Generic Types, Traits, and Lifetimes

Every programming language has tools for effectively handling the duplication of concepts. In Rust, one such tool is *generics*: abstract stand-ins for concrete types or other properties. We can express the behavior of generics or how they relate to other generics without knowing what will be in their place when compiling and running the code.

Functions can take parameters of some generic type, instead of a concrete type like `i32` or `String`, in the same way they take parameters with unknown values to run the same code on multiple concrete values. In fact, we've already used generics in Chapter 6 with `Option<T>`, in Chapter 8 with `Vec<T>` and `HashMap<K, V>`, and in Chapter 9 with `Result<T, E>`. In this chapter, you'll explore how to define your own types, functions, and methods with generics!

First we'll review how to extract a function to reduce code duplication. We'll then use the same technique to make a generic function from two functions that differ only in the types of their parameters. We'll also explain how to use generic types in struct and enum definitions.

Then you'll learn how to use *traits* to define behavior in a generic way. You can combine traits with generic types to constrain a generic type to accept only those types that have a particular behavior, as opposed to just any type.

Finally, we'll discuss *lifetimes*: a variety of generics that give the compiler information about how references relate to each other. Lifetimes allow us to give the compiler enough information about borrowed values so that it can ensure references will be valid in more situations than it could without our help.

Removing Duplication by Extracting a Function

Generics allow us to replace specific types with a placeholder that represents multiple types to remove code duplication. Before diving into generics syntax, let's first look at how to remove duplication in a way that doesn't involve generic types by extracting a function that replaces specific values with a placeholder that represents multiple values. Then we'll apply the same technique to extract a generic function! By looking at how to recognize duplicated code you can extract into a function, you'll start to recognize duplicated code that can use generics.

We'll begin with the short program in Listing 10-1 that finds the largest number in a list.

Filename: src/main.rs

```
fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let mut largest = &number_list[0];

    for number in &number_list {
        if number > largest {
            largest = number;
        }
    }

    println!("The largest number is {largest}");
}
```

Listing 10-1: Finding the largest number in a list of numbers

We store a list of integers in the variable `number_list` and place a reference to the first number in the list in a variable named `largest`. We then iterate through all the numbers in the list, and if the current number is greater than the number stored in `largest`, we replace the reference in that variable. However, if the current number is less than or equal to the largest number seen so far, the variable doesn't change, and the code moves on to the next number in the list. After considering all the numbers in the list, `largest` should refer to the largest number, which in this case is 100.

We've now been tasked with finding the largest number in two different lists of numbers. To do so, we can choose to duplicate the code in Listing 10-1 and use the same logic at two different places in the program, as shown in Listing 10-2.

Filename: src/main.rs

```
fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let mut largest = &number_list[0];

    for number in &number_list {
        if number > largest {
            largest = number;
        }
    }

    println!("The largest number is {largest}");

    let number_list = vec![102, 34, 6000, 89, 54, 2, 43, 8];

    let mut largest = &number_list[0];

    for number in &number_list {
        if number > largest {
            largest = number;
        }
    }

    println!("The largest number is {largest}");
}
```

Listing 10-2: Code to find the largest number in *two* lists of numbers

Although this code works, duplicating code is tedious and error prone. We also have to remember to update the code in multiple places when we want to change it.

To eliminate this duplication, we'll create an abstraction by defining a function that operates on any list of integers passed in a parameter. This solution makes our code clearer and lets us express the concept of finding the largest number in a list abstractly.

In Listing 10-3, we extract the code that finds the largest number into a function named `largest`. Then we call the function to find the largest number in the two lists from Listing 10-2. We could also use the function on any other list of `i32` values we might have in the future.

Filename: src/main.rs

```
fn largest(list: &[i32]) -> &i32 {
    let mut largest = &list[0];

    for item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let result = largest(&number_list);
    println!("The largest number is {result}");

    let number_list = vec![102, 34, 6000, 89, 54, 2, 43, 8];

    let result = largest(&number_list);
    println!("The largest number is {result}");
}
```

Listing 10-3: Abstracted code to find the largest number in two lists

The `largest` function has a parameter called `list`, which represents any concrete slice of `i32` values we might pass into the function. As a result, when we call the function, the code runs on the specific values that we pass in.

In summary, here are the steps we took to change the code from Listing 10-2 to Listing 10-3:

1. Identify duplicate code.
2. Extract the duplicate code into the body of the function, and specify the inputs and return values of that code in the function signature.
3. Update the two instances of duplicated code to call the function instead.

Next, we'll use these same steps with generics to reduce code duplication. In the same way that the function body can operate on an abstract `list` instead of specific values, generics allow code to operate on abstract types.

For example, say we had two functions: one that finds the largest item in a slice of `i32` values and one that finds the largest item in a slice of `char` values. How would we eliminate that duplication? Let's find out!

