2

FOUNDATIONS

As you dive into the more advanced corners of Rust, it's important that you ensure you have a solid understanding of the fundamentals. In Rust, as in any programming language, the precise meaning of various keywords and concepts becomes important as you begin to use the language in more sophisticated ways. In this chapter, we'll walk through many of Rust's primitives and try to define more clearly what they mean, how they work, and why they are exactly the way that they are. Specifically, we'll look at how variables and values differ, how they are represented in memory, and the different memory regions a program has. We'll then discuss some of the subtleties of ownership, borrowing, and lifetimes that you'll need to have a handle on before you continue with the book.

You can read this chapter top-to-bottom if you wish, or you can use it as a reference to brush up on the concepts that you feel less sure about. I recommend that you only move on when you feel completely comfortable with

the content of this chapter, as misconceptions about how these primitives work will quickly get in the way of understanding the more advanced topics, or lead to you using them incorrectly.

Talking About Memory

Not all memory is created equal. In most programming environments, your programs have access to a stack, a heap, registers, text segments, memory-mapped registers, memory-mapped files, and perhaps nonvolatile RAM. Which one you choose to use in a particular situation has implications for what you can store there, how long it remains accessible, and what mechanisms you use to access it. The exact details of these memory regions vary between platforms and are beyond the scope of this book, but some are so important to how you reason about Rust code that they are worth covering here.

Referring to Memory Locations

Before we dive into regions of memory, you first need to know about the difference between values, variables, and pointers. A *value* in Rust is the combination of a type and an element of that type's domain of values. A value can be turned into a sequence of bytes using its type's *representation*, but on its own you can think of a value more like "what you, the programmer, meant." For example, the number `6` in the type `u8` is an instance of the mathematical integer 6, and its in-memory representation is the byte `0x06`. Similarly, the `str` `"Hello world"` is a value in the domain of all strings whose representation is its UTF-8 encoding. A value's meaning is independent of the location where those bytes are stored.

A value is stored in a *place*, which is the Rust terminology for "a location that can hold a value." This place can be on the stack, on the heap, or a number of other locations. The most common place to store a value is a *variable*, which is a named value slot on the stack.

A *pointer* is a value that holds the address of a region of memory, so the pointer points to a place. A pointer can be dereferenced to access the value stored in the memory location it points to. We can store the same pointer in more than one variable, and therefore have multiple variables that indirectly refer to the same location in memory and thus the same underlying value.

Consider the code in Listing 2-1 that illustrates these three elements.

```
let x = 42;
let y = 43;
let var1 = &x;
let mut var2 = &x;
❶ var2 = &y;
```

Listing 2-1: Values, variables, and pointers

Here, there are four distinct values: `42` (an `i32`), `43` (an `i32`), the address of `x` (a pointer), and the address of `y` (a pointer). There are also four variables: `x`, `y`, `var1`, and `var2`. The latter two variables both hold values of the pointer type, because references are pointers. While `var1` and `var2` store the same value initially, they store separate, independent copies of that value; when we change the value stored in `var2` ❶, the value in `var1` does not change. In particular, the `=` operator stores the value of the right-hand side expression in the place named by the left-hand side.

An interesting example of where the distinction between variables, values, and pointers becomes important is in a statement such as:

```
let string = "Hello world";
```

Even though we assign a string value to the variable `string`, the *actual* value of the variable is a pointer to the first character in the string value `"Hello world"`, and not the string value itself. At this point you might say, "But hang on, where is the string value stored, then? Where does the pointer point?" If so, you have a keen eye—we'll get to that in a second.

What Is a Variable?

The definition of a variable I gave earlier is broad, and is unlikely to be all that useful in and of itself. As you encounter more complex code, you'll need a more accurate mental model to help you reason through what the programs are really doing. There are many such models that we can make use of. Describing them all in detail would take up several chapters and is beyond the scope of this book, but broadly speaking they can be divided into two categories: high-level models and low-level models. High-level models are useful when thinking about code at the level of lifetimes and borrows, while low-level models are good for when you are reasoning about unsafe code and raw pointers. The models for variables described in the following two sections will suffice for most of the material in this book.

High-Level Model

In the high-level model, we don't think of variables as places that hold bytes. Instead, we think of them just as names given to values as they are instantiated, moved, and used throughout a program. When you assign a value to a variable, that value is from then on named by that variable. When a variable is later accessed, you can imagine drawing a line from the previous access of that variable to the new access, which establishes a dependency relationship between the two accesses. If the value in a variable is moved, no lines can be drawn from it any more.

