
Programming Rust

Fast, Safe Systems Development

Jim Blandy and Jason Orendorff

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Ownership

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 Nordin

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 runtime, 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 4-1](#).

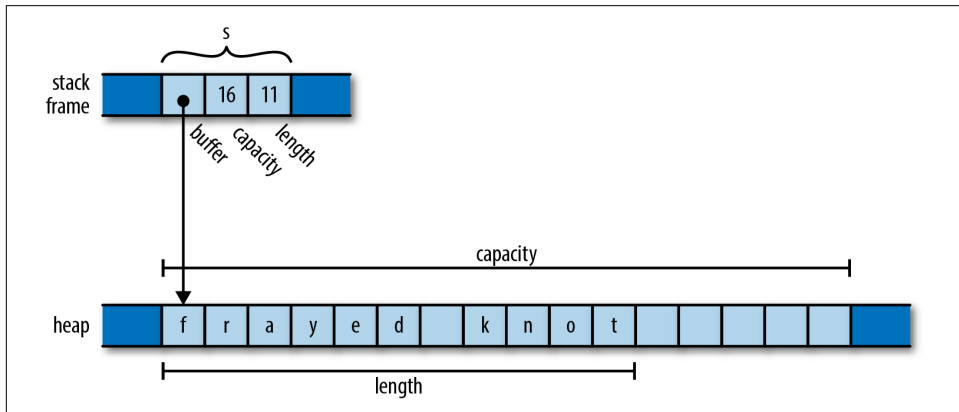


Figure 4-1. A C++ `std::string` value on the stack, pointing to its heap-allocated buffer

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 4-2](#).

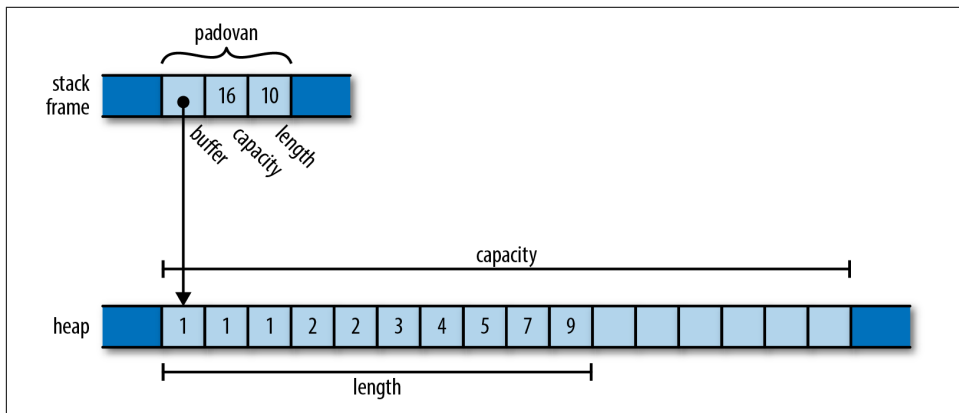


Figure 4-2. A `Vec 32` on the stack, pointing to its buffer in the heap

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 4-3](#).

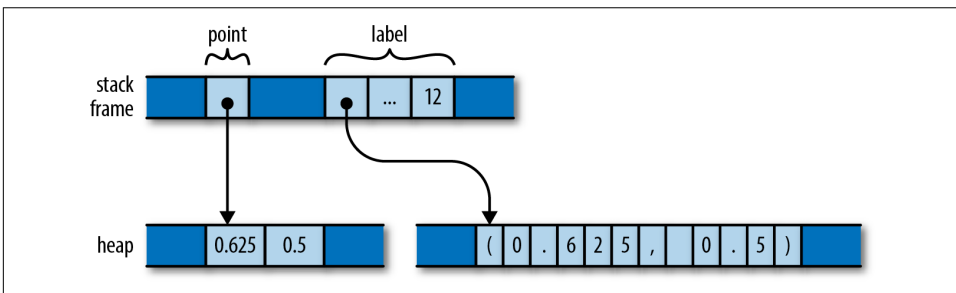


Figure 4-3. Two local variables, each owning memory in the heap

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 4-4](#).

