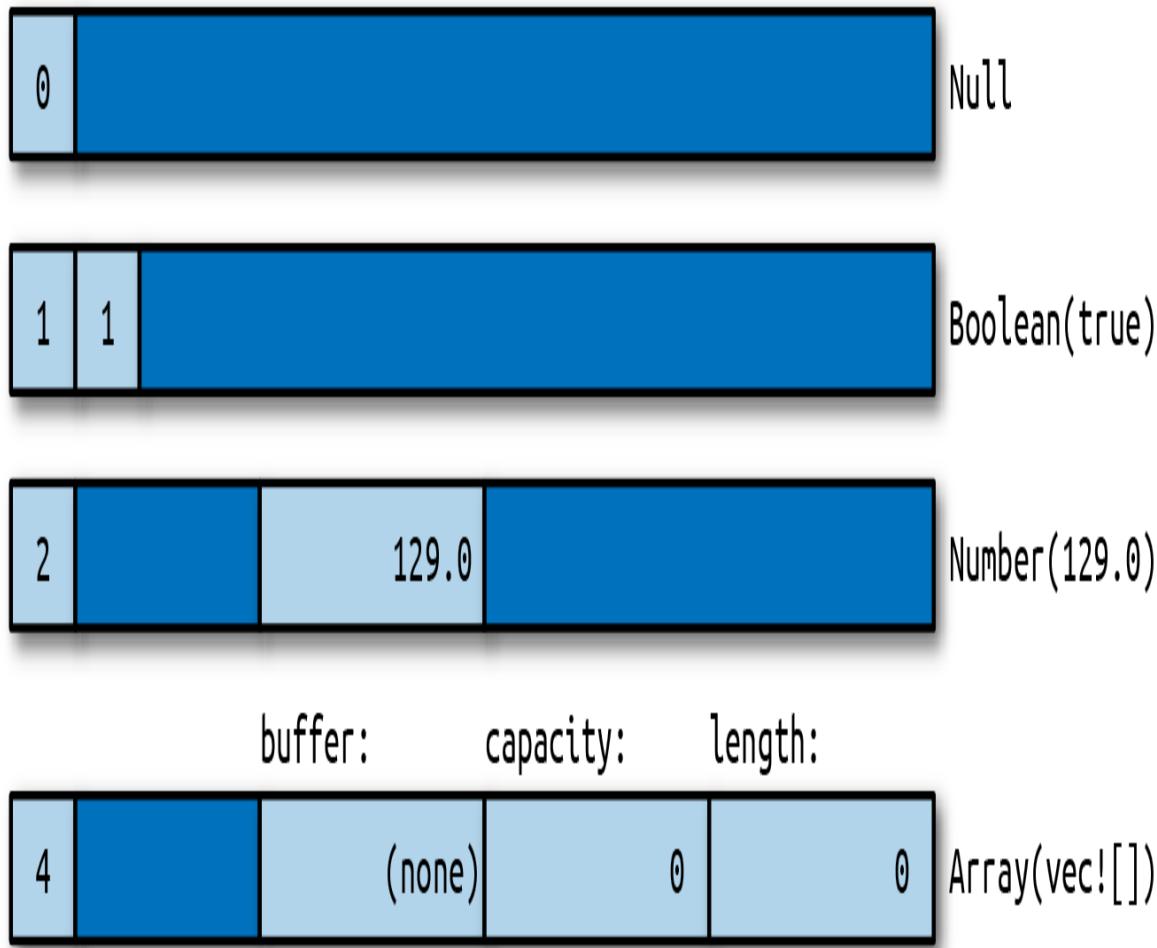


Figure 9-2. Json values in memory

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 eight 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:

```
// An ordered collection of `T`s.  
enum BinaryTree<T> {  
    Empty,  
    NonEmpty(Box<TreeNode<T>>),  
}  
  
// A part of a BinaryTree.  
struct TreeNode<T> {  
    element: T,  
    left: BinaryTree<T>,  
    right: BinaryTree<T>,  
}
```

These few 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 these two definitions, 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 has a `Box`, a pointer to a heap-allocated `TreeNode`.

Each `TreeNode` value contains one actual element, as well as two more `BinaryTree` values. This means a tree can contain subtrees, and thus a `NonEmpty` tree can have any number of descendants.

A sketch of a value of type `BinaryTree<&str>` is shown in [Figure 9-3](#). As with `Option<Box<T>>`, Rust eliminates the tag field, so a `BinaryTree` value

is just one machine word.

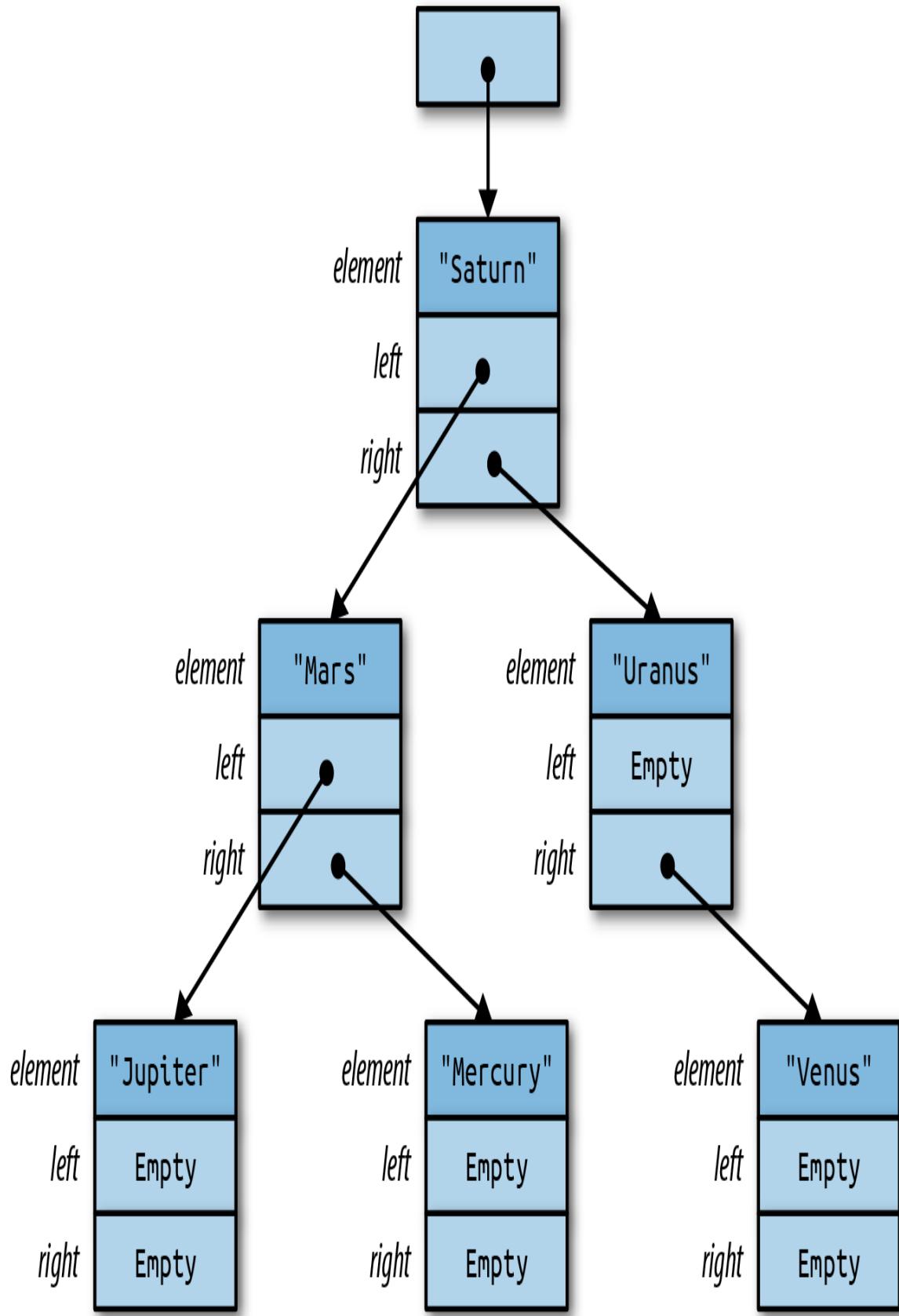


Figure 9-3. A BinaryTree containing six strings

Building any particular node in this tree is straightforward:

```
use self::BinaryTree::*;

let jupiter_tree = NonEmpty(Box::new(TreeNode {
    element: "Jupiter",
    left: Empty,
    right: Empty,
}));
```

Larger trees can be built from smaller ones:

```
let mars_tree = NonEmpty(Box::new(TreeNode {
    element: "Mars",
    left: jupiter_tree,
    right: mercury_tree,
}));
```

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 node is no different from the others:

```
let tree = NonEmpty(Box::new(TreeNode {
    element: "Saturn",
    left: mars_tree,
    right: uranus_tree,
}));
```

Later in this chapter, we'll show how to implement an `add` method on the `BinaryTree` type, so that we can instead write:

```
let mut tree = BinaryTree::Empty;
for planet in planets {
    tree.add(planet);
}
```

No matter what language you’re coming from, creating data structures like `BinaryTree` in Rust will likely take some practice. It won’t be obvious at first where to put the `Boxes`. One way to find a design that will work is to draw a picture like [Figure 9-3](#) that shows how you want things laid out in memory. Then work backward from the picture to the code. Each collection of rectangles is a struct or tuple; each arrow is a `Box` or other smart pointer. Figuring out the type of each field is a bit of a puzzle, but a manageable one. The reward for solving the puzzle is control over your program’s memory usage.