In this model, a variable only exists so long as it holds a legal value; you cannot draw lines from a variable whose value is uninitialized or has been moved, so effectively it isn't there. Using this model, your entire program

consists of many of these dependency lines, often called *flows*, each one tracing the lifetime of a particular instance of a value. Flows can fork and merge when there are branches, with each split tracing a distinct lifetime for that value. The compiler can check that at any given point in your program, all flows that can exist in parallel with each other are compatible. For example, there cannot be two parallel flows with mutable access to a value. Nor can there be a flow that borrows a value, but no flow that owns the value. Listing 2-2 shows examples of both of these cases.

```
let mut x;
// this access would be illegal, nowhere to draw the flow from:
// assert_eq!(x, 42);
❶ x = 42;
// this is okay, can draw a flow from the value assigned above:
❷ let y = &x;
// this establishes a second, mutable flow from x:
❸ x = 43;
// this continues the flow from y, which in turn draws from x:
❹ assert_eq!(*y, 42);
```

Listing 2-2: Illegal flows that the borrow checker will catch

First, we cannot use `x` before it is initialized, because we have nowhere to draw the flow from. Only when we assign a value to `x` can we draw flows from it. This code has two flows: one exclusive (`&mut`) flow from ❶ to ❸, and one shared (`&`) flow from ❶ through ❷ to ❹. The borrow checker inspects every vertex of every flow, and checks that no other incompatible flows exist concurrently. In this case, when the borrow checker inspects the exclusive flow at ❸, it sees the shared flow that terminates at ❹. Since you cannot have an exclusive and a shared use of a value at the same time, it (correctly) rejects the code. Notice that if ❹ was not there, this code would compile fine! The shared flow would terminate at ❷, and when the exclusive flow is checked at ❸, no conflicting flows would exist.

If a new variable is declared with the same name as a previous one, they are still considered distinct variables. This is called *shadowing*—the later variable "shadows" the former by the same name. The two variables coexist, though subsequent code no longer has a way to name the earlier one. This model matches roughly how the compiler, and the borrow checker in particular, reasons about your program, and is actually used internally in the compiler to produce efficient code.

Low-Level Model

Variables name memory locations that may or may not hold legal values. You can think of a variable as a "value slot." When you assign to it, the slot is filled, and its old value (if it had one) is dropped and replaced. When you access it, the compiler checks that the slot isn't empty, as that would mean the variable is uninitialized or its value has been moved. A pointer to a variable refers to the variable's backing memory, and can be dereferenced to get at its value. For example, in the statement `let x: usize`, the variable `x` is

a name for a region of memory that has room for a value the size of a `usize`, though it does not have a well-defined value (its slot is empty). If you assign a value to that variable, such as with `x = 6`, that region of memory will then hold the bits representing the value `6`. `&x` does not change when you assign to `x`. If you declare multiple variables with the same name, they still end up with different memory backing them. This model matches the memory model used by C and C++, and many other low-level languages, and is useful for when you need to reason explicitly about memory.

NOTE *In this example, we ignore CPU registers and treat them as an optimization. In reality, the compiler may use a register to back a variable instead of a region of memory if no memory address is needed for that variable.*

You may find that one of these matches your previous model better than the other, but I urge you to try to wrap your head around both of them. They are both equally valid, and both are simplifications, like any useful mental model has to be. If you are able to consider a piece of code from both of these perspectives, you will find it much easier to work through complicated code segments and understand why they do or do not compile and work as you expect.

Memory Regions

Now that you have a grip on how we refer to memory, we need to talk about what memory actually is. There are many different regions of memory, and perhaps surprisingly not all of them are stored in the DRAM of your computer. Which part of memory you use has a significant impact on how you write your code. The three most important regions for the purposes of writing Rust code are the stack, the heap, and static memory.

The Stack

The *stack* is a segment of memory that your program uses as scratch space for function calls. Each time a function is called, a contiguous chunk of memory called a *frame* is allocated at the top of the stack. Near the bottom of the stack is the frame for the `main` function, and as functions call other functions, additional frames are pushed onto the stack. A function's frame contains all the variables contained within that function, along with any arguments the function takes. When the function returns, its stack frame is reclaimed.