Generic Data Types

We use generics to create definitions for items like function signatures or structs, which we can then use with many different concrete data types. Let's first look at how to define functions, structs, enums, and methods using generics. Then we'll discuss how generics affect code performance.

In Function Definitions

When defining a function that uses generics, we place the generics in the signature of the function where we would usually specify the data types of the parameters and return value. Doing so makes our code more flexible and provides more functionality to callers of our function while preventing code duplication.

Continuing with our `largest` function, Listing 10-4 shows two functions that both find the largest value in a slice. We'll then combine these into a single function that uses generics.

Filename: src/main.rs

```
fn largest_i32(list: &[i32]) -> &i32 {
    let mut largest = &list[0];

    for item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn largest_char(list: &[char]) -> &char {
    let mut largest = &list[0];

    for item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let result = largest_i32(&number_list);
    println!("The largest number is {result}");

    let char_list = vec!['y', 'm', 'a', 'q'];

    let result = largest_char(&char_list);
    println!("The largest char is {result}");
}
```

Listing 10-4: Two functions that differ only in their names and in the types in their signatures

The `largest_i32` function is the one we extracted in Listing 10-3 that finds the largest `i32` in a slice. The `largest_char` function finds the largest `char` in a slice. The function bodies have the same code, so let's eliminate the duplication by introducing a generic type parameter in a single function.

To parameterize the types in a new single function, we need to name the type parameter, just as we do for the value parameters to a function. You can use any identifier as a type parameter name. But we'll use `T` because, by convention, type parameter names in Rust are short, often just one letter, and Rust's type-naming convention is UpperCamelCase. Short for *type*, `T` is the default choice of most Rust programmers.

IMP

When we use a parameter in the body of the function, we have to declare the parameter name in the signature so the compiler knows what that name means. Similarly, when we use a type parameter name in a function signature, we have to declare the type parameter name before we use it. To define the generic `largest` function, we place type name declarations inside angle brackets, `<>`, between the name of the function and the parameter list, like this:

IMP

```
fn largest<T>(list: &[T]) -> &T {
```

We read this definition as: the function `largest` is generic over some type `T`. This function has one parameter named `list`, which is a slice of values of type `T`. The `largest` function will return a reference to a value of the same type `T`.

Listing 10-5 shows the combined `largest` function definition using the generic data type in its signature. The listing also shows how we can call the function with either a slice of `i32` values or `char` values. Note that this code won't compile yet, but we'll fix it later in this chapter.

Filename: src/main.rs

```
fn largest<T>(list: &[T]) -> &T {
    let mut largest = &list[0];

    for item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let result = largest(&number_list);
    println!("The largest number is {result}");

    let char_list = vec!['y', 'm', 'a', 'q'];

    let result = largest(&char_list);
    println!("The largest char is {result}");
}
```

Listing 10-5: The `largest` function using generic type parameters; this doesn't compile yet

If we compile this code right now, we'll get this error:

```
$ cargo run
   Compiling chapter10 v0.1.0 (file:///projects/chapter10)
error[E0369]: binary operation `>` cannot be applied to type `&T`
 --> src/main.rs:5:17
  |
5 |         if item > largest {
  |            ---- ^ ------- &T
  |            |
  |            &T
  |
help: consider restricting type parameter `T`
  |
1 | fn largest<T: std::cmp::PartialOrd>(list: &[T]) -> &T {
  |             ++++++++++++++++++++++

For more information about this error, try `rustc --explain E0369`.
error: could not compile `chapter10` (bin "chapter10") due to 1 previous error
```

The help text mentions `std::cmp::PartialOrd`, which is a *trait*, and we're going to talk about traits in the next section. For now, know that this error states that the body of `largest` won't work for all possible types that `T` could be. Because we want to compare values of type `T` in the body, we can only use types whose values can be ordered. To enable comparisons, the standard library has the `std::cmp::PartialOrd` trait that you can implement on types (see Appendix C for more on this trait). By following the help text's suggestion, we restrict the types valid for `T` to only those that implement `PartialOrd` and this example will compile, because the standard library implements `PartialOrd` on both `i32` and `char`.

In Struct Definitions

We can also define structs to use a generic type parameter in one or more fields using the `<>` syntax. Listing 10-6 defines a `Point<T>` struct to hold `x` and `y` coordinate values of any type.

Filename: src/main.rs

```
struct Point<T> {
    x: T,
    y: T,
}

fn main() {
    let integer = Point { x: 5, y: 10 };
    let float = Point { x: 1.0, y: 4.0 };
}
```

Listing 10-6: A `Point<T>` struct that holds `x` and `y` values of type `T`

The syntax for using generics in struct definitions is similar to that used in function definitions. First we declare the name of the type parameter inside angle brackets just after the name of the struct. Then we use the generic type in the struct definition where we would otherwise specify concrete data types.