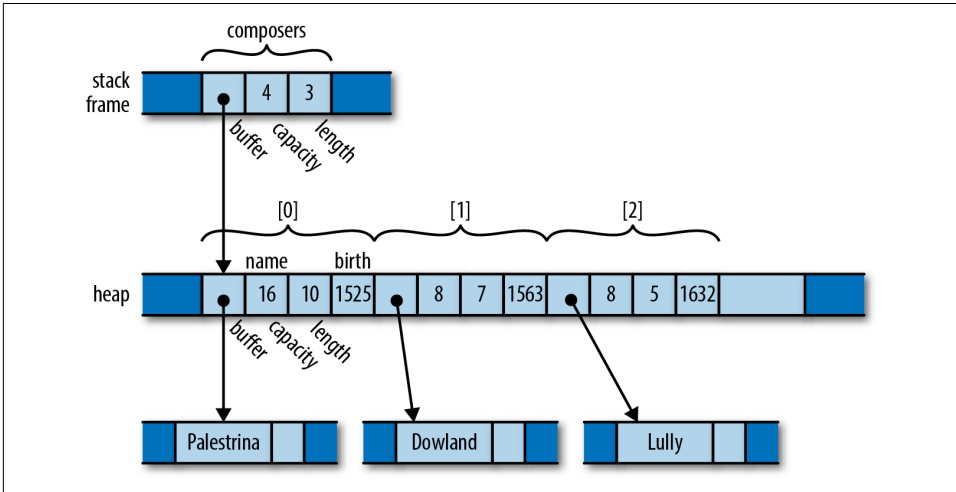


Figure 4-4. A more complex tree of ownership

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

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

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

than a tree. Every value in a Rust program is a member of some tree, rooted in some variable.

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

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

That said, 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 4-5** (note that some fields are left out).

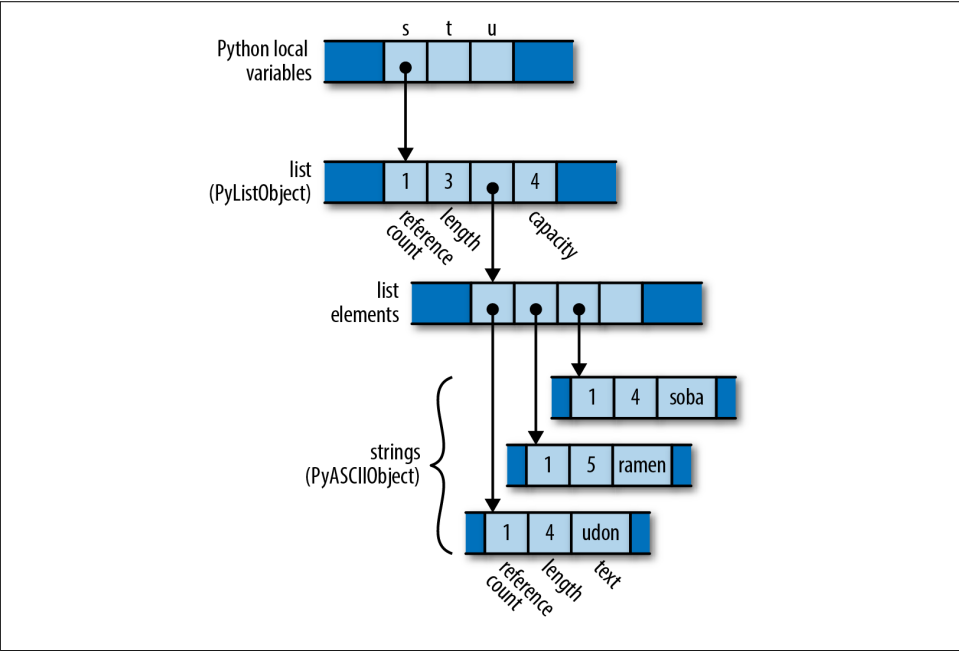


Figure 4-5. How Python represents a list of strings in memory

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 4-6](#).