The bytes that make up the values of the function's local variables are not immediately wiped, but it's not safe to access them as they may have been overwritten by a subsequent function call whose frame overlaps with the reclaimed one. And even if they haven't been overwritten, they may contain values that are illegal to use, such as ones that were moved when the function returned.

Stack frames, and crucially the fact that they eventually disappear, are very closely tied to the notion of lifetimes in Rust. Any variable stored in a

frame on the stack cannot be accessed after that frame goes away, so any reference to it must have a lifetime that is at most as long as the lifetime of the frame.

The Heap

The *heap* is a pool of memory that isn't tied to the current call stack of the program. Values in heap memory live until they are explicitly deallocated. This is useful when you want a value to live beyond the lifetime of the current function's frame. If that value is the function's return value, the calling function can leave some space on its stack for the called function to write that value into before it returns. But if you want to, say, send that value to a different thread with which the current thread may share no stack frames at all, you can store it on the heap.

The heap is a large pool of memory from which you can explicitly allocate contiguous segments of memory. When you do so, you get a pointer to the start of that segment of memory. That memory segment is reserved for you until you later deallocate it; this process is often referred to as *freeing*, after the name of the corresponding function in the C standard library. Since allocations from the heap do not go away when a function returns, you can allocate memory for a value in one place, pass the pointer to it to another thread, and have that thread safely continue to operate on that value. Or, phrased differently, when you heap-allocate memory, the resulting pointer has an unconstrained lifetime—its lifetime is however long your program keeps it alive.

The primary mechanism for interacting with the heap in Rust is the `Box` type. When you write `Box::new(value)`, the value is placed on the heap, and what you are given back (the `Box<T>`) is a pointer to that value on the heap. When the `Box` is eventually dropped, that memory is freed.

If you forget to deallocate heap memory, it will stick around forever, and your application will eventually eat up all the memory on your machine. This is called *leaking memory*, and is usually something you want to avoid. However, there are some cases where you explicitly want to leak memory. For example, say you have a read-only configuration that the entire program should be able to access. You can allocate that on the heap and explicitly leak it with `Box::leak` to get a `'static` reference to it.

Static Memory

Static memory is really a catch-all term for several closely related regions located in the file your program is compiled into. These regions are automatically loaded into your program's memory when that program is executed. Values in static memory live for the entire execution of your program. Your program's static memory contains the program's binary code, which is usually mapped as read-only. As your program executes, it walks through the binary code in the text segment instruction by instruction, and jumps around whenever a function is called. Static memory also holds the memory for variables you declare with the `static` keyword, as well as certain constant values in your code, like strings.

The special lifetime `'static`, which get its name from the static memory region, marks a reference as being valid for "as long as static memory is around," which is until the program shuts down. Since a static variable's memory is allocated when the program starts, a reference to a variable in static memory is, by definition, `'static`, as it is not deallocated until the program shuts down. The inverse is not true—there can be `'static` references that do not point to static memory—but the name is still appropriate: once you create a reference with a static lifetime, whatever it points to might as well be in static memory as far as the rest of the program is concerned, as it can be used for however long your program wishes.

You will encounter the `'static` lifetime much more often than you will encounter truly static memory (through the `static` keyword, for example) when working with Rust. This is because `'static` often shows up in trait bounds on type parameters. A bound like `T: 'static` indicates that the type parameter `T` is able to live for however long we keep it around for, up to and including the remaining execution of the program. Essentially, this bound requires that `T` is owned and self-sufficient, either in that it does not borrow other (non-static) values or that anything it does borrow is also `'static` and thus will stick around until the end of the program. A good example of `'static` as a bound is the `std::thread::spawn` function that creates a new thread, which requires that the closure you pass it is `'static`. The new thread may outlive the current thread, so the new thread cannot refer to anything stored on the old thread's stack. The new thread can only refer to values that will live for its entire lifetime, which may be for the remaining duration of the program.

NOTE *You may wonder how `const` differs from `static`. The `const` keyword declares the following item as constant. Constant items can be completely computed at compile time, and any code that refers to them is replaced with the constant's computed value during compilation. A constant has no memory or other storage associated with it (it is not a place). You can think of constant as a convenient name for a particular value.*

Ownership

Rust's memory model centers around the idea that all values have a single *owner*—that is, there is exactly one location (usually a scope) that is responsible for ultimately deallocating each value. This is enforced through the borrow checker. If the value is moved, such as by assigning it to a new variable, pushing it to a vector, or placing it on the heap, the ownership of the value moves from the old location to the new one. At that point, you can no longer access the value through variables that flow from the original owner, even though the bits that make up the value are technically still there. Instead, you must access the moved value through variables that refer to its new location.