Note that because we've used only one generic type to define `Point<T>`, this definition says that the `Point<T>` struct is generic over some type `T`, and the fields `x` and `y` are *both* that same type, whatever that type may be. If we create an instance of a `Point<T>` that has values of different types, as in Listing 10-7, our code won't compile.

Filename: src/main.rs

```
struct Point<T> {
    x: T,
    y: T,
}

fn main() {
    let wont_work = Point { x: 5, y: 4.0 };
}
```

Listing 10-7: The fields `x` and `y` must be the same type because both have the same generic data type `T`.

In this example, when we assign the integer value `5` to `x`, we let the compiler know that the generic type `T` will be an integer for this instance of `Point<T>`. Then when we specify `4.0` for `y`, which we've defined to have the same type as `x`, we'll get a type mismatch error like this:

```
$ cargo run
   Compiling chapter10 v0.1.0 (file:///projects/chapter10)
error[E0308]: mismatched types
 --> src/main.rs:7:38
  |
7 |     let wont_work = Point { x: 5, y: 4.0 };
  |                                      ^^^ expected integer, found floating-
point number

For more information about this error, try `rustc --explain E0308`.
error: could not compile `chapter10` (bin "chapter10") due to 1 previous error
```

To define a `Point` struct where `x` and `y` are both generics but could have different types, we can use multiple generic type parameters. For example, in Listing 10-8, we change the definition of `Point` to be generic over types `T` and `U` where `x` is of type `T` and `y` is of type `U`.

Filename: src/main.rs

```
struct Point<T, U> {
    x: T,
    y: U,
}

fn main() {
    let both_integer = Point { x: 5, y: 10 };
    let both_float = Point { x: 1.0, y: 4.0 };
    let integer_and_float = Point { x: 5, y: 4.0 };
}
```

Multiple Generic Types

Listing 10-8: A `Point<T, U>` generic over two types so that `x` and `y` can be values of different types

Now all the instances of `Point` shown are allowed! You can use as many generic type parameters in a definition as you want, but using more than a few makes your code hard to read. If you're finding you need lots of generic types in your code, it could indicate that your code needs restructuring into smaller pieces.

In Enum Definitions

As we did with structs, we can define enums to hold generic data types in their variants. Let's take another look at the `Option<T>` enum that the standard library provides, which we used in Chapter 6:

```
enum Option<T> {
    Some(T),
    None,
}
```

This definition should now make more sense to you. As you can see, the `Option<T>` enum is generic over type `T` and has two variants: `Some`, which holds one value of type `T`, and a `None` variant that doesn't hold any value. By using the `Option<T>` enum, we can express the abstract concept of an optional value, and because `Option<T>` is generic, we can use this abstraction no matter what the type of the optional value is.

Enums can use multiple generic types as well. The definition of the `Result` enum that we used in Chapter 9 is one example:

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

IMP

The `Result` enum is generic over two types, `T` and `E`, and has two variants: `Ok`, which holds a value of type `T`, and `Err`, which holds a value of type `E`. This definition makes it convenient to use the `Result` enum anywhere we have an operation that might succeed (return a value of some type `T`) or fail (return an error of some type `E`). In fact, this is what we used to open a file in Listing 9-3, where `T` was filled in with the type `std::fs::File` when the file was opened successfully and `E` was filled in with the type `std::io::Error` when there were problems opening the file.

When you recognize situations in your code with multiple struct or enum definitions that differ only in the types of the values they hold, you can avoid duplication by using generic types instead.

In Method Definitions

We can implement methods on structs and enums (as we did in Chapter 5) and use generic types in their definitions too. Listing 10-9 shows the `Point<T>` struct we defined in Listing 10-6 with a method named `x` implemented on it.

Filename: src/main.rs

```
struct Point<T> {
    x: T,
    y: T,
}

impl<T> Point<T> {
    fn x(&self) -> &T {
        &self.x
    }
}

fn main() {
    let p = Point { x: 5, y: 10 };

    println!("p.x = {}", p.x());
}
```

Listing 10-9: Implementing a method named `x` on the `Point<T>` struct that will return a reference to the `x` field of type `T`

Here, we've defined a method named `x` on `Point<T>` that returns a reference to the data in the field `x`.

Note that we have to declare `T` just after `impl` so we can use `T` to specify that we're implementing methods on the type `Point<T>`. By declaring `T` as a generic type after `impl`,

Rust can identify that the type in the angle brackets in `Point` is a generic type rather than a concrete type. We could have chosen a different name for this generic parameter than the generic parameter declared in the struct definition, but using the same name is conventional. Methods written within an `impl` that declares the generic type will be defined on any instance of the type, no matter what concrete type ends up substituting for the generic type.