Now comes the “price” we 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 real 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 `radius` on type `Shape`
```

The only way to access the data in an enum is the safe way: using patterns.

Patterns

Recall the definition of our `RoughTime` type from earlier in this chapter:

```
enum RoughTime {
    InThePast(TimeUnit, u32),
    JustNow,
    InTheFuture(TimeUnit, u32),
}
```

Suppose you have a `RoughTime` value and you’d like to display it on a web page. You need to access the `TimeUnit` and `u32` fields inside the value. Rust doesn’t let you access them directly, by writing `rough_time.0` and `rough_time.1`, because after all, the value might be `RoughTime::JustNow`, which has no fields. But then, how can you get the data out?

You need a `match` expression:

```
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 }
```

`match` performs pattern matching; in this example, the *patterns* are the parts that appear before the `=>` symbol on lines 3, 5, and 7. Patterns that match `RoughTime` values look just like the expressions used to create `RoughTime` values. This is no coincidence. Expressions *produce* values; patterns *consume* values. The two use a lot of the same syntax.

Let's step through what happens when this `match` expression runs. Suppose `rt` is the value `RoughTime::InTheFuture(TimeUnit::Months, 1)`. Rust first tries to match this value against the pattern on line 3. As you can see in Figure 9-4, it doesn't match.

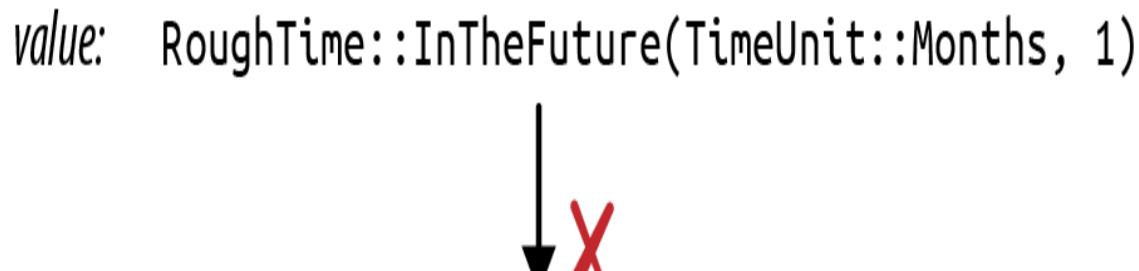


Figure 9-4. A `RoughTime` value and pattern that do not match

Pattern matching 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 ([Figure 9-5](#)).

value: RoughTime::InTheFuture(TimeUnit::Months, 1)



pattern: RoughTime::InTheFuture(units, count)

Figure 9-5. A successful match

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".

That 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 will warn about an “unreachable pattern” if you make this kind of mistake.

Unfortunately, even with the new code, there is still a problem with `RoughTime::InTheFuture(TimeUnit::Hours, 1)`: the result "a hour from now" is not quite right. Such is the English language. This too can be fixed by adding another arm to the `match`.

As this example shows, pattern matching works hand in hand with enums and can even test the data they contain, making `match` a powerful, flexible replacement for C’s `switch` statement.

So far, we’ve only seen patterns that match enum values. There’s more to it than that. Rust patterns are their own little language, summarized in [Table 9-1](#). We’ll spend most of the rest of the chapter on the features shown in this table.

Table 9-1. Patterns

Pattern type	Example	Notes
Literal	<code>100</code>	Matches an exact value; the name of a <code>const</code> is also allowed
	<code>"name"</code>	
Range	<code>0 ..= 100</code>	Matches any value in range, including the end value
	<code>'a' ..= 'k'</code>	
Wildcard	<code>_</code>	Matches any value and ignores it
Variable	<code>name</code>	Like <code>_</code> but moves or copies the value into a new local variable
	<code>mut count</code>	
<code>ref</code> variable	<code>ref field</code>	Borrows a reference to the matched value instead of moving or copying it
	<code>ref mut field</code>	

Pattern type	Example	Notes
Binding with subpattern	<pre>val @ 0 ..= 99</pre>	Matches the pattern to the right of @, using the variable name to the left
	<pre>ref circle @ Shape::C circle { .. }</pre>	
Enum pattern	<pre>Some(value)</pre>	
	<pre>None</pre>	
	<pre>Pet::Orca</pre>	
Tuple pattern	<pre>(key, value)</pre>	
	<pre>(r, g, b)</pre>	
Array pattern	<pre>[a, b, c, d, e, f, g]</pre>	
	<pre>[heading, carom, correction]</pre>	

Pattern type	Example	Notes
Slice pattern	<pre>[first, second]</pre>	Slice length must match pattern.
	<pre>[first, _, third]</pre>	
	<pre>[]</pre>	
Struct pattern	<pre>Color(r, g, b)</pre>	
	<pre>Point { x, y }</pre>	
	<pre>Card { suit: Clubs, rank: n }</pre>	
	<pre>Account { id, name, .. }</pre>	
Reference	<pre>&value</pre>	Matches only reference values
	<pre>&(k, v)</pre>	
Multiple patterns	<pre>'a' 'A'</pre>	In refutable patterns only (<code>match</code> , <code>if</code> <code>let</code> , <code>while let</code>)

Pattern type	Example	Notes
Guard expression	<code>x if x * x <= r2</code>	In <code>match</code> only (not valid in <code>let</code> , etc.)

Literals, Variables, and Wildcards in Patterns

So far, we've shown `match` expressions working with enums. Other types can be matched too. When you need something like a C `switch` statement, use `match` with an integer value. Integer literals like `0` and `1` can serve as patterns:

```
match meadow.count_rabbits() {
    0 => {} // nothing to say
    1 => println!("A rabbit is nosing around in the clover."),
    n => println!("There are {} rabbits hopping about in the meadow", n),
}
```

The pattern `0` matches if there are no rabbits in the meadow. `1` matches if there is just one. If there are two or more rabbits, we reach the third pattern, `n`. This pattern is just a variable name. It can match any value, and the matched value is moved or copied into a new local variable. So in this case, the value of `meadow.count_rabbits()` is stored in a new local variable `n`, which we then print.

Other literals can be used as patterns too, including Booleans, characters, and even strings:

```
let calendar = match settings.get_string("calendar") {
    "gregorian" => Calendar::Gregorian,
    "chinese" => Calendar::Chinese,
    "ethiopian" => Calendar::Ethiopian,
    other => return parse_error("calendar", other),
};
```

In this example, `other` serves as a catch-all pattern, like `n` in the previous example. These patterns play the same role as a `default` case in a `switch` statement, matching values that don't match any of the other patterns.

If you need a catch-all pattern, but you don't care about the matched value, you can use a single underscore `_` as a pattern, the *wildcard pattern*:

```
let caption = match photo.tagged_pet() {
    Pet::Tyrannosaur => "RRRAAAAHHHHH",
    Pet::Samoyed => "*dog thoughts*",
    _ => "I'm cute, love me", // generic caption, works for any pet
};
```

The wildcard pattern matches any value, but without storing it anywhere. Since Rust requires every `match` expression to handle all possible values, a wildcard is often required at the end. Even if you're very sure the remaining cases can't occur, you must at least add a fallback arm that panics:

```
// There are many Shapes, but we only support "selecting"
// either some text, or everything in a rectangular area.
// You can't select an ellipse or trapezoid.
match document.selection() {
    Shape::TextSpan(start, end) => paint_text_selection(start, end),
    Shape::Rectangle(rect) => paint_rect_selection(rect),
    _ => panic!("unexpected selection type"),
}
```

It's worth noting that existing variables can't be used in patterns. Suppose we're implementing a board game with hexagonal spaces, and the player just clicked to move a piece. To confirm that the click was valid, we might try something like this:

```
fn check_move(current_hex: Hex, click: Point) -> game::Result<Hex> {
    match point_to_hex(click) {
        None =>
            Err("That's not a game space."),
        Some(current_hex) => // try to match if user clicked the current_hex
                            // (it doesn't work: see explanation below)
            Err("You are already there! You must click somewhere else."),
        Some(other_hex) =>
            Ok(other_hex)
    }
}
```

This fails because identifiers in patterns introduce *new* variables. The pattern `Some(current_hex)` here creates a new local variable `current_hex`, shadowing the argument `current_hex`. Rust emits several warnings about this code—in particular, the last arm of the `match` is unreachable. To fix it, use an `if` expression:

```
Some(hex) => {
    if hex == current_hex {
        Err("You are already there! You must click somewhere else")
    } else {
        Ok(hex)
    }
}
```

In a few pages, we'll cover **guards**, which offer another way to solve this problem.

Tuple and Struct Patterns

Tuple patterns match tuples. They're useful any time you want 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",
    }
}
```