Some types are rebels, and do not follow this rule. If a value's type implements the special `Copy` trait, the value is not considered to have moved even if it is reassigned to a new memory location. Instead, the value is *copied*, and

both the old and new locations remain accessible. Essentially, another identical instance of that same value is constructed at the destination of the move. Most primitive types in Rust, such as the integer and floating-point types, are `Copy`. To be `Copy`, it must be possible to duplicate the type's values simply by copying their bits. This eliminates all types that *contain* non-`Copy` types, as well as any type that owns a resource it must deallocate when the value is dropped.

To see why, consider what would happen if a type like `Box` were `Copy`. If we executed `box2 = box1`, then `box1` and `box2` would both believe that they owned the heap memory allocated for the box, and they would both attempt to free it when they went out of scope. Freeing the memory twice could have catastrophic consequences.

When a value's owner no longer has use for it, it is the owner's responsibility to do any necessary cleanup for that value by *dropping* it. In Rust, dropping happens automatically when the variable that holds the value is no longer in scope. Types usually recursively drop values they contain, so dropping a variable of a complex type may result in many values being dropped. Because of Rust's discrete ownership requirement, we cannot accidentally drop the same value multiple times. A variable that holds a reference to another value does not own that other value, and so it's not dropped when the variable drops.

The code in Listing 2-3 gives a quick summary of the rules around ownership, move and copy semantics, and dropping.

```
let x1 = 42;
let y1 = Box::new(84);
{ // starts a new scope
❶   let z = (x1, y1);
    // z goes out of scope, and is dropped;
    // it in turn drops the values from x1 and y1
❷ }
  // x1's value is Copy, so it was not moved into z
❸ let x2 = x1;
  // y1's value is not Copy, so it was moved into z
❹ // let y2 = y1;
```

Listing 2-3: Moving and copying semantics

We start out with two values, the number `42` and a `Box` (a heap-allocated value) containing the number `84`. The former is `Copy`, whereas the latter is not. When we place `x1` and `y1` into the tuple `z` ❶, `x1` is *copied* into `z`, whereas `y1` is *moved* into `z`. At this point, `x1` continues to be accessible, and can be used again ❸. On the other hand, `y1` is rendered inaccessible once its value has been moved ❹, and any attempt to access it would incur a compiler error. When `z` goes out of scope ❷, the tuple value it contains is dropped, and this in turn drops the value copied from `x1` and the one moved from `y1`. When the `Box` from `y1` is dropped, it also deallocates the heap memory used to store `y1`'s value.

DROP ORDER

Rust automatically drops values when they go out of scope, such as `x1` and `x2` in the inner scope in Listing 2-3. The rules for the order in which to drop are fairly simple: variables (including function arguments) are dropped in reverse order and nested values are dropped in source-code order.

This might sound weird at first—why the discrepancy? If we look at it closely though, it makes a lot of sense. Say you write a function that declares a string, and then inserts a reference to that string into a new hash table. When the function returns, the hash table must be dropped first; if the string were dropped first, the hash table would then hold an invalid reference! In general, later variables may contain references to earlier values, whereas the inverse cannot happen due to Rust's lifetime rules. And for that reason, Rust drops variables in reverse order.

Now, we could have the same behavior for nested values, like the values in a tuple, array, or struct, but that would likely surprise users. If you constructed an array that contained two values, it'd seem odd if the last element of the array were dropped first. The same applies to tuples and structs, where the most intuitive behavior is for the first tuple element or field to be dropped first, then the second, and so on. Unlike for variables, there is no need to reverse the drop order in this case, since Rust doesn't (currently) allow self-references in a single value. So, Rust goes with the intuitive option.

Borrowing and Lifetimes

Rust allows the owner of a value to lend out references to that value to others without giving up ownership. *References* are pointers that come with an additional contract for how they can be used, such as whether the reference provides exclusive access to the referenced value, or whether the referenced value may also have other references point to it.

Shared References

A shared reference, `&T`, is, as the name implies, a pointer that may be shared. Any number of other references may exist to the same value, and each shared reference is `Copy`, so you can trivially make more of them. Values behind shared references are not mutable; you cannot modify or reassign the value a shared reference points to, nor can you cast a shared reference to a mutable one.