We can also specify constraints on generic types when defining methods on the type. We could, for example, implement methods only on `Point<f32>` instances rather than on `Point<T>` instances with any generic type. In Listing 10-10 we use the concrete type `f32`, meaning we don't declare any types after `impl`.

Filename: src/main.rs

```
impl Point<f32> {
    fn distance_from_origin(&self) -> f32 {
        (self.x.powi(2) + self.y.powi(2)).sqrt()
    }
}
```

Listing 10-10: An `impl` block that only applies to a struct with a particular concrete type for the generic type parameter `T`

This code means the type `Point<f32>` will have a `distance_from_origin` method; other instances of `Point<T>` where `T` is not of type `f32` will not have this method defined. The method measures how far our point is from the point at coordinates (0.0, 0.0) and uses mathematical operations that are available only for floating-point types.

Generic type parameters in a struct definition aren't always the same as those you use in that same struct's method signatures. Listing 10-11 uses the generic types `X1` and `Y1` for the `Point` struct and `X2` `Y2` for the `mixup` method signature to make the example clearer. The method creates a new `Point` instance with the `x` value from the `self` `Point` (of type `X1`) and the `y` value from the passed-in `Point` (of type `Y2`).

Filename: src/main.rs

```
struct Point<X1, Y1> {
    x: X1,
    y: Y1,
}

impl<X1, Y1> Point<X1, Y1> {
    fn mixup<X2, Y2>(self, other: Point<X2, Y2>) -> Point<X1, Y2> {
        Point {
            x: self.x,
            y: other.y,
        }
    }
}

fn main() {
    let p1 = Point { x: 5, y: 10.4 };
    let p2 = Point { x: "Hello", y: 'c' };

    let p3 = p1.mixup(p2);

    println!("p3.x = {}, p3.y = {}", p3.x, p3.y);
}
```

Listing 10-11: A method that uses generic types different from its struct's definition

In `main`, we've defined a `Point` that has an `i32` for `x` (with value `5`) and an `f64` for `y` (with value `10.4`). The `p2` variable is a `Point` struct that has a string slice for `x` (with value `"Hello"`) and a `char` for `y` (with value `c`). Calling `mixup` on `p1` with the argument `p2` gives us `p3`, which will have an `i32` for `x` because `x` came from `p1`. The `p3` variable will have a `char` for `y` because `y` came from `p2`. The `println!` macro call will print `p3.x = 5, p3.y = c`.

The purpose of this example is to demonstrate a situation in which some generic parameters are declared with `impl` and some are declared with the method definition. Here, the generic parameters `X1` and `Y1` are declared after `impl` because they go with the struct definition. The generic parameters `X2` and `Y2` are declared after `fn mixup` because they're only relevant to the method.

Performance of Code Using Generics

You might be wondering whether there is a runtime cost when using generic type parameters. The good news is that using generic types won't make your program run any slower than it would with concrete types. IMP

Rust accomplishes this by performing monomorphization of the code using generics at compile time. *Monomorphization* is the process of turning generic code into specific code by filling in the

concrete types that are used when compiled. In this process, the compiler does the opposite of the steps we used to create the generic function in Listing 10-5: the compiler looks at all the places where generic code is called and generates code for the concrete types the generic code is called with.

Let's look at how this works by using the standard library's generic `Option<T>` enum:

```
let integer = Some(5);
let float = Some(5.0);
```

When Rust compiles this code, it performs monomorphization. During that process, the compiler reads the values that have been used in `Option<T>` instances and identifies two kinds of `Option<T>`: one is `i32` and the other is `f64`. As such, it expands the generic definition of `Option<T>` into two definitions specialized to `i32` and `f64`, thereby replacing the generic definition with the specific ones. IMP

The monomorphized version of the code looks similar to the following (the compiler uses different names than what we're using here for illustration):

Filename: src/main.rs

```
enum Option_i32 {
    Some(i32),
    None,
}

enum Option_f64 {
    Some(f64),
    None,
}

fn main() {
    let integer = Option_i32::Some(5);
    let float = Option_f64::Some(5.0);
}
```

The generic `Option<T>` is replaced with the specific definitions created by the compiler. Because Rust compiles generic code into code that specifies the type in each instance, we pay no runtime cost for using generics. When the code runs, it performs just as it would if we had duplicated each definition by hand. The process of monomorphization makes Rust's generics extremely efficient at runtime.

In monomorphization, we pay a compile time cost, rather than run-time cost

Traits: Defining Shared Behavior

A *trait* defines the functionality a particular type has and can share with other types. We can use traits to define shared behavior in an abstract way. We can use *trait bounds* to specify that a generic type can be any type that has certain behavior.