Struct patterns use curly braces, just like struct expressions. They contain a subpattern for each field:

```
match balloon.location {
```

```

Point { x: 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 stored in the new local variable `height`.

Suppose `balloon.location` is `Point { x: 30, y: 40 }`. As always, Rust checks each component of each pattern in turn [Figure 9-6](#).

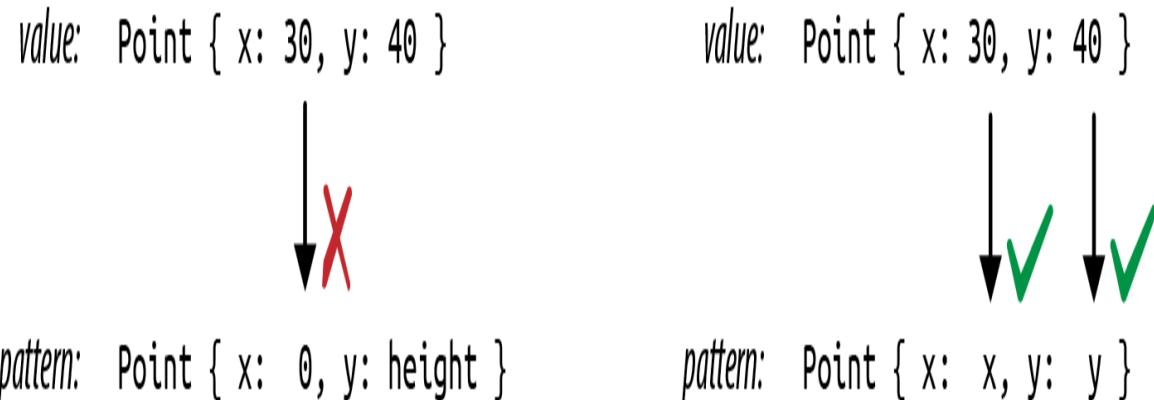


Figure 9-6. Pattern matching with structs

The second arm matches, so the output would be “at (30m, 40m)”.

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),
```

Array and Slice Patterns

Array patterns match arrays. They're often used to filter out some special-case values, and are useful any time you're working with arrays whose values have a different meaning based on position.

For example, when converting hue, saturation, and lightness (HSL) color values to red, green, blue (RGB) color values, colors with zero lightness or full lightness are just black or white. We could use a match expression to deal with those cases simply.

```
fn hsl_to_rgb(hsl: [u8; 3]) -> [u8; 3] {  
    match hsl {  
        [_, _, 0] => [0, 0, 0],  
        [_, _, 255] => [255, 255, 255],  
        ...  
    }  
}
```

Slice patterns are similar, but unlike arrays, slices have variable lengths, so slice patterns match not only on values but also on length. Because slices can be indefinitely long, it's not possible to exhaustively match slices; you have to add a wildcard pattern.

```
fn greet_people(names: &[&str]) {  
    match names {  
        [] => { println!("Hello, nobody.") },  
        [a] => { println!("Hello, {}", a) },
```

```

        [a, b] => { println!("Hello, {} and {}.", a, b) },
        _ => { println!("Hello, everyone.") }
    }
}

```

Reference Patterns

Rust patterns support two features for working with references. `ref` patterns borrow parts of a matched value. `&` patterns match references. We'll cover `ref` patterns first.

Matching a non-copyable value moves the value. Continuing with the account example, this code would be invalid:

```

match account {
    Account { name, language, .. } => {
        ui.greet(&name, &language);
        ui.show_settings(&account); // error: borrow of moved value:
`account`
    }
}

```

Here, the fields `account.name` and `account.language` are moved into local variables `name` and `language`. The rest of `account` is dropped. That's why we can't borrow a reference to it afterward.

If `name` and `language` were both copyable values, Rust would copy the fields instead of moving them, and this code would be fine. But suppose these are `Strings`. What can we do?

We need a kind of pattern that *borrow*s matched values instead of moving them. The `ref` keyword does just that:

```

match account {
    Account { ref name, ref language, .. } => {
        ui.greet(name, language);
        ui.show_settings(&account); // ok
    }
}

```

Now the local variables `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.

You can use `ref mut` to borrow `mut` references:

```
match line_result {
    Err(ref err) => log_error(err), // `err` is &Error (shared ref)
    Ok(ref mut line) => {
        trim_comments(line); // modify the String in place
        handle(line);
    }
}
```

The pattern `Ok(ref mut line)` matches any success result and borrows a `mut` reference to the success value stored inside 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, then `sphere.center()` returns `&Point3d { x: 0.0, y: 0.0, z: 0.0 }`.

So pattern matching proceeds as shown in [Figure 9-7](#).

value: &Point3d { x: 0.0, y: 0.0, z: 0.0}



pattern: &Point3d { x, y, z }

Figure 9-7. Pattern matching with references

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, `&` creates a reference. In a pattern, `&` matches a reference.

Matching a reference follows all the rules we've come to expect. Lifetimes are enforced. You can't get `mut` access via a shared reference. And you can't move a value out of a reference, even a `mut` reference. When we match `&Point3d { x, y, z }`, the variables `x`, `y`, and `z` receive copies of the coordinates, leaving the original `Point3d` value intact. It works because those fields are copyable. If we try the same thing on a struct with non-copyable fields, we'll get an error:

```
match friend.borrow_car() {
    Some(&Car { engine, .. }) => // error: can't move out of borrow
    ...
    None => {}
}
```

Scrapping a borrowed car for parts is not nice, and Rust won't stand for it. You can use a `ref` pattern to borrow a reference to a part. You just don't own it.

```
Some(&Car { ref engine, .. }) => // ok, engine is 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 XREF HERE.)

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(),
}
```

Ranges in patterns are *inclusive*, so that both '`0`' and '`9`' match the pattern `'0' ..= '9'`. By contrast, range expressions (written with two dots, as in `for n in 0..100`) are half-open, or *exclusive* (covering `0` but not `100`). The reason for the inconsistency is simply that exclusive ranges are more useful for loops and slicing, but inclusive ranges are more useful in pattern matching.

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

Finally, `x @ pattern` matches exactly like the given `pattern`, but on success, instead of creating variables for parts of the matched value, it creates a single variable `x` and moves or copies the whole value into it. For example, say you have this code:

```
match self.get_selection() {
    Shape::Rect(top_left, bottom_right) => {
        optimized_paint(&Shape::Rect(top_left, bottom_right))
    }
    other_shape => {
        paint_outline(other_shape.get_outline())
    }
}
```

```
    }
}
```

Note that the first case unpacks a `Shape::Rect` value, only to rebuild an identical `Shape::Rect` value on the next line. This can be rewritten to use an @ pattern:

```
rect @ Shape::Rect(..) => {
    optimized_paint(&rect)
}
```

@ patterns are also useful with ranges:

```
match chars.next() {
    Some(digit @ '0'..='9') => read_number(digit, chars),
    ...
},
```

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 Track { album, track_number, title, .. } = song;

// ...unpack a function argument that's a tuple
fn distance_to((x, y): (f64, f64)) -> f64 { ... }

// ...iterate over keys and values of a HashMap
for (id, document) in &cache_map {
    println!("Document #{}: {}", id, document.title);
}

// ...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. The same concept exists in some other languages: in JavaScript, it's called *destructuring*; in Python, *unpacking*.

Note that in all four examples, we use patterns that are guaranteed to match. The pattern `Point3d { x, y, z }` matches every possible value of the `Point3d` struct type; `(x, y)` matches any `(f64, f64)` pair; and so on. Patterns that always match are special in Rust. They're called *irrefutable patterns*, and they're the only patterns allowed in the four places shown here (after `let`, in function arguments, after `for`, and in closure arguments).

A *refutable pattern* is one that might not match, like `Ok(x)`, which doesn't match an error result, or `'0' ..= '9'`, which doesn't match the character '`'Q'`'. Refutable patterns can be used in `match` arms, because `match` is designed for them: if one pattern fails to match, it's clear what happens next. The four examples above are places in Rust programs where a pattern can be handy, but the language doesn't allow for match failure.

Refutable patterns are also allowed in `if let` and `while let` expressions, which can be used to...

```
// ...handle just one enum variant specially
if let RoughTime::InTheFuture(_, _) = user.date_of_birth() {
    user.set_time_traveler(true);
}

// ...run some code only if a table lookup succeeds
if let Some(document) = cache_map.get(&id) {
    return send_cached_response(document);
}

// ...repeatedly try something until it succeeds
while let Err(err) = present_cheesy_anti_robot_task() {
    log_robot_attempt(err);
    // let the user try again (it might still be a human)
}
```

```
// ...manually loop over an iterator
while let Some(_) = lines.peek() {
    read_paragraph(&mut lines);
}
```

For details about these expressions, see “[if let](#)” and “[Loops](#)”.