The Rust compiler is allowed to assume that the value a shared reference points to *will not change* while that reference lives. For example, if the Rust compiler sees that the value behind a shared reference is read multiple times in a function, it is within its rights to only read it once and reuse that value. More concretely, the assertion in Listing 2-4 should never fail.

```
fn cache(input: &i32, sum: &mut i32) {
  *sum = *input + *input;
  assert_eq!(*sum, 2 * *input);
}
```

Listing 2-4: Rust assumes that shared references are immutable.

Whether the compiler chooses to apply a given optimization or not is more or less irrelevant. The compiler heuristics change over time, so you generally want to code against what the compiler is allowed to do, rather than what it actually does in a particular case at a particular moment in time.

Mutable References

The alternative to a shared reference is a mutable reference: `&mut T`. With mutable references, the Rust compiler is again allowed to make full use of the contract that the reference comes with: the compiler assumes that there are no other threads accessing the target value, whether through a shared reference or a mutable one. In other words, it assumes that the mutable reference is *exclusive*. This enables some interesting optimizations that are not readily available in other languages. Take, for example, the code in Listing 2-5.

```
fn noalias(input: &i32, output: &mut i32) {
  if *input == 1 {
❶   *output = 2;
  }
❷ if *input != 1 {
    *output = 3;
  }
}
```

Listing 2-5: Rust assumes that mutable references are exclusive.

In Rust, the compiler can assume that `input` and `output` do not point to the same memory. Therefore, the reassignment of `output` at ❶ cannot affect the check at ❷, and the entire function can be compiled as an `if-else` block. If the compiler could not rely on the mutability contract, that optimization would be invalid, since an `input` of `1` could then result in an `output` of `3` in a case like `noalias(&x, &mut x)`.

A mutable reference only lets you mutate the memory location that the reference points to. Whether you can mutate values that lie beyond the immediate reference depends on the methods provided by the type that lies between. This may be easier to understand with an example, so consider Listing 2-6.

```
let x = 42;
let mut y = &x; // y is of type &i32
let z = &mut y; // z is of type &mut &i32
```

Listing 2-6: Mutability only applies to the immediately referenced memory.

In this example, you are able to change the value of the pointer `y` to a different value (that is, a different pointer) by making it reference a different variable, but you cannot change the value that is pointed to (that is, the value of `x`). Similarly, you can change the pointer value of `y` through `z`, but you cannot change `z` itself to point to a different value.

The primary difference between owning a value and having a mutable reference to it is that the owner is responsible for dropping the value when it is no longer necessary. Apart from that, you can do anything through a mutable reference that you can if you own the value, with one caveat: if you move the value behind the mutable reference, then you must leave another value in its place. The reason for this is simple; if you did not, the owner would still think it needed to drop the value, but there would be no value there for it to drop!

Listing 2-7 gives an example of the ways in which you can move the value behind a mutable reference.

```
fn replace_with_84(s: &mut Box<i32>) {
  // this is not okay, as *s would be empty:
❶ // let was = *s;
  // but this is:
❷ let was = std::mem::take(s);
  // so is this:
❸ *s = was;
  // we can exchange values behind &mut:
  let mut r = Box::new(84);
❹ std::mem::swap(s, &mut r);
  assert_ne!(*r, 84);
}
let mut s = Box::new(42);
replace_with_84(&mut s);
❺
```

Listing 2-7: Mutability only applies to the immediately referenced memory.

I've added commented-out lines that represent illegal operations. You cannot simply move the value out ❶, since the caller would still think they owned that value and would free it again at ❺, leading to a double free. If you just want to leave some valid value behind, `std::mem::take` ❷ is a good candidate. It is equivalent to `std::mem::replace(&mut value, Default::default())`; it moves `value` out from behind the mutable reference, but leaves a new, default value for the type in its place. The default is a separate, owned value, so it is safe for the caller to drop it when the scope is ended at ❺.

Alternatively, if you don't need the old value behind the reference, you can overwrite it with a value that you already own ❸, leaving it to the caller to drop the value later. When you do this, the value that used to be behind the mutable reference is dropped immediately.

Finally, if you have two mutable references, you can swap their values without owning either of them ❹, since both references will end up with a legal owned value for their owners to eventually free.