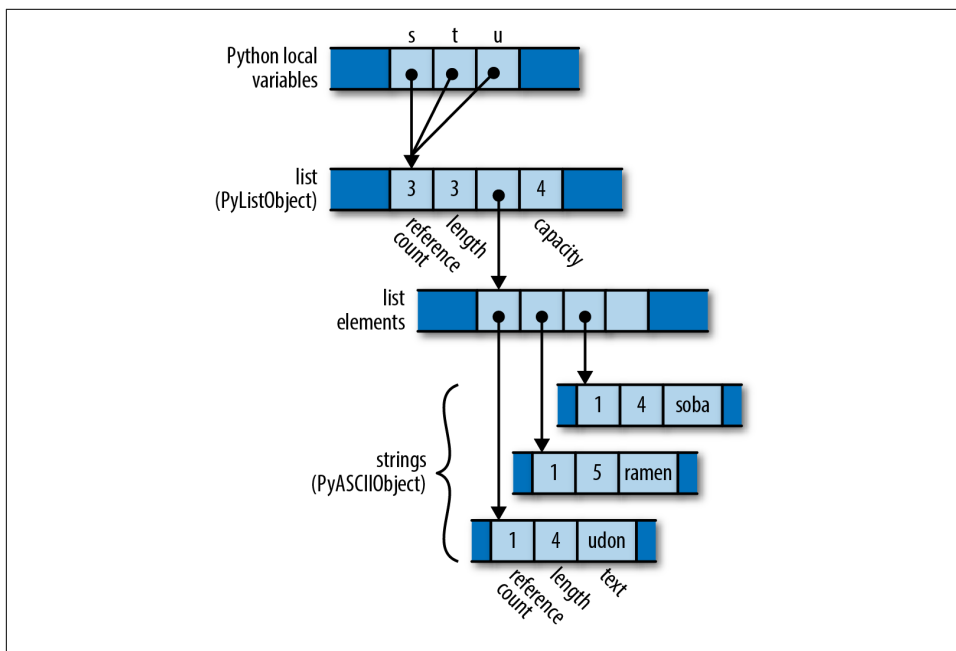


Figure 4-6. The result of assigning `s` to both `t` and `u` in Python

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 4-7](#) in memory.

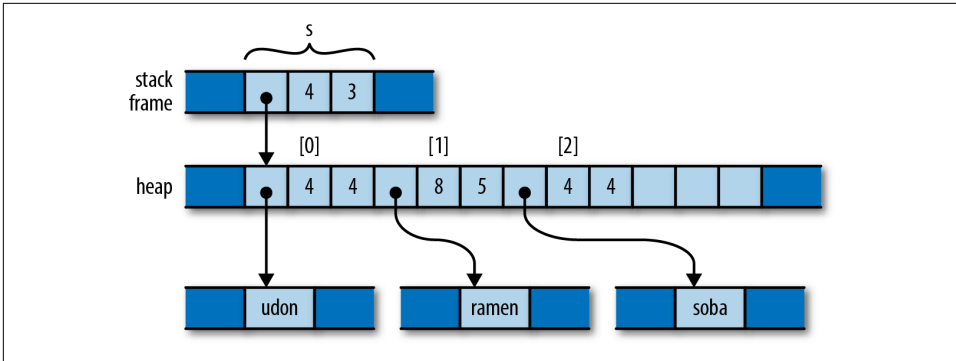


Figure 4-7. How C++ represents a vector of strings in memory

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 4-8).

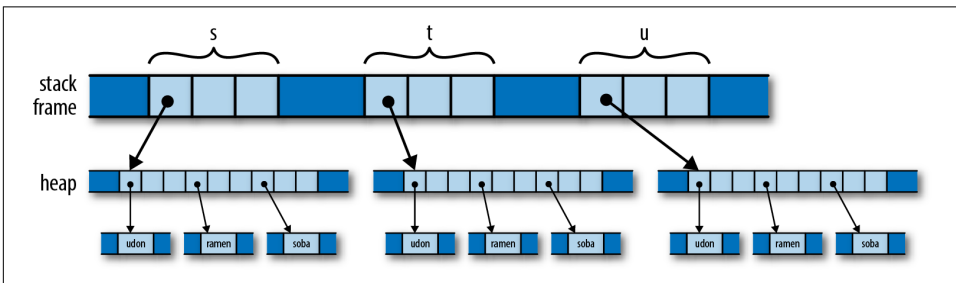


Figure 4-8. The result of assigning `s` to both `t` and `u` in C++

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 4-9).