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:

```
// An ordered collection of `T`s.
enum BinaryTree<T> {
    Empty,
    NonEmpty(Box<TreeNode<T>>),
}

// A part of a BinaryTree.
struct TreeNode<T> {
    element: T,
    left: BinaryTree<T>,
    right: BinaryTree<T>,
}
```

You 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             BinaryTree::Empty => {
5                 *self = BinaryTree::NonEmpty(Box::new(TreeNode {
6                     element: value,
7                     left: BinaryTree::Empty,
8                     right: BinaryTree::Empty,
9                     })),
10                }
11             BinaryTree::NonEmpty(ref mut node) => {
```

```

12         if value <= node.element {
13             node.left.add(value);
14         } else {
15             node.right.add(value);
16         }
17     }
18 }
19 }
20 }
```

Line 1 tells Rust that we’re defining a method on `BinaryTrees` of ordered types. This is exactly the same syntax we use to define methods on generic structs, explained in “[Defining Methods with `impl`](#)”.

If the existing tree `*self` is empty, that’s the easy case. Lines 5–9 run, changing the `Empty` tree to a `NonEmpty` one. The call to `Box::new()` here allocates a new `TreeNode` in the heap. When we’re done, the tree contains one element. Its left and right subtrees are both `Empty`.

If `*self` is not empty, we match the pattern on line 11:

```
BinaryTree::NonEmpty(ref mut node) => {
```

This pattern borrows a mutable reference to the `Box<TreeNode<T>>`, so we can access and modify data in that tree node. That reference is named `node`, and it’s in scope from line 12 to line 16. Since there’s already an element in this node, the code must recursively call `.add()` to add the new element to either the left or the right subtree.

The new method can be used like this:

```

let mut tree = BinaryTree::Empty;
tree.add("Mercury");
tree.add("Venus");
...
}
```

The Big Picture

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've 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 be added only 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

A NOTE FOR EARLY RELEASE READERS

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

This will be the 11th chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at jimb@red-bean.com.

[A] computer scientist tends to be able to deal with nonuniform structures—case 1, case 2, case 3—while a mathematician will tend to want one unifying axiom that governs an entire system.

—Donald Knuth

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*. Here are 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 `Files` and `TcpStreams`. 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. These concepts will be familiar to many programmers, but Rust takes a fresh approach inspired by Haskell's typeclasses.

Traits are Rust's take on interfaces or abstract base classes. At first, they look just like interfaces in Java or C#. 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<()>;

    fn write_all(&mut self, buf: &[u8]) -> Result<()> { ... }
    ...
}
```

This trait offers several methods; we've shown only the first three.

The standard types `File` and `TcpStream` both implement `std::io::Write`. So does `Vec<u8>`. All three types provide methods named `.write()`, `.flush()`, and so on. Code that uses a writer without caring about its type looks like this:

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

fn say_hello(out: &mut dyn Write) -> std::io::Result<()> {
    out.write_all(b"hello world\n")?;
    out.flush()
}
```

The type of `out` is `&mut dyn Write`, meaning “a mutable reference to any value that implements the `Write` trait.”

```

use std::fs::File;
let mut local_file = File::create("hello.txt")?;
say_hello(&mut local_file)?; // works

let mut bytes = vec![];
say_hello(&mut bytes)?; // also works
assert_eq!(bytes, b"hello world\n");

```

This chapter begins by showing how traits are used, how they work, and how to define your own. But there is more to traits than we've hinted at so far. We'll use them to add extension methods to existing types, even built-in types like `str` and `bool`. We'll explain why adding a trait to a type costs no extra memory and how to use traits without virtual method call overhead. We'll see that built-in traits are the hook into the language that Rust provides for operator overloading and other features. And we'll cover the `Self` type, associated methods, and associated types, three features Rust lifted from Haskell that elegantly solve problems that other languages address with workarounds and hacks.

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 dyn 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 any given type may or may not support. 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 produce a sequence of values.
- A value that implements `std::clone::Clone` can make clones of itself in memory.
- A value that implements `std::fmt::Debug` can be printed using `println!()` with 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![];
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![];
buf.write_all(b"hello")?; // ok
```

Rust has this rule because, as we'll see later in this chapter, you can use traits to add new methods to any type—even standard library types like `u32` and `str`. Third-party crates can do the same thing. Clearly, this could lead to naming conflicts! But since Rust makes you import the traits you plan to use, crates are free to take advantage of this superpower. To get a conflict, you'd have to import two traits that add a method with the same name to the same type. It's not something that happens in practice.

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 XREF HERE.

C++ and C# programmers will already have noticed that trait methods are like virtual methods. 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++ and C# will often do the same, although the possibility of

subclassing sometimes precludes this.) Only calls through `&mut dyn Write` incur the overhead of a dynamic dispatch, also known as a virtual method call, which is indicated by the `dyn` keyword in the type.

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.

Rust doesn't permit variables of type `dyn Write`:

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

let mut buf: Vec<u8> = vec![];
let writer: dyn Write = buf; // error: `Write` does not have a constant size
```

A variable's size has to be known at compile time, and types that implement `Write` can be any size.

This may be surprising if you're coming from C# or Java, but the reason is simple. In Java, a variable of type `OutputStream` (the Java standard interface analogous to `std::io::Write`) is a reference to any object that implements `OutputStream`. The fact that it's a reference goes without saying. It's the same with interfaces in C# and most other languages.

What we want in Rust is the same thing, but in Rust, references are explicit:

```
let mut buf: Vec<u8> = vec![];
let writer: &mut dyn Write = &mut buf; // ok
```

A reference to a trait type, like `writer`, is called a *trait object*. Like any other reference, a trait object points to some value, it has a lifetime, and it can be either `mut` or shared.

What makes a trait object different is that Rust usually doesn't know the type of the referent at compile time. So a trait object includes a little extra

information about the referent's type. This is strictly for Rust's own use behind the scenes: when you call `writer.write(data)`, Rust needs the type information to dynamically call the right `write` method depending on the type of `*writer`. You can't query the type information directly, and Rust does not support downcasting from the trait object `&mut dyn Write` back to a concrete type like `Vec<u8>`.

Trait Object Layout

In memory, a 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 10-1](#).

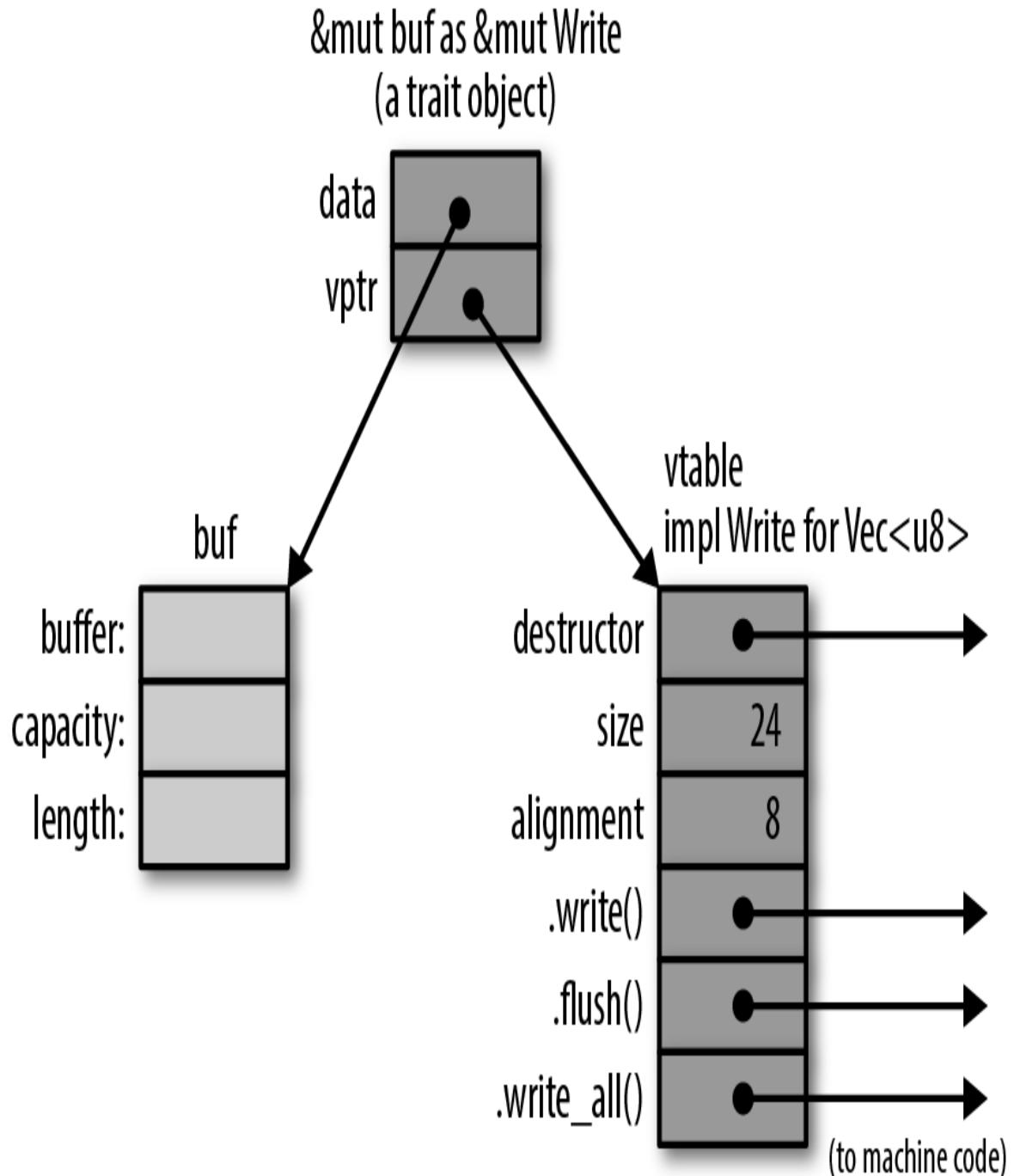


Figure 10-1. Trait objects in memory

C++ has this kind of run-time type information as well. 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 10-1**, including the vtable, is a private implementation detail of Rust. Again, 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, to determine which implementation to call.

Seasoned C++ programmers will notice that Rust and C++ use memory a bit differently. In C++, the vtable pointer, or *vptr*, is stored as part of the struct. Rust uses fat pointers instead. The struct itself contains nothing but its fields. This way, a struct can implement dozens of traits without containing dozens of vptrs. Even types like `i32`, which aren't big enough to accommodate a vptr, can implement traits.

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 = File::create("hello.txt")?;
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 dyn 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<dyn Write>`, a value that owns a writer in the heap:

```
let w: Box<dyn Write> = Box::new(local_file);
```

`Box<dyn Write>`, like `&mut dyn Write`, is a fat pointer: it contains the address of the writer itself and the address of the vtable. The same goes for other pointer types, like `Rc<dyn Write>`.

This kind of conversion is the only way to create a trait object. What the computer is actually doing here is very simple. At the point where the conversion happens, Rust knows the referent's true type (in this case, `File`), so it just adds the address of the appropriate vtable, turning the regular pointer into a fat pointer.

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<()> {
    out.write_all(b"hello world\n")?;
    out.flush()
}
```

Only the type signature has changed:

```
fn say_hello(out: &mut dyn 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 some type that implements the `Write` trait. Type parameters are usually single uppercase letters, by convention.

Which type `W` stands for depends on how the generic function is used:

```
say_hello(&mut local_file)?; // calls say_hello::<File>
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:

```
say_hello::<File>(&mut local_file)?;
```

but it's seldom necessary, because Rust can usually deduce the type parameters by looking at the arguments. Here, the `say_hello` generic function expects a `&mut W` argument, and we're passing it a `&mut File`, so Rust infers that `W = File`.

If the generic function you're calling doesn't have any arguments that provide useful clues, you may have to spell it out:

```
// calling a generic method collect<C>() that takes no arguments
let v1 = (0 .. 1000).collect(); // error: can't infer type
let v2 = (0 .. 1000).collect::<Vec<i32>>(); // ok
```

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 to 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, as shown in [Figure 10-2](#).

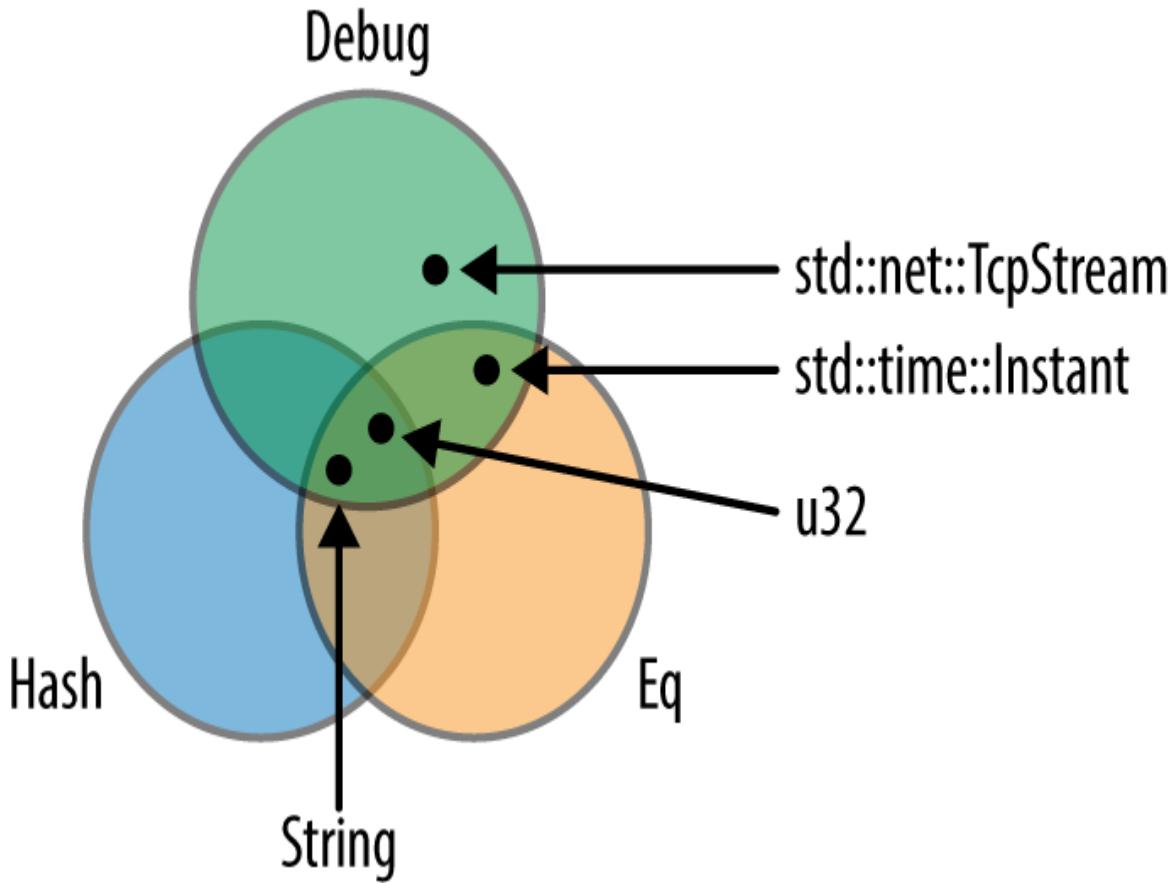


Figure 10-2. Traits as sets of types

It's also possible for a type parameter to have no bounds at all, but 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 into a box or vector. That's about it.

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](#)” introduced the syntax for lifetime parameters. A generic function can have both lifetime parameters and type parameters. Lifetime parameters come first.

```
/// Return a reference to the point in `candidates` that's  
/// closest to the `target` point.  
fn nearest<'t, 'c, P>(target: &'t P, candidates: &'c [P]) -> &'c P  
    where P: MeasureDistance  
{  
    ...  
}
```

This function takes two arguments, `target` and `candidates`. Both are references, and we give them distinct lifetimes `'t` and `'c` (as discussed in “[Distinct Lifetime Parameters](#)”). Furthermore, the function works with any type `P` that implements the `MeasureDistance` trait, so we might use it on `Point2d` values in one program and `Point3d` values in another.

Lifetimes never have any impact on machine code. Two calls to `nearest()` using the same type `P`, but different lifetimes, will call the same compiled function. Only differing types cause Rust to compile 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 “[Generic Structs](#)” and “[Generic Enums](#)”.

- 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<()> {
        goop.pour(&self);
        self.absorb_topping(goop)
    }
}
```

- 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 right choice whenever you need a collection of values of mixed types, all together. It is technically possible to make generic salad:

```
trait Vegetable {
    ...
}

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 \$14 for a `Salad<IcebergLettuce>` and has never quite gotten over the experience.

How can we build a better salad? Since `Vegetable` values can be all different sizes, we can't ask Rust for a `Vec<Vegetable>`:

```
struct Salad {
    veggies: Vec<Vegetable> // error: `Vegetable` does not have
                           //           a constant size
}
```

Trait objects are the solution:

```
struct Salad {
    veggies: Vec<Box<dyn Vegetable>>
}
```

Each `Box<dyn Vegetable>` can own any type of vegetable, but the box itself has a constant size—two pointers—suitable for storing in a vector. 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. Note the absence of the `dyn` keyword in generic function signatures. Because you specify the types at compile time, either explicitly or through type inference, the compiler knows exactly which `write` method to call. The `dyn` keyword isn't used because there are no trait objects - and thus no dynamic dispatch - involved.

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, a call to `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();
say_hello(&mut sink);
```

`std::io::sink()` returns a writer of type `Sink` that 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 the following:

- `Sink::write_all()` does nothing.
- `Sink::flush()` does nothing.
- 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, such as static methods, that work only with generics: they rule out trait objects entirely. We'll point out these features as we come to them.

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 {  
    /// Render this object on the given canvas.  
    fn draw(&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 draw(&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::draw()`, we would have to define it in a separate `impl` block:

```
impl Broom {
    /// Helper function used by Broom::draw() below.
    fn broomstick_range(&self) -> Range<i32> {
        self.y - self.height - 1 .. self.y
    }
}
```

These helper functions can be used within the trait `impl` blocks:

```
impl Visible for Broom {
    fn draw(&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> {
        // Claim to have successfully written the whole buffer.
        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:

```
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 += self.write(&buf[bytes_written..])?;
        }
        Ok(())
    }

    ...
}

```

The `write` and `flush` methods are the basic methods that 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. XREF HERE 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 sole purpose of this particular trait is to add a method to an existing type, `char`. This is called an *extension trait*. Of course, you can add this trait to types, too, by writing `impl IsEmoji for str { ... }` and so forth.