> Note: Traits are similar to a feature often called *interfaces* in other languages, although with some differences.

Defining a Trait

A type's behavior consists of the methods we can call on that type. Different types share the same behavior if we can call the same methods on all of those types. Trait definitions are a way to group method signatures together to define a set of behaviors necessary to accomplish some purpose.

For example, let's say we have multiple structs that hold various kinds and amounts of text: a `NewsArticle` struct that holds a news story filed in a particular location and a `Tweet` that can have, at most, 280 characters along with metadata that indicates whether it was a new tweet, a retweet, or a reply to another tweet.

We want to make a media aggregator library crate named `aggregator` that can display summaries of data that might be stored in a `NewsArticle` or `Tweet` instance. To do this, we need a summary from each type, and we'll request that summary by calling a `summarize` method on an instance. Listing 10-12 shows the definition of a public `Summary` trait that expresses this behavior.

Filename: src/lib.rs

```
pub trait Summary {
    fn summarize(&self) -> String;
}
```

Listing 10-12: A `Summary` trait that consists of the behavior provided by a `summarize` method

Here, we declare a trait using the `trait` keyword and then the trait's name, which is `Summary` in this case. We also declare the trait as `pub` so that crates depending on this crate can make use of this trait too, as we'll see in a few examples. Inside the curly brackets, we declare the

method signatures that describe the behaviors of the types that implement this trait, which in this case is `fn summarize(&self) -> String`.

After the method signature, instead of providing an implementation within curly brackets, we use a semicolon. Each type implementing this trait must provide its own custom behavior for the body of the method. The compiler will enforce that any type that has the `Summary` trait will have the method `summarize` defined with this signature exactly.

A trait can have multiple methods in its body: the method signatures are listed one per line, and each line ends in a semicolon.

Implementing a Trait on a Type

Now that we've defined the desired signatures of the `Summary` trait's methods, we can implement it on the types in our media aggregator. Listing 10-13 shows an implementation of the `Summary` trait on the `NewsArticle` struct that uses the headline, the author, and the location to create the return value of `summarize`. For the `Tweet` struct, we define `summarize` as the username followed by the entire text of the tweet, assuming that the tweet content is already limited to 280 characters.

Filename: src/lib.rs

```
pub struct NewsArticle {
    pub headline: String,
    pub location: String,
    pub author: String,
    pub content: String,
}

impl Summary for NewsArticle {
    fn summarize(&self) -> String {
        format!("{}, by {} ({})", self.headline, self.author, self.location)
    }
}

pub struct Tweet {
    pub username: String,
    pub content: String,
    pub reply: bool,
    pub retweet: bool,
}

impl Summary for Tweet {
    fn summarize(&self) -> String {
        format!("{}: {}", self.username, self.content)
    }
}
```

Listing 10-13: Implementing the `Summary` trait on the `NewsArticle` and `Tweet` types

Implementing a trait on a type is similar to implementing regular methods. The difference is that after `impl`, we put the trait name we want to implement, then use the `for` keyword, and then specify the name of the type we want to implement the trait for. Within the `impl` block, we put the method signatures that the trait definition has defined. Instead of adding a semicolon after each signature, we use curly brackets and fill in the method body with the specific behavior that we want the methods of the trait to have for the particular type.

Now that the library has implemented the `Summary` trait on `NewsArticle` and `Tweet`, users of the crate can call the trait methods on instances of `NewsArticle` and `Tweet` in the same way we call regular methods. The only difference is that the user must bring the trait into scope as well as the types. Here's an example of how a binary crate could use our `aggregator` library crate:

```
use aggregator::{Summary, Tweet};

fn main() {
    let tweet = Tweet {
        username: String::from("horse_ebooks"),
        content: String::from(
            "of course, as you probably already know, people",
        ),
        reply: false,
        retweet: false,
    };

    println!("1 new tweet: {}", tweet.summarize());
}
```

This code prints `1 new tweet: horse_ebooks: of course, as you probably already know, people`.

Other crates that depend on the `aggregator` crate can also bring the `Summary` trait into scope to implement `Summary` on their own types. One restriction to note is that we can implement a trait on a type only if either the trait or the type, or both, are local to our crate. For example, we can implement standard library traits like `Display` on a custom type like `Tweet` as part of our `aggregator` crate functionality because the type `Tweet` is local to our `aggregator` crate. We can also implement `Summary` on `Vec<T>` in our `aggregator` crate because the trait `Summary` is local to our `aggregator` crate.