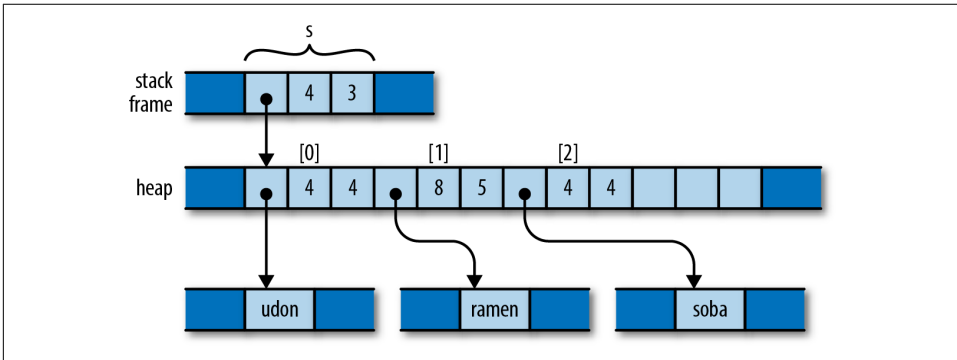


Figure 4-9. How Rust represents a vector of strings in memory

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 4-10.

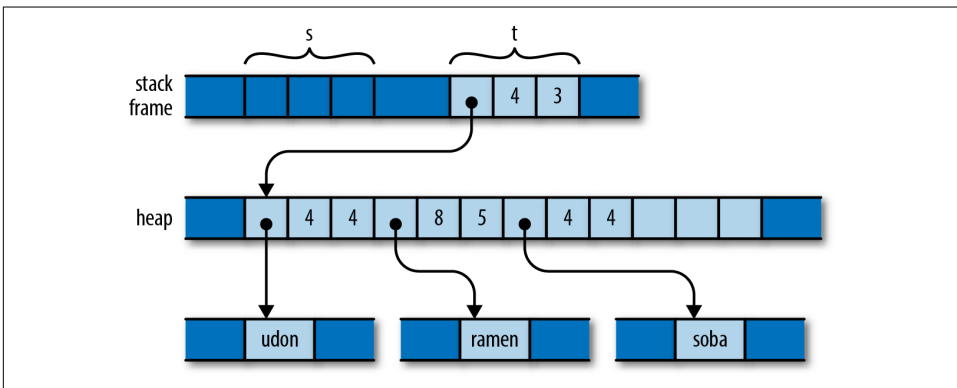


Figure 4-10. The result of assigning `s` to `t` in Rust

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” on page 90](#).

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 indexed content
--> ownership_move_out_of_vector.rs:14:17
|
14 |         let third = v[2];
|                        ^^^^
|                        |
|                        help: consider using a reference instead `&v[2]`
|                        cannot move out of indexed content
```

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().unwrap();
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 indexed content” error shown earlier. But because you’ve changed the type of the `name` field from `String` to `Option<String>`, that means that `None` is a legitimate value for the field to hold, so this works:

```

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

```

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 = "sommnambulance".to_string();
let str2 = str1;

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

After running this code, memory looks like [Figure 4-11](#).

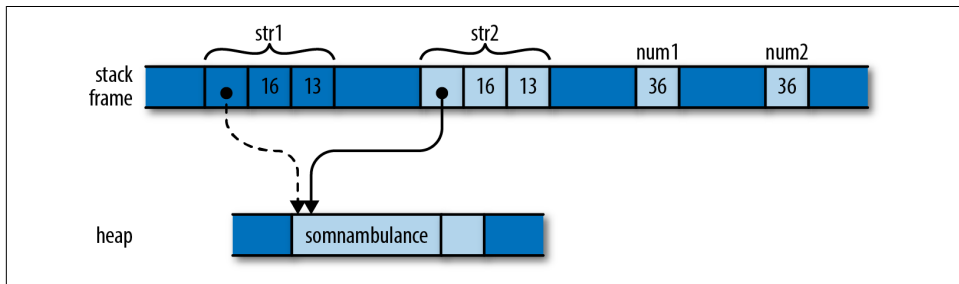


Figure 4-11. Assigning a string moves the value, whereas assigning an `i32` copies it

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[E0382]: use of moved value: `l.number`
--> ownership_struct.rs:12:40
|
11 |     print(l);
|         - value moved here
12 |     println!("My label number is: {}", l.number);
|                                         ^^^^^^^^^ value used here after move
|
= note: move occurs because `l` has type `main::Label`, which does not
       implement the `Copy` trait
```

