```
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 We then move out of `x` at ❸, which ends the lifetime `'a`, since it is no longer valid.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.