But we can't implement external traits on external types. For example, we can't implement the `Display` trait on `Vec<T>` within our `aggregator` crate because `Display` and `Vec<T>` are both defined in the standard library and aren't local to our `aggregator` crate. This restriction is part of a property called *coherence*, and more specifically the *orphan rule*, so named because the

parent type is not present. This rule ensures that other people's code can't break your code and vice versa. Without the rule, two crates could implement the same trait for the same type, and Rust wouldn't know which implementation to use.

Default Implementations

Sometimes it's useful to have default behavior for some or all of the methods in a trait instead of requiring implementations for all methods on every type. Then, as we implement the trait on a particular type, we can keep or override each method's default behavior.

In Listing 10-14, we specify a default string for the `summarize` method of the `Summary` trait instead of only defining the method signature, as we did in Listing 10-12.

Filename: src/lib.rs

```
pub trait Summary {
    fn summarize(&self) -> String {
        String::from("(Read more...)")
    }
}
```

Listing 10-14: Defining a `Summary` trait with a default implementation of the `summarize` method

To use a default implementation to summarize instances of `NewsArticle`, we specify an empty `impl` block with `impl Summary for NewsArticle {}`.

Even though we're no longer defining the `summarize` method on `NewsArticle` directly, we've provided a default implementation and specified that `NewsArticle` implements the `Summary` trait. As a result, we can still call the `summarize` method on an instance of `NewsArticle`, like this:

```
let article = NewsArticle {
    headline: String::from("Penguins win the Stanley Cup Championship!"),
    location: String::from("Pittsburgh, PA, USA"),
    author: String::from("Iceburgh"),
    content: String::from(
        "The Pittsburgh Penguins once again are the best \
         hockey team in the NHL.",
    ),
};

println!("New article available! {}", article.summarize());
```

This code prints `New article available! (Read more...)`.

Creating a default implementation doesn't require us to change anything about the implementation of `Summary` on `Tweet` in Listing 10-13. The reason is that the syntax for overriding a default implementation is the same as the syntax for implementing a trait method that doesn't have a default implementation.

Default implementations can call other methods in the same trait, even if those other methods don't have a default implementation. In this way, a trait can provide a lot of useful functionality and only require implementors to specify a small part of it. For example, we could define the `Summary` trait to have a `summarize_author` method whose implementation is required, and then define a `summarize` method that has a default implementation that calls the `summarize_author` method:

```
pub trait Summary {
    fn summarize_author(&self) -> String;

    fn summarize(&self) -> String {
        format!("(Read more from {}...)", self.summarize_author())
    }
}
```

To use this version of `Summary`, we only need to define `summarize_author` when we implement the trait on a type:

```
impl Summary for Tweet {
    fn summarize_author(&self) -> String {
        format!("@{}", self.username)
    }
}
```

After we define `summarize_author`, we can call `summarize` on instances of the `Tweet` struct, and the default implementation of `summarize` will call the definition of `summarize_author` that we've provided. Because we've implemented `summarize_author`, the `Summary` trait has given us the behavior of the `summarize` method without requiring us to write any more code. Here's what that looks like:

```
    let tweet = Tweet {
        username: String::from("horse_ebooks"),
        content: String::from(
            "of course, as you probably already know, people",
        ),
        reply: false,
        retweet: false,
    };

    println!("1 new tweet: {}", tweet.summarize());
```

This code prints `1 new tweet: (Read more from @horse_ebooks...)`.

Note that it isn't possible to call the default implementation from an overriding implementation of that same method.

Traits as Parameters

Now that you know how to define and implement traits, we can explore how to use traits to define functions that accept many different types. We'll use the `Summary` trait we implemented on the `NewsArticle` and `Tweet` types in Listing 10-13 to define a `notify` function that calls the `summarize` method on its `item` parameter, which is of some type that implements the `Summary` trait. To do this, we use the `impl Trait` syntax, like this:

```
pub fn notify(item: &impl Summary) {
    println!("Breaking news! {}", item.summarize());
}
```

Instead of a concrete type for the `item` parameter, we specify the `impl` keyword and the trait name. This parameter accepts any type that implements the specified trait. In the body of `notify`, we can call any methods on `item` that come from the `Summary` trait, such as `summarize`. We can call `notify` and pass in any instance of `NewsArticle` or `Tweet`. Code that calls the function with any other type, such as a `String` or an `i32`, won't compile because those types don't implement `Summary`.

Trait Bound Syntax

The `impl Trait` syntax works for straightforward cases but is actually syntax sugar for a longer form known as a *trait bound*; it looks like this:

```
pub fn notify<T: Summary>(item: &T) {
    println!("Breaking news! {}", item.summarize());
}
```

This longer form is equivalent to the example in the previous section but is more verbose. We place trait bounds with the declaration of the generic type parameter after a colon and inside angle brackets.