You can even use a generic `impl` block to add an extension trait to a whole family of types at once. This trait could be implemented on any type:

```
use std::io::{self, Write};

/// Trait for values to which you can send HTML.
```

```
trait WriteHtml {
    fn write_html(&mut self, html: &HtmlDocument) -> io::Result<()>;
}
```

Implementing the trait for all writers make it an extension trait, adding a method to all Rust writers:

```
/// You can write HTML to any std::io writer.
impl<W: Write> WriteHtml for W {
    fn write_html(&mut self, html: &HtmlDocument) -> io::Result<()> {
        ...
    }
}
```

The line `impl<W: Write> WriteHtml for W` means “for every type `W` that implements `Write`, here’s an implementation of `WriteHtml` for `W`. ”

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 upshot of all this is that `serde` adds a `.serialize()` method to all these types. It can be used like this:

```
use serde::Serialize;
use serde_json;

pub fn save_configuration(config: &HashMap<String, String>)
    -> std::io::Result<()
{
    // Create a JSON serializer to write the data to a file.
    let writer = File::create(config_filename())?;
    let mut serializer = serde_json::Serializer::new(writer);
```

```
// The serde `serialize()` method does the rest.  
config.serialize(&mut serializer)?;  
  
Ok(())  
}
```

We said earlier that when you implement a trait, either the trait or the type must be new in the current crate. This is called the *coherence rule*. It 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 of `Write` for `u8`, 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 `dyn 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 {
    fn splice(&self, other: &Self) -> Self {
        ...
    }
}

impl Spliceable for Mammoth {
    fn splice(&self, other: &Self) -> Self {
        ...
    }
}
```

then inside the first `impl`, `Self` is simply an alias for `CherryTree`, and in the second, it's an alias for `Mammoth`. 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: &dyn Spliceable, right: &dyn 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 at compile time 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 abstract base classes 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: &dyn MegaSpliceable) -> Box;  
}
```

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`.

Subtraits are like subinterfaces in Java or C#. They're a way to describe a trait that extends an existing trait with a few more methods. In this example, all your code that works with `Creatures` can also use the methods from the `Visible` trait.

Static Methods

In most object-oriented languages, interfaces can't include static methods or constructors, but 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 these four associated functions. The first two, `new()` and `from_slice()`, don't take a `self` argument. They serve as constructors.

In non-generic code, these functions can be called using `::` syntax, just like any other 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()` shown here:

```
// 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
}
```

Like Java and C# interfaces, trait objects don't support static methods. If you want to use `&dyn StringSet` trait objects, you must change the trait, adding the bound `where Self: Sized` to each static method:

```
trait StringSet {
    fn new() -> Self
        where Self: Sized;

    fn from_slice(strings: &[&str]) -> Self
        where Self: Sized;

    fn contains(&self, string: &str) -> bool;

    fn add(&mut self, string: &str);
}
```

This bound tells Rust that trait objects are excused from supporting this method. `StringSet` trait objects are then allowed; they still don't support the two static methods, but you can create them and use them to call `.contains()` and `.add()`. The same trick works for any other method that is incompatible with trait objects. (We will forgo the rather tedious technical explanation of why this works, but the `Sized` trait is covered in XREF HERE.)

Fully Qualified Method Calls

A method is just a special kind of function. These two calls are equivalent:

```
"hello".to_string()  
str::to_string("hello")
```

The second form looks exactly like a static method call. This works even though the `to_string` method takes a `self` argument. Simply pass `self` as the function's first argument.

Since `to_string` is a method of the standard `ToString` trait, there are two more forms you can use:

```
ToString::to_string("hello")  
<str as ToString>::to_string("hello")
```

All four of these method calls do exactly the same thing. Most often, you'll just write `value.method()`. The other forms are *qualified* method calls.

They specify the type or trait that a method is associated with. The last form, with the angle brackets, specifies both: a *fully qualified* method call.

When you write `"hello".to_string()`, using the `.` operator, you don't say exactly which `to_string` method you're calling. Rust has a method lookup algorithm that figures this out, depending on the types, deref coercions, and so on. With fully qualified calls, you can say exactly which method you mean, and that can help in a few odd cases:

- When two methods have the same name. The classic hokey example is the `Outlaw` with two `.draw()` methods from two different traits, one for drawing it on the screen and one for interacting with the law:

```
outlaw.draw(); // error: draw on screen or draw pistol?
```

```
Visible::draw(&outlaw); // ok: draw on screen
```

```
HasPistol::draw(&outlaw); // ok: corral
```

Normally you're better off just renaming one of the methods, but sometimes you can't.

- When the type of the `self` argument can't be inferred:

```
let zero = 0; // type unspecified; could be `i8`, `u8`, ...
```

```
zero.abs(); // error: method `abs` not found
i64::abs(zero); // ok
```

- When using the function itself as a function value:

```
let words: Vec<String> =
    line.split_whitespace() // iterator produces &str values
        .map(<str as ToString>::to_string) // ok
        .collect();
```

Here the fully qualified `<str as ToString>::to_string` is just a way to name the specific function we want to pass to `.map()`.

- When calling trait methods in macros. We'll explain in XREF HERE.

Fully qualified syntax also works for static methods. In the previous section, we wrote `S::new()` to create a new set in a generic function. We could also have written `StringSet::new()` or `<S as StringSet>::new()`.

Traits That Define Relationships Between Types

So far, every trait we've looked at stands alone: a trait is a set of methods that types can implement. Traits can also be used in situations where there are multiple types that have to work together. They can describe relationships between types.

- 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. In the expression `a * b`, the values `a` and `b` can be either the same type, or different types.
- 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`). The traits themselves define exactly how these types work together.

You won't need to create traits like these every day, but you'll come across them throughout the standard library and in third-party crates. In this section, we'll show how each of these examples 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 the `Self` type show up explicitly in the code everywhere their fields, methods, and so on are used.

Here's what it looks like to implement `Iterator` for a type:

```
// (code from the std::env standard library module)
impl Iterator for Args {
    type Item = String;

    fn next(&mut self) -> Option<String> {
        ...
    }
    ...
}
```

`std::env::Args` is the type of iterator returned by the standard library function `std::env::args()` that we used in XREF HERE to access command-line arguments. It produces `String` values, so the `impl` declares `type Item = String;`.

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 preceding example is not code that you would write out yourself, because after reading XREF HERE, 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>(iter: I)
    where I: Iterator
{
    for (index, value) in iter.enumerate() {
        println!("{}: {:?}", index, value); // error
    }
}

```

This almost works. There is just one problem: `value` might not be a printable type.

```

error[E0277]: the trait bound `<I as std::iter::Iterator>::Item:
            std::fmt::Debug` is not satisfied
--> traits_dump.rs:10:37
|
10 |     println!("{}: {:?}", index, value); // error
|             ^^^^^^ the trait `std::fmt::Debug`
|                     is not implemented for
|                     `<I as std::iter::Iterator>::Item`
|
= help: consider adding a
      `where <I as std::iter::Iterator>::Item: std::fmt::Debug` bound
= note: required by `std::fmt::Debug::fmt`

```

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 `Debug` trait, the trait for formatting values with `{:?}`. We can do this by placing a bound on `I::Item`:

```
use std::fmt::Debug;

fn dump<I>(iter: I)
    where I: Iterator, I::Item: Debug
{
    ...
}
```

Or, we could write, “I must be an iterator over `String` values”:

```
fn dump<I>(iter: I)
    where I: Iterator<Item=String>
{
    ...
}
```

`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 `Strings`. This syntax can be used anywhere the name of a trait can be used, including trait object types:

```
fn dump(iter: &mut dyn Iterator<Item=String>) {
    for (index, s) in iter.enumerate() {
        println!("{}: {:?}", index, s);
    }
}
```

Traits with associated types, like `Iterator`, are compatible with trait methods, but only if all the associated types are spelled out, as shown here. 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 `Pattern` 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> {
    /// 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*.

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 and `Vec<i32>` and `Vec<String>` are different types.

A single type—say, `WindowSize`—can implement both `Mul<f64>` and `Mul<i32>`, and many more. You would then be able to multiply a `WindowSize` by many other types. Each implementation would have its own associated `Output` type.

Generic traits get a special dispensation when it comes to the coherence rule: you can implement a foreign trait for a foreign type, so long as one of the trait’s type parameters is a type defined in the current crate. So, if you’ve defined `WindowSize` yourself, you can implement `Mul<WindowSize>` for `f64`, even though you didn’t define either `Mul` or `f64`. These implementations can even be generic, such as `impl<T> Mul<WindowSize> for Vec<T>`. This works because there’s no way any other crate could define `Mul<WindowSize>` on anything, and thus no way a conflict among implementations could arise. (We introduced the coherence rule back in “[Traits and Other People’s Types](#)”.) This is how crates like `nalgebra` define arithmetic operations on vectors.

The trait shown above is missing one minor detail. The real `Mul` trait looks like this:

```
pub trait Mul<RHS=Self> {  
    ...  
}
```

The syntax `RHS=Self` means that `RHS` defaults to `Self`. If I write `impl Mul for Complex`, without specifying `Mul`’s type parameter, it means `impl Mul<Complex> for Complex`. In a bound, if I write `where T: Mul`, it means `where T: Mul<T>`.

In Rust, the expression `lhs * rhs` is shorthand for `Mul::mul(lhs, rhs)`. So overloading the `*` operator in Rust is as simple as implementing the `Mul` trait. We’ll show examples in the next chapter.

impl Trait

As you might imagine, combinations of many generic types can get messy. For example, combining just a few iterators using standard library combinators rapidly turns your return type into an eyesore.