Interior Mutability

Some types provide *interior mutability*, meaning they allow you to mutate a value through a shared reference. These types usually rely on additional mechanisms (like atomic CPU instructions) or invariants to provide safe mutability without relying on the semantics of exclusive references. These normally fall into two categories: those that let you get a mutable reference through a shared reference, and those that just let you mutate through a shared reference.

The first category consists of types like `Mutex` and `RefCell` which contain safety mechanisms to ensure that, for any value they give a mutable reference to, only one mutable reference (and no shared references) can exist at a time. Under the hood, these types (and those like them) all rely on a type called `UnsafeCell`, whose name should immediately make you hesitate to use it. We will cover `UnsafeCell` in more detail in Chapter 10, but for now you should know that it is the *only* correct way to go from a `&T` to a `&mut T`.

Another category of types that provide interior mutability are those that do not give out a mutable reference to the inner value, but instead just give you methods for manipulating that value in place. The atomic integer types in `std::sync::atomic` and the `std::cell::Cell` type fall into this category. You cannot get a reference directly to the `usize` or `i32` behind such a type, but you can read and replace its value at a given point in time.

NOTE *The `Cell` type in the standard library is an interesting example of safe interior mutability through invariants. It is not shareable across threads, and never gives out a reference to the value contained in the `Cell`. Instead, the methods all either replace the value entirely or return a copy of the contained value. Since no references can exist to the inner value, it is always okay to move it. And since `Cell` isn't shareable across threads, the inner value will never be concurrently mutated even though mutation happens through a shared reference.*

Lifetimes

If you're reading this book, you're probably already familiar with the concept of lifetimes, likely through repeated complaining from the compiler about lifetime rules violations. That level of understanding will serve you well for the majority of Rust code you will write, but as we dive deeper into the more complex parts of Rust you will need a more rigorous mental model to work with.

Newer Rust developers are often taught to think of lifetimes as corresponding to scopes: a lifetime begins when you take a reference to some variable, and ends when that variable is moved or goes out of scope. That's often correct, and usually useful, but the reality is a little more complex. A *lifetime* is really a name for a region of code that some reference must be valid for. While a lifetime will frequently coincide with a scope, it does not have to, as we will see later in this section.

Lifetimes and the Borrow Checker

At the heart of Rust lifetimes is the *borrow checker*. Whenever a reference with some lifetime `'a` is used, the borrow checker checks that `'a` is still *alive*. It does this by tracing the path back to where `'a` starts—where the reference was taken—from the point of use, and checking that there are no conflicting uses along that path. This ensures that the reference still points to a value that it is safe to access. This is similar to the high-level "data flow" mental model we discussed earlier in the chapter; the compiler checks that the flow of the reference we are accessing does not conflict with any other parallel flows.

Listing 2-8 shows a simple code example with lifetime annotations for the reference to `x`.

```
  let mut x = Box::new(42);
❶ let r = &x;           // 'a
  if rand() > 0.5 {
❷  *x = 84;
  } else {
❸  println!("{}", r);  // 'a
  }
❹
```

Listing 2-8: Lifetimes do not need to be contiguous.

The lifetime starts at ❶ when we take a reference to `x`. In the first branch ❷, we then immediately try to modify `x` by changing its value to `84`, which requires a `&mut x`. The borrow checker takes out a mutable reference to `x` and immediately checks its use. It finds no conflicting uses between when the reference was taken and when it was used, so it accepts the code. This may come as a surprise if you are used to thinking about lifetimes as scopes, since `r` is still in scope at ❷ (it goes out of scope at ❹). But the borrow checker is smart enough to realize that `r` is never used later if this branch is taken, and therefore it is fine for `x` to be mutably accessed here. Or, phrased differently, the lifetime created at ❶ does not extend into this branch: there is no flow from `r` beyond ❷, and therefore there are no conflicting flows. The borrow checker then finds the use of `r` in the print statement at ❸. It walks the path back to ❶ and finds no conflicting uses (❷ is not on that path), so it accepts this use as well.

If we were to add another use of `r` at ❹ in Listing 2-8, the code would no longer compile. The lifetime `'a` would then last from ❶ all the way until ❹ (the last use of `r`), and when the borrow checker checked our new use of `r`, it would discover a conflicting use at ❷.