The `impl Trait` syntax is convenient and makes for more concise code in simple cases, while the fuller trait bound syntax can express more complexity in other cases. For example, we can have two parameters that implement `Summary`. Doing so with the `impl Trait` syntax looks like this:

```
pub fn notify(item1: &impl Summary, item2: &impl Summary) {
```

Using `impl Trait` is appropriate if we want this function to allow `item1` and `item2` to have different types (as long as both types implement `Summary`). If we want to force both parameters to have the same type, however, we must use a trait bound, like this:

```
pub fn notify<T: Summary>(item1: &T, item2: &T) {
```

The generic type `T` specified as the type of the `item1` and `item2` parameters constrains the function such that the concrete type of the value passed as an argument for `item1` and `item2` must be the same.

Specifying Multiple Trait Bounds with the + Syntax

We can also specify more than one trait bound. Say we wanted `notify` to use display formatting as well as `summarize` on `item`: we specify in the `notify` definition that `item` must implement both `Display` and `Summary`. We can do so using the `+` syntax:

```
pub fn notify(item: &(impl Summary + Display)) {
```

The `+` syntax is also valid with trait bounds on generic types:

```
pub fn notify<T: Summary + Display>(item: &T) {
```

With the two trait bounds specified, the body of `notify` can call `summarize` and use `{}` to format `item`.

Clearer Trait Bounds with `where` Clauses

Using too many trait bounds has its downsides. Each generic has its own trait bounds, so functions with multiple generic type parameters can contain lots of trait bound information between the function's name and its parameter list, making the function signature hard to read. For this reason, Rust has alternate syntax for specifying trait bounds inside a `where` clause after the function signature. So, instead of writing this:

```
fn some_function<T: Display + Clone, U: Clone + Debug>(t: &T, u: &U) -> i32 {
```

we can use a `where` clause, like this:

```
fn some_function<T, U>(t: &T, u: &U) -> i32
where
    T: Display + Clone,
    U: Clone + Debug,
{
```

This function's signature is less cluttered: the function name, parameter list, and return type are close together, similar to a function without lots of trait bounds.

Returning Types That Implement Traits

We can also use the `impl Trait` syntax in the return position to return a value of some type that implements a trait, as shown here:

```
fn returns_summarizable() -> impl Summary {
    Tweet {
        username: String::from("horse_ebooks"),
        content: String::from(
            "of course, as you probably already know, people",
        ),
        reply: false,
        retweet: false,
    }
}
```

By using `impl Summary` for the return type, we specify that the `returns_summarizable` function returns some type that implements the `Summary` trait without naming the concrete type. In this case, `returns_summarizable` returns a `Tweet`, but the code calling this function doesn't need to know that.

The ability to specify a return type only by the trait it implements is especially useful in the context of closures and iterators, which we cover in Chapter 13. Closures and iterators create types that only the compiler knows or types that are very long to specify. The `impl Trait` syntax lets you concisely specify that a function returns some type that implements the `Iterator` trait without needing to write out a very long type.

However, you can only use `impl Trait` if you're returning a single type. For example, this code that returns either a `NewsArticle` or a `Tweet` with the return type specified as `impl Summary` wouldn't work:

```
fn returns_summarizable(switch: bool) -> impl Summary {
    if switch {
        NewsArticle {
            headline: String::from(
                "Penguins win the Stanley Cup Championship!",
            ),
            location: String::from("Pittsburgh, PA, USA"),
            author: String::from("Iceburgh"),
            content: String::from(
                "The Pittsburgh Penguins once again are the best \
                 hockey team in the NHL.",
            ),
        }
    } else {
        Tweet {
            username: String::from("horse_ebooks"),
            content: String::from(
                "of course, as you probably already know, people",
            ),
            reply: false,
            retweet: false,
        }
    }
}
```

Returning either a `NewsArticle` or a `Tweet` isn't allowed due to restrictions around how the `impl Trait` syntax is implemented in the compiler. We'll cover how to write a function with this behavior in the "Using Trait Objects That Allow for Values of Different Types" section of Chapter 17.

Using Trait Bounds to Conditionally Implement Methods

By using a trait bound with an `impl` block that uses generic type parameters, we can implement methods conditionally for types that implement the specified traits. For example, the type `Pair<T>` in Listing 10-15 always implements the `new` function to return a new instance of `Pair<T>` (recall from the "Defining Methods" section of Chapter 5 that `Self` is a type alias for the type of the `impl` block, which in this case is `Pair<T>`). But in the next `impl` block, `Pair<T>` only implements the `cmp_display` method if its inner type `T` implements the `PartialOrd` trait that enables comparison *and* the `Display` trait that enables printing.