```
fn cyclical_zip(v: Vec<u8>, u: Vec<u8>) ->  
    iter::Cycle<iter::Chain<IntoIter<u8>, IntoIter<u8>>> {
```

```
    v.into_iter().chain(u.into_iter()).cycle()
}
```

We could easily replace this hairy return type with a trait object:

```
fn cyclical_zip(v: Vec<u8>, u: Vec<u8>) -> Box<dyn Iterator<Item=u8>> {
    Box::new(v.into_iter().chain(u.into_iter()).cycle())
}
```

However, taking the overhead of dynamic dispatch and an unavoidable heap allocation every time this function is called just to avoid an ugly type signature doesn't seem like a good trade, in most cases.

Rust has a feature called `impl Trait` designed for precisely this situation. `impl Trait` allows us to “erase” the type of a return value, specifying only the trait or traits it implements, without dynamic dispatch or a heap allocation.

```
fn cyclical_zip(v: Vec<u8>, u: Vec<u8>) -> impl Iterator<Item=u8> {
    v.into_iter().chain(u.into_iter()).cycle()
}
```

Now, rather than specifying a particular nested type of iterator combinator structs, `cyclical_zip`'s signature just states that it returns some kind of iterator over `u8`. The return type expresses the intent of the function, rather than its implementation details.

This has definitely cleaned up the code and made it more readable, but `impl Trait` is more than just a convenient shorthand. Using `impl Trait` means that you can change the actual type being returned in the future as long as it still implements `Iterator<Item=u8>`, and any code calling the function will continue to compile without an issue. This provides a lot of flexibility for library authors, because only the relevant functionality is encoded in the type signature.

For example, if the first version of a library uses iterator combinators as above, but a better algorithm for the same process is discovered, the library

author can use different combinators or even make a custom type that implements `Iterator` and users of the library can get the performance improvements without changing their code at all.

It might be tempting to use `impl Trait` to approximate a statically-dispatched version of the factory pattern that's commonly used in object oriented languages. For example, you might define a trait like this:

```
trait Shape {
    fn new() -> Self;
    fn area(&self) -> f64;
}
```

After implementing it for a few types, you might want to different `Shapes` depending on a runtime value, like a string that a user enters. This doesn't work with `impl Shape` as the return type:

```
fn make_shape(shape: &str) -> impl Shape {
    match shape {
        "circle" => Circle::new(),
        "triangle" => Triangle::new(), // error: incompatible types
        "shape" => Rectangle::new(),
    }
}
```

From the perspective of the caller, a function like this doesn't make much sense. `impl Trait` is a form of static dispatch, so the compiler has to know the type being returned from the function at compile time in order to allocate the right amount of space on the stack and correctly access fields and methods on that type. Here, it could be `Circle`, `Triangle`, or `Rectangle`, which could all take up different amounts of space and all have different implementations of `area()`.

It's important to note that Rust doesn't allow trait methods to use `impl Trait` return values. Supporting this will require some improvements in the language's type system. Until that work is done, only free functions and functions associated with specific types can use `impl Trait` returns.

`impl Trait` can also be used in functions that take generic arguments. For instance, this simple generic function:

```
fn print<T: Display>(val: T) {
    println!("{}", val);
}
```

is identical to this version using `impl Trait`:

```
fn print(val: impl Display) {
    println!("{}", val);
}
```

with one important exception. Using generics allows callers of the function to specify the type of the generic arguments, like `print::<i32>(42)`, while using `impl Trait` does not.

Each `impl Trait` argument is assigned its own anonymous type parameter, so `impl Trait` for arguments is limited to only the simplest generic functions, with no relationships between the types of arguments.

Associated Consts

Like structs and enums, traits can have associated constants. You can declare a trait with an associated constant using the same syntax as for a struct or enum:

```
trait Greet {
    const GREETING: &'static str = "Hello";
    fn greet(&self) -> String;
}
```

Associated consts in traits have a special power, though. Like associated types and functions, you can declare them but not give them a value.

```
trait Float {
    const ZERO: Self;
    const ONE: Self;
}
```

Then, implementors of the trait can define these values.

```
impl Float for f32 {
    const ZERO: f32 = 0.0;
    const ONE: f32 = 1.0;
}

impl Float for f64 {
    const ZERO: f64 = 0.0;
    const ONE: f64 = 1.0;
}
```

This allows you to write generic code that uses these values, but associated constants can't be used with trait objects, since the compiler relies on type information about the implementation in order to pick the right value at compile time.

```
fn add_one<T: Float + Add<Output=T>>(value: T) -> T {
    value + T::ONE
}
```

Even a simple trait with no behavior at all, like `Float`, can give enough information about a type, in combination with a few operators, to implement common mathematical functions like Fibonacci.

```
fn fib<T: Float + Add<Output=T>>(n: usize) -> T {
    match n {
        0 => T::ZERO,
        1 => T::ONE,
        n => fib::<T>(n - 1) + fib::<T>(n - 2)
    }
}
```

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 `rand` library provides a few different implementations, including `XorShiftRng` (a fast pseudorandom number generator) and `OsRng` (much slower, but truly unpredictable, for use in cryptography).

The buddy 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;
}
```

Types like `f64` and `bool` implement this trait. Pass any random number generator to their `::rand()` method, and it returns a random value:

```
let x = f64::rand(rng);
let b = bool::rand(rng);
```

In fact `random()` is nothing but a thin wrapper that passes a globally allocated `Rng` to this `rand` method. One way to implement it is like this:

```
pub fn random<T: Rand>() -> T {
    T::rand(&mut global_rng())
}
```

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, as shown in [Figure 10-3](#).

types that implement Rng

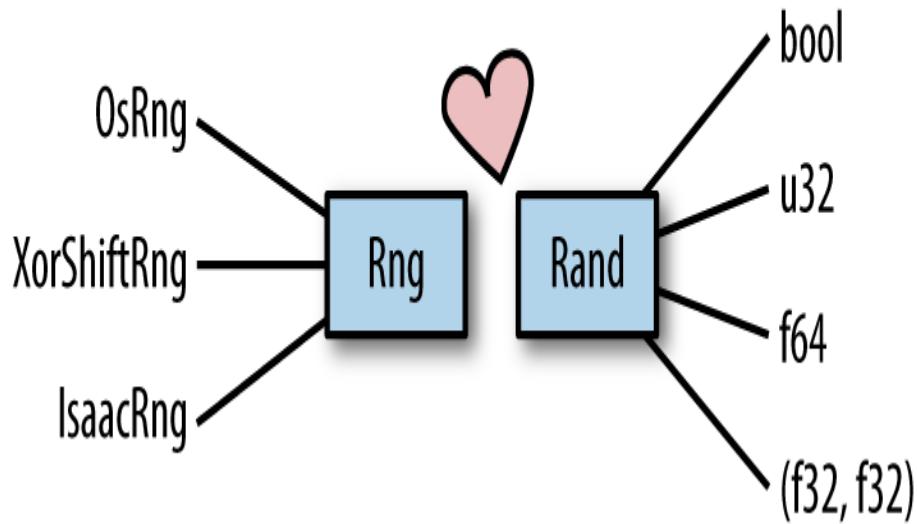


Figure 10-3. Buddy traits illustrated. The `Rng` types listed on the left are real random number generators provided by the `rand` crate.

The standard library’s support for computing hash codes provides another example of buddy 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::hash()` that accepts any type of `Hasher` as an argument.

Another example is the `serde` library’s `Serialize` trait, which you saw in “[Traits and Other People’s Types](#)”. It has a buddy trait we didn’t talk about: the `Serializer` trait, which represents the output format. `serde` supports pluggable serialization formats. There are `Serializer` implementations for JSON, YAML, a binary format called CBOR, and so on. Thanks to the close relationship between the two traits, every format automatically supports every serializable type.

In the last three sections, we’ve shown three 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.

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 do some computation:

```
fn dot(v1: &[i64], v2: &[i64]) -> i64 {
    let mut total = 0;
    for i in 0 .. v1.len() {
        total = total + v1[i] * v2[i];
    }
    total
}
```

Now we want to use the same code with floating-point values. We might try something like this:

```
fn dot<N>(v1: &[N], v2: &[N]) -> N {
    let mut total: N = 0;
    for i in 0 .. v1.len() {
        total = total + v1[i] * v2[i];
    }
    total
}
```

No such luck: Rust complains about the use of `+` and `*` and the type of `0`. We can require `N` to be a type that supports `+` and `*` using the `Add` and `Mul` traits. Our use of `0` needs to change, though, because `0` is always an integer in Rust; the corresponding floating-point value is `0.0`. Fortunately, there is a standard `Default` trait for types that have default values. For numeric types, the default is always 0.

```
use std::ops::{Add, Mul};

fn dot<N: Add + Mul + Default>(v1: &[N], v2: &[N]) -> N {
    let mut total = N::default();
    for i in 0 .. v1.len() {
        total = total + v1[i] * v2[i];
    }
    total
}
```

```
    }
    total
}
```

This is closer, but still does not quite work:

```
error: mismatched types
|
11 |     total = total + v1[i] * v2[i];
|           ^^^^^^^^^^^^^ expected type parameter, found associated
type
|
= note: expected type `N`
      found type `<N as std::ops::Mul>::Output`
```