Lifetimes can get quite convoluted. In Listing 2-9 you can see an example of a lifetime that has *holes*, where it's intermittently invalid between where it starts and where it ultimately ends.

```
   let mut x = Box::new(42);
❶ let mut z = &x;          // 'a
   for i in 0..100 {
❷   println!("{}", z);    // 'a
```

```
❸  x = Box::new(i);
❹  z = &x;               // 'a
   }
   println!("{}", z);       // 'a
```

Listing 2-9: Lifetimes can have holes.

The lifetime starts at ❶ when we take a reference to `x`. We then move out of `x` at ❸, which ends the lifetime `'a`, since it is no longer valid. The borrow checker accepts this move by considering `'a` ended at ❷, which leaves no conflicting flows from `x` at ❸. Then, we restart the lifetime by updating the reference in `z` ❹. Whether the code now loops back around to ❷ or continues to the final print statement, both of those uses now have a valid value to flow from, and there are no conflicting flows, so the borrow checker accepts the code!

Again, this aligns perfectly with the data-flow model of memory we discussed earlier. When `x` is moved, `z` stops existing. When we reassign `z` later, we are creating an entirely new variable that only exists from that point forward. With that model in mind, this example is not weird.

Generic Lifetimes

Occasionally you need to store references within your own types. Those references need to have a lifetime so that the borrow checker can check their validity when they are used in the various methods on that type. This is especially true if you want a method on your type to return a reference that outlives the reference to `self`.

Rust allows you to make a type definition generic over one or more lifetimes, just like it allows you to make it generic over types. *The Rust Programming Language* by Steve Klabnik and Carol Nichols (No Starch Press, 2018) covers this topic in some detail, so I won't reiterate the basics here. But as you write more complex types of this nature, there are two subtleties around the interaction between such types and lifetimes that you should be aware of:

- If your type also implements `Drop`, then dropping your type counts as a use of any lifetime or type your type is generic over. Essentially, when an instance of your type is dropped, the borrow checker will check that it's still legal to use any of your type's generic lifetimes before dropping it. This is necessary in case your drop code *does* use any of those references. If your type does not implement `Drop`, dropping the type does *not* count as a use, and users are free to invalidate any references stored in your type as long as they do not use it any more, like we saw in Listing 2-7. We'll talk more about these rules around dropping in Chapter 10.
- While a type can be generic over multiple lifetimes, making it so often only serves to unnecessarily complicate your type signature. Usually, a type being generic over a single lifetime is fine, and the compiler will use the shorter of the lifetimes for any references inserted into your type as that one lifetime. You should only really use multiple generic

lifetime parameters if you have a type that contains multiple references, and its methods return references that should be tied to the lifetime of only *one* of those references.

Consider the type in Listing 2-10, which gives you an iterator over parts of a string separated by a particular string.

```
struct StrSplit<'s, 'p> {
  delimiter: &'p str,
  document: &'s str,
}
impl<'s, 'p> Iterator for StrSplit<'s, 'p> {
  type Output = &'s str;
  fn next(&self) -> Option<Self::Output> {
    todo!()
  }
}
fn str_before(s: &str, c: char) -> Option<&str> {
  StrSplit { document: s, delimiter: &c.to_string() }.next()
}
```

Listing 2-10: A type that needs to be generic over multiple lifetimes

When you construct this type, you have to give the `delimiter` and `document` to search, both of which are references to string values. When you ask for the next string, you get a reference into the document. Consider what would happen if you used a single lifetime in this type. The values yielded by the iterator would be tied to the lifetime of the `document` *and* the `delimiter`. This would make `str_before` impossible to write: the return type would have a lifetime associated with a variable local to the function—the `String` produced by `to_string`—and the borrow checker would reject the code.

Lifetime Variance

Variance is a concept that programmers are often exposed to but rarely know the name of, because it's mostly invisible. At a glance, variance describes what types are subtypes of other types, and when a subtype can be used in place of a supertype (and vice versa). Broadly speaking, a type `A` is a subtype of another type `B` if `A` is at least as useful as `B`. Variance is the reason why, in Java, you can pass a `Turtle` to a function that accepts an `Animal` if `Turtle` is a subtype of `Animal`, or why, in Rust, you can pass a `&'static str` to a function that accepts a `&'a str`.