Filename: src/lib.rs

```
use std::fmt::Display;

struct Pair<T> {
    x: T,
    y: T,
}

impl<T> Pair<T> {
    fn new(x: T, y: T) -> Self {
        Self { x, y }
    }
}

impl<T: Display + PartialOrd> Pair<T> {
    fn cmp_display(&self) {
        if self.x >= self.y {
            println!("The largest member is x = {}", self.x);
        } else {
            println!("The largest member is y = {}", self.y);
        }
    }
}
```

Listing 10-15: Conditionally implementing methods on a generic type depending on trait bounds

We can also conditionally implement a trait for any type that implements another trait. Implementations of a trait on any type that satisfies the trait bounds are called *blanket implementations* and are used extensively in the Rust standard library. For example, the standard library implements the `ToString` trait on any type that implements the `Display` trait. The `impl` block in the standard library looks similar to this code:

```
impl<T: Display> ToString for T {
    // --snip--
}
```

Because the standard library has this blanket implementation, we can call the `to_string` method defined by the `ToString` trait on any type that implements the `Display` trait. For example, we can turn integers into their corresponding `String` values like this because integers implement `Display`:

```
let s = 3.to_string();
```

Blanket implementations appear in the documentation for the trait in the “Implementors” section.

Traits and trait bounds let us write code that uses generic type parameters to reduce duplication but also specify to the compiler that we want the generic type to have particular

behavior. The compiler can then use the trait bound information to check that all the concrete types used with our code provide the correct behavior. In dynamically typed languages, we would get an error at runtime if we called a method on a type which didn't define the method. But Rust moves these errors to compile time so we're forced to fix the problems before our code is even able to run. Additionally, we don't have to write code that checks for behavior at runtime because we've already checked at compile time. Doing so improves performance without having to give up the flexibility of generics.

Validating References with Lifetimes

Lifetimes are another kind of generic that we've already been using. Rather than ensuring that a type has the behavior we want, lifetimes ensure that references are valid as long as we need them to be.

One detail we didn't discuss in the "References and Borrowing" section in Chapter 4 is that every reference in Rust has a *lifetime*, which is the scope for which that reference is valid. Most of the time, lifetimes are implicit and inferred, just like most of the time, types are inferred. We must annotate types only when multiple types are possible. In a similar way, we must annotate lifetimes when the lifetimes of references could be related in a few different ways. Rust requires us to annotate the relationships using generic lifetime parameters to ensure the actual references used at runtime will definitely be valid.

Annotating lifetimes is not a concept most other programming languages have, so this is going to feel unfamiliar. Although we won't cover lifetimes in their entirety in this chapter, we'll discuss common ways you might encounter lifetime syntax so you can get comfortable with the concept.

Preventing Dangling References with Lifetimes

The main aim of lifetimes is to prevent *dangling references*, which cause a program to reference data other than the data it's intended to reference. Consider the program in Listing 10-16, which has an outer scope and an inner scope.

```
fn main() {
    let r;

    {
        let x = 5;
        r = &x;
    }

    println!("r: {r}");
}
```

Listing 10-16: An attempt to use a reference whose value has gone out of scope

---

Note: The examples in Listing 10-16, 10-17, and 10-23 declare variables without giving them an initial value, so the variable name exists in the outer scope. At first glance, this

might appear to be in conflict with Rust's having no null values. However, if we try to use a variable before giving it a value, we'll get a compile-time error, which shows that Rust indeed does not allow null values.

The outer scope declares a variable named `r` with no initial value, and the inner scope declares a variable named `x` with the initial value of `5`. Inside the inner scope, we attempt to set the value of `r` as a reference to `x`. Then the inner scope ends, and we attempt to print the value in `r`. This code won't compile because the value that `r` is referring to has gone out of scope before we try to use it. Here is the error message:

```
$ cargo run
   Compiling chapter10 v0.1.0 (file:///projects/chapter10)
error[E0597]: `x` does not live long enough
 --> src/main.rs:6:13
  |
5 |         let x = 5;
  |             - binding `x` declared here
6 |         r = &x;
  |             ^^ borrowed value does not live long enough
7 |     }
  |     - `x` dropped here while still borrowed
8 |
9 |     println!("r: {r}");
  |                  --- borrow later used here

For more information about this error, try `rustc --explain E0597`.
error: could not compile `chapter10` (bin "chapter10") due to 1 previous error
```

The error message says that the variable `x` "does not live long enough." The reason is that `x` will be out of scope when the inner scope ends on line 7. But `r` is still valid for the outer scope; because its scope is larger, we say that it "lives longer." If Rust allowed this code to work, `r` would be referencing memory that was deallocated when `x` went out of scope, and anything we tried to do with `r` wouldn't work correctly. So how does Rust determine that this code is invalid? It uses a borrow checker.

The Borrow Checker

The Rust compiler has a *borrow checker* that compares scopes to determine whether all borrows are valid. Listing 10-17 shows the same code as Listing 10-