Since `Label` is not Copy, passing it to `print` moved ownership of the value to the `print` function, which then dropped it before returning. But this is silly; a `Label` is nothing but an `i32` with pretensions. There's no reason passing `l` to `print` should move the value.

But 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 11](#), and Copy and Clone in particular in [Chapter 13](#).

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 4-12](#) 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.

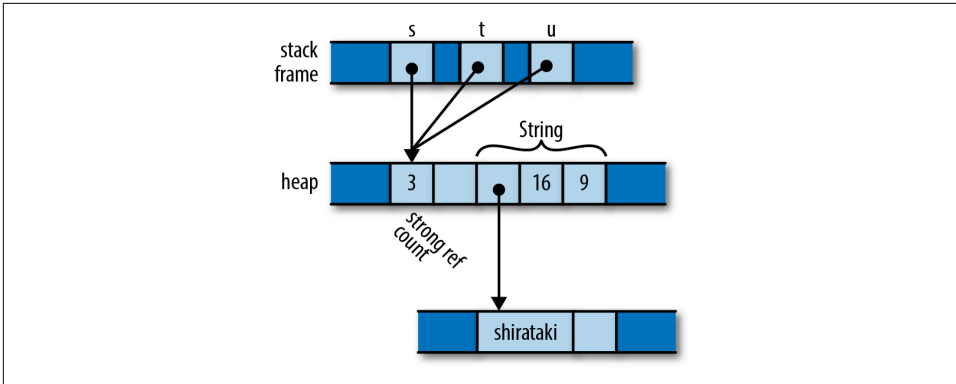


Figure 4-12. A reference-counted string, with three references

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!("{}", u);
```

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

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

Rust will decline:

```
error: cannot borrow immutable borrowed content as mutable
--> ownership_rc_mutability.rs:12:5
|
12 |     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 5](#).

One well-known problem with using reference counts to manage memory is that, if there are ever two reference-counted values that point to each other, each will hold the other's reference count above zero, so the values will never be freed ([Figure 4-13](#)).

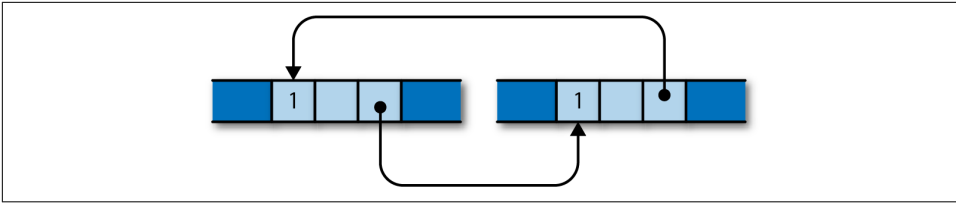


Figure 4-13. A reference-counting loop; these objects will not be freed

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](#)” on page 205. 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.