While variance usually hides out of sight, it comes up often enough that we need to have a working knowledge of it. `Turtle` is a subtype of `Animal` because a `Turtle` is more "useful" than some unspecified `Animal`—a `Turtle` can do anything an `Animal` can do, and likely more. Similarly, `'static` is a subtype of `'a`, because a `'static` lives at least as long as any `'a`, and so is more useful. Or, more generally, if `'b: 'a` (`'b` outlives `'a`), then `'b` is a subtype of `'a`. This is obviously not the formal definition, but it gets close enough to be of practical use.

All types have a variance, which defines what other similar types can be used in that type's place. There are three kinds of variance: covariant, invariant, and contravariant. A type is *covariant* if you can just use a subtype in place of the type. For example, if a variable is of type `&'a T`, you can provide a value of type `&'static T` to it, because `&'a T` is covariant in `'a`. `&'a T` is also covariant in `T`, so you can pass a `&Vec<&'static str>` to a function that takes `&Vec<&'a str>`.

Some types are *invariant*, which means that you must provide exactly the given type. `&mut T` is an example of this—if a function takes a `&mut Vec<&'a str>`, you cannot pass it a `&mut Vec<&'static str>`. The reason for this is simple: if you could, the function could put a short-lived string inside the `Vec`, which the caller would then continue using thinking that it were a `Vec<&'static str>` and thus that the contained string were `'static`! Any type that provides mutability is generally invariant for the same reason—for example, `Cell<T>` is invariant in `T`.

The last category, *contravariance*, comes up for function arguments. Function types are more useful if they're okay with their arguments being *less* useful. This is clearer if you contrast the variance of the argument types on their own with their variance when used as function arguments:

```
let x: &'static str; // more useful, lives longer
let x: &'a      str; // less useful, lives shorter

fn take_func1(&'static str) // stricter, so less useful
fn take_func2(&'a str)      // less strict, more useful
```

This flipped relationship indicates that `Fn(T)` is contravariant in `T`.

So why do you need to learn about variance when it comes to lifetimes? Variance becomes relevant when you consider how generic lifetime parameters interact with the borrow checker. Consider a type like the one shown in Listing 2-11, which uses multiple lifetimes in a single field.

```
struct MutStr<'a, 'b> {
  s: &'a mut &'b str
}
let mut s = "hello";
❶ *MutStr { s: &mut s }.s = "world";
println!("{}", s);
```

Listing 2-11: A type that needs to be generic over multiple lifetimes

At first glance, using two lifetimes here seems unnecessary—we have no methods that need to differentiate between a borrow of different parts of the structure, as we did with `StrSplit` in Listing 2-10. But if you replace the two lifetimes here with a single `'a`, the code no longer compiles! And it's all because of variance.

At ❶, the compiler must determine what lifetime the lifetime parameter(s) should be set to. If there are two, `'a` is set to the to-be-determined lifetime of the borrow of `s`, and `'b` is set to `'static`, since that's the lifetime of the provided string `"hello"`. If there is just one lifetime `'a`,

however, the compiler infers that that lifetime must be `'static`. When we then attempt to access the string reference `s` through a shared reference to print it, the compiler tries to shorten the mutable borrow of `s` used by `MutStr` so that we are allowed to borrow `s` again.

In the two-lifetime case, `'a` simply ends just before the `println`, and `'b` stays the same. In the single-lifetime case, on the other hand, we run into issues. The compiler wants to shorten the borrow of `s`, but to do so it would also have to shorten the borrow of the `str`. While `&'static str` can in general be shortened to any `&'a str` (`&'a T` is covariant in `'a`), here it's behind a `&mut T`, which is invariant in `T`. Invariance requires that the relevant type is never replaced with a sub- or supertype, so the compiler's attempt to shorten the borrow fails, and it reports that the list is still mutably borrowed. Ouch!

In general, you want to ensure that your types remain covariant (or contravariant where appropriate) over as many of their generic parameters as possible. If that requires introducing additional lifetime arguments, you need to carefully weigh the cognitive cost of adding another parameter against the ergonomic cost of invariance.

Summary

The aim of this chapter has been to establish a solid, shared foundation that we can build on in the chapters to come. By now, I hope you feel that you have a firm grasp on Rust's memory and ownership model, and that those errors you may have gotten from the borrow checker seem less mysterious. You might have known bits and pieces of what we covered here already, but hopefully the chapter has given you a more holistic image of how it all fits together. In the next chapter, we will do something similar for types. We'll go over how types are represented in memory, see how generics and traits produce running code, and take a look at some of the special type and trait constructs Rust offers for more advanced use cases.