Our new code assumes that multiplying two values of type `N` produces another value of type `N`. This isn't necessarily the case. You can overload the multiplication 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 multiplication, where multiplying `N * N` returns an `N`. We do this by replacing `Mul` with `Mul<Output=N>`, and the same for `Add`:

```
fn dot<N: Add<Output=N> + Mul<Output=N> + Default>(v1: &[N], v2: &[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 dot<N>(v1: &[N], v2: &[N]) -> N
    where N: Add<Output=N> + Mul<Output=N> + Default
{
    ...
}
```

Great. But Rust still complains about this line of code:

```

error: cannot move out of type `[N]`, a non-copy slice
|
8 |     total = total + v1[i] * v2[i];
|     ^^^^^^
|     |
|     cannot move out of here
|     move occurs because `v1[_]` has type `N`,
|     which does not implement the `Copy` trait
|

```

Since we haven't required N to be a copyable type, Rust interprets `v1[i]` as an attempt to move a value out of the slice, which is forbidden. But we don't want to modify the slice at all; we just want to copy the values out to operate on them. Fortunately, all of Rust's built-in numeric types implement `Copy`, so we can simply add that to our constraints on N:

```
where N: Add<Output=N> + Mul<Output=N> + Default + Copy
```

With this, the code compiles and runs. The final code looks like this:

```

use std::ops::{Add, Mul};

fn dot<N>(v1: &[N], v2: &[N]) -> N
    where N: Add<Output=N> + Mul<Output=N> + Default + Copy
{
    let mut total = N::default();
    for i in 0 .. v1.len() {
        total = total + v1[i] * v2[i];
    }
    total
}

#[test]
fn test_dot() {
    assert_eq!(dot(&[1, 2, 3, 4], &[1, 1, 1, 1]), 10);
    assert_eq!(dot(&[53.0, 7.0], &[1.0, 5.0]), 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` that defines such a trait! Had we known, we could have added `num` to our `Cargo.toml` and written:

```
use num::Num;

fn dot<N: Num + Copy>(v1: &[N], v2: &[N]) -> N {
    let mut total = N::zero();
    for i in 0 .. v1.len() {
        total = total + v1[i] * v2[i];
    }
    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, à 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 two chapters cover 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.

About the Authors

Jim Blandy has been programming since 1981, and writing free software since 1990. He has been the maintainer of GNU Emacs and GNU Guile, and a maintainer of GDB, the GNU Debugger. He is one of the original designers of the Subversion version control system. Jim now works on Firefox's web developer tools for Mozilla.

Jason Orendorff hacks C++ for Mozilla, where he is module owner of the JavaScript engine that's in Firefox. He is an active member of the Nashville developer community and an occasional organizer of homegrown tech events. He is interested in grammar, baking, time travel, and helping people learn about complicated topics.

Leonora Tindall is a type system enthusiast and software engineer who uses Rust, Elixir, and other advanced languages to build robust and resilient systems software in high-impact areas like healthcare and data ownership. She works on a variety of open source projects, from genetic algorithms that evolve programs in strange languages to the Rust core libraries and crate ecosystem, and enjoys the experience of contributing to supportive and diverse community projects. In her free time, Leonora builds electronics for audio synthesis and is an avid radio hobbyist, and her love of hardware extends to her software engineering practice as well. She has built applications software for LoRa radios in Rust and Python, and uses software and DIY hardware to create experimental electronic music on a Eurorack synthesizer.

Colophon

The animal on the cover of *Programming Rust* is a Montagu's crab (*Xantho hydromilus*).

This robust-looking crab has a muscly appearance with a broad carapace about 70 mm wide. The edge of the carapace is furrowed and the color is yellowish or reddish-brown. It has 10 legs (five pairs): the front pair of legs (the chelipeds) are equal in size with black-tipped claws or pincers; then there are three pairs of walking legs that are stout and relatively short; and the last pair of legs are for swimming. They walk and swim sideways.

Montagu's crab has been found in the northeastern Atlantic ocean and in the Mediterranean Sea. It lives under rocks and boulders during low tide. If one is exposed when a rock has been lifted, it will aggressively hold its pincers up and spread them wide open to make itself appear bigger.

This crab eats algae, snails, and crabs of other species. It is mostly active at night. Egg-bearing females are found from March through July and the larvae are present on plankton for most of the summer.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The cover image is from *Wood's Natural History*. The cover fonts are URW Typewriter and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.

1. 1. Why Rust?

a. Type Safety

2. 2. Basic Types

a. Machine Types

- i. Integer Types
- ii. Checked, Wrapping, and Saturating Arithmetic
- iii. Floating-Point Types
- iv. The bool Type
- v. Characters

b. Tuples

c. Pointer Types

- i. References
- ii. Boxes
- iii. Raw Pointers

d. Arrays, Vectors, and Slices

- i. Arrays
- ii. Vectors
- iii. Slices

e. String Types

- i. String Literals
- ii. Byte Strings
- iii. Strings in Memory

- iv. String
- v. Using Strings
- vi. Other String-Like Types

- f. Beyond the Basics

3. 3. Ownership

- a. Ownership
- b. Moves
 - i. More Operations That Move
 - ii. Moves and Control Flow
 - iii. Moves and Indexed Content
- c. Copy Types: The Exception to Moves
- d. Rc and Arc: Shared Ownership

4. 4. References

- a. References as Values
 - i. Rust References Versus C++ References
 - ii. Assigning References
 - iii. References to References
 - iv. Comparing References
 - v. References Are Never Null
 - vi. Borrowing References to Arbitrary Expressions
 - vii. References to Slices and Trait Objects
- b. Reference Safety

- i. Borrowing a Local Variable
 - ii. Receiving References as Parameters
 - iii. Passing References as Arguments
 - iv. Returning References
 - v. Structs Containing References
 - vi. Distinct Lifetime Parameters
 - vii. Omitting Lifetime Parameters
- c. Sharing Versus Mutation
 - d. Taking Arms Against a Sea of Objects
5. 5. Expressions
- a. An Expression Language
 - b. Blocks and Semicolons
 - c. Declarations
 - d. if and match
 - i. if let
 - e. Loops
 - f. return Expressions
 - g. Why Rust Has loop
 - h. Function and Method Calls
 - i. Fields and Elements
 - j. Reference Operators
 - k. Arithmetic, Bitwise, Comparison, and Logical Operators

- l. Assignment
 - m. Type Casts
 - n. Closures
 - o. Precedence and Associativity
 - p. Onward
6. 6. Error Handling
- a. Panic
 - i. Unwinding
 - ii. Aborting
 - b. Result
 - i. Catching Errors
 - ii. Result Type Aliases
 - iii. Printing Errors
 - iv. Propagating Errors
 - v. Working with Multiple Error Types
 - vi. Dealing with Errors That “Can’t Happen”
 - vii. Ignoring Errors
 - viii. Handling Errors in main()
 - ix. Declaring a Custom Error Type
 - x. Why Results?
7. 7. Crates and Modules
- a. Crates

- i. Build Profiles
 - ii. Editions
- b. Modules
- i. Modules in Separate Files
 - ii. Paths and Imports
 - iii. The Standard Prelude
 - iv. Items, the Building Blocks of Rust
- c. Turning a Program into a Library
- d. The src/bin Directory
- e. Attributes
- f. Tests and Documentation
- i. Integration Tests
 - ii. Documentation
 - iii. Doc-Tests
- g. Specifying Dependencies
- i. Versions
 - ii. Cargo.lock
- h. Publishing Crates to crates.io
- i. Workspaces
 - j. More Nice Things

8. 8. Structs

- a. Named-Field Structs

- b. Tuple-Like Structs
- c. Unit-Like Structs
- d. Struct Layout
- e. Defining Methods with `impl`
 - i. Passing Self as a Box, Rc, or Arc
 - ii. Static Methods
- f. Static Values
- g. Generic Structs
- h. Structs with Lifetime Parameters
- i. Deriving Common Traits for Struct Types
- j. Interior Mutability

9.9. Enums and Patterns

- a. Enums
 - i. Enums with Data
 - ii. Enums in Memory
 - iii. Rich Data Structures Using Enums
 - iv. Generic Enums
- b. Patterns
 - i. Literals, Variables, and Wildcards in Patterns
 - ii. Tuple and Struct Patterns
 - iii. Array and Slice Patterns
 - iv. Reference Patterns

- v. Matching Multiple Possibilities
- vi. Pattern Guards
- vii. `@` Patterns
- viii. Where Patterns Are Allowed
- ix. Populating a Binary Tree

c. The Big Picture

10. Traits and Generics

a. Using Traits

- i. Trait Objects
- ii. Trait Object Layout
- iii. Generic Functions
- iv. Which to Use

b. Defining and Implementing Traits

- i. Default Methods
- ii. Traits and Other People's Types
- iii. Self in Traits
- iv. Subtraits
- v. Static Methods

c. Fully Qualified Method Calls

d. Traits That Define Relationships Between Types

- i. Associated Types (or How Iterators Work)
- ii. Generic Traits (or How Operator Overloading Works)

- iii. `impl Trait`
 - iv. `Associated Consts`
 - v. `Buddy Traits (or How rand::random() Works)`
- e. `Reverse-Engineering Bounds`
 - f. `Conclusion`