References

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[E0382]: use of moved value: `table`
--> references_show_moves_table.rs:29:16
   |
28 |     show(table);
   |           ----- value moved here
29 |     assert_eq!(table["Gesualdo"][0], "many madrigals");
   |                   ^^^^^ value used here after move
   |
   = note: move occurs because `table` has type `HashMap<String, Vec<String>>`,
           which does not implement the `Copy` trait
```

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 the two calls shown here 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 to 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 5-1](#).

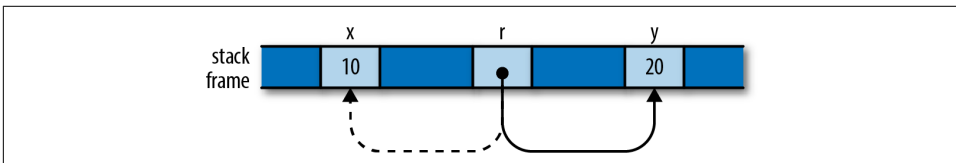


Figure 5-1. The reference `r`, now pointing to `y` instead of `x`

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 5-2](#).

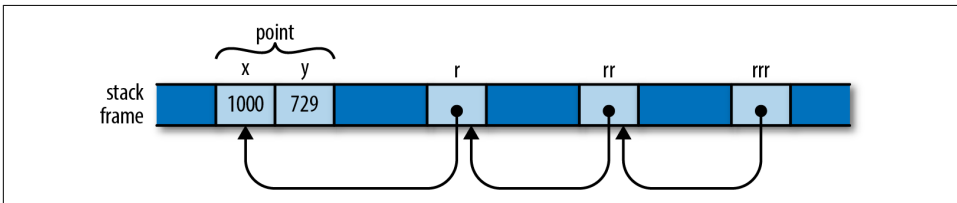


Figure 5-2. A chain of references to references

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

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

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);
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 3](#).

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” on page 238](#).

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 lexical block, a statement, an expression, the scope of some variable, or the like. Lifetimes are entirely figments of Rust's compile-time imagination. At runtime, a reference is nothing but an address; its lifetime is part of its type and has no runtime 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 5-3](#).

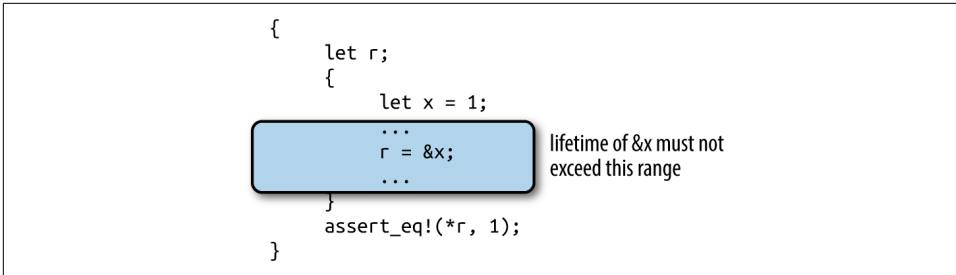


Figure 5-3. Permissible lifetimes for `&x`

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 5-4.

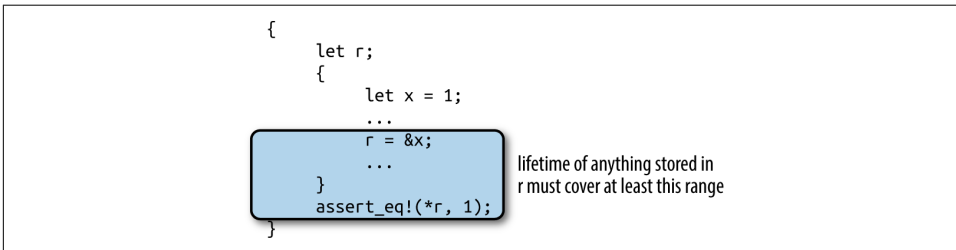


Figure 5-4. Permissible lifetimes for reference stored in `r`

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 5-5.

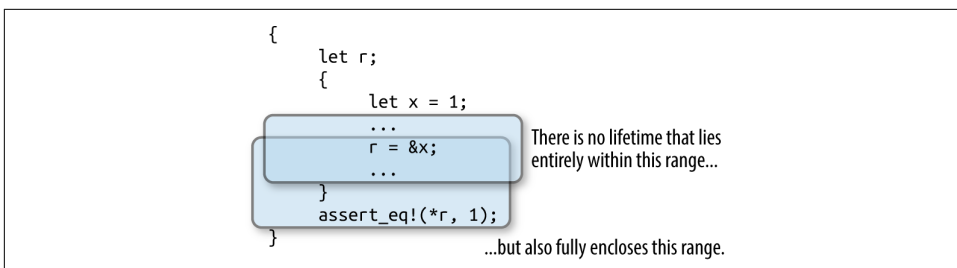


Figure 5-5. A reference with contradictory constraints on its lifetime

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 5-6.

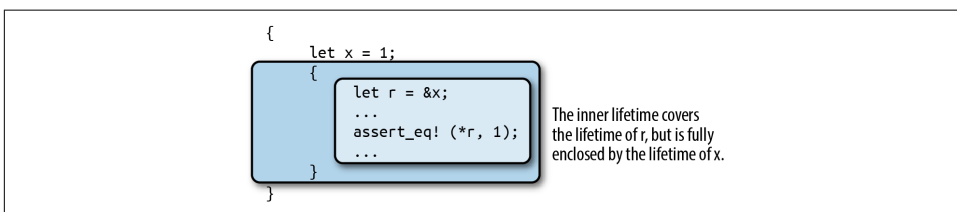


Figure 5-6. A reference with a lifetime enclosing `r`'s scope, but within `x`'s scope

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 8](#), 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 |     }
   |     ^ `parabola` dropped here while still borrowed
13 |     assert_eq!(*s, 0); // bad: points to element of dropped array
14 | }
   | - 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 case, if your function doesn't return any references (or other types that require lifetime parameters), then you 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.

But as one final shorthand, 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” on page 198](#).)

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 5-7](#).

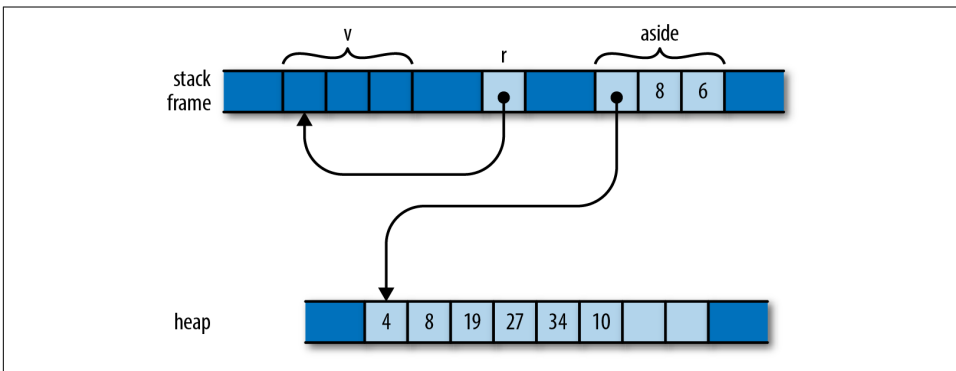


Figure 5-7. A reference to a vector that has been moved away

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 5-8](#).

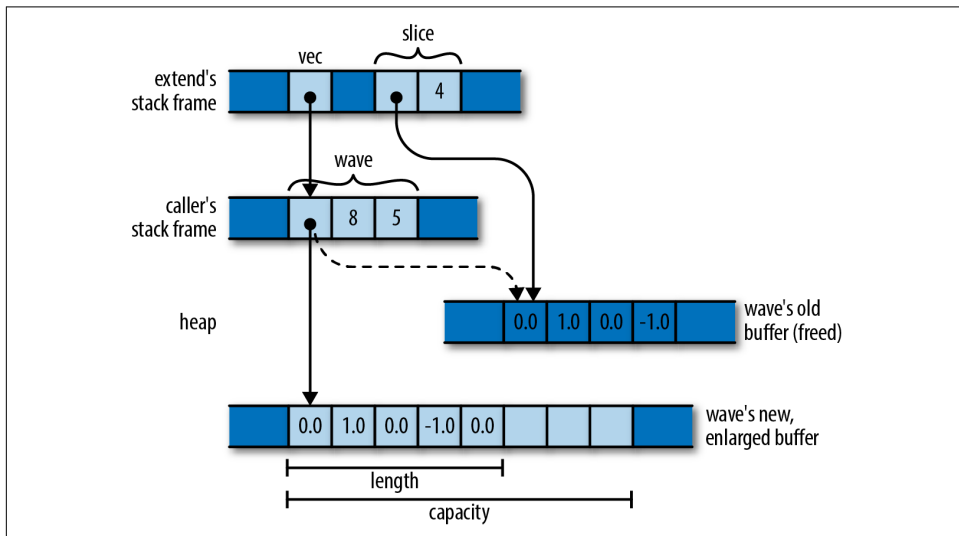


Figure 5-8. A slice turned into a dangling pointer by a vector reallocation

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 5-9](#)).

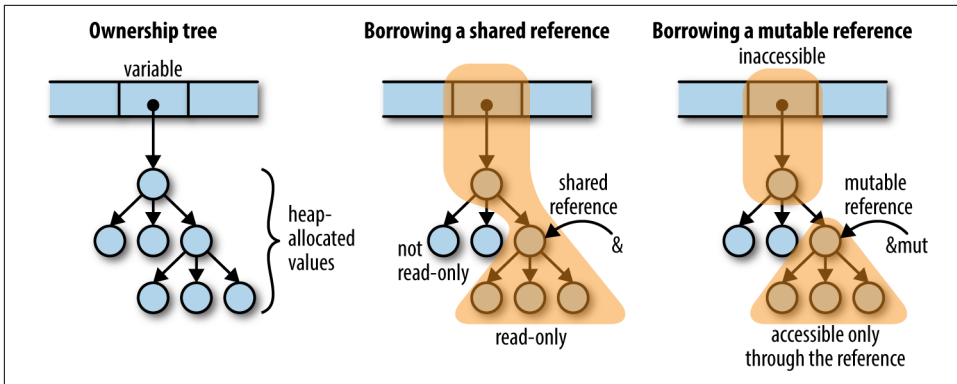


Figure 5-9. Borrowing a reference affects what you can do with other values in the same ownership tree

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

let mut y = 20;
let m1 = &mut y;
let m2 = &mut y; // error: cannot borrow as mutable more than once
let z = y;       // error: cannot use `y` because it was mutably borrowed
```

It is 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
```

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
```

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);
    }
};
```

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 9](#), 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 [Chapter 19](#), 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;       // nonconst 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 5-10](#).

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.")

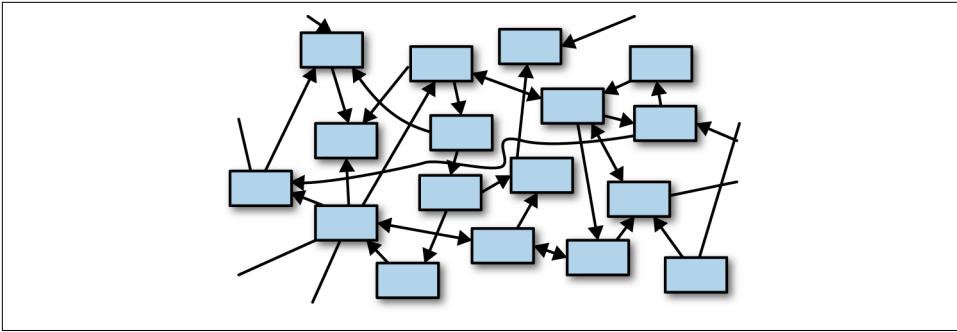


Figure 5-10. A sea of objects

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 5-11**.

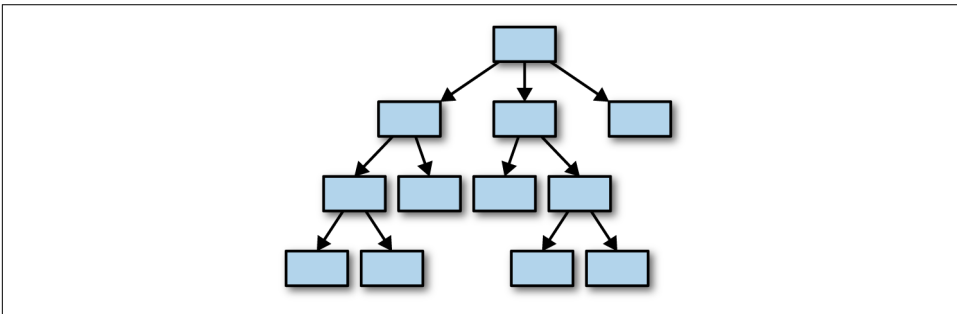


Figure 5-11. A tree of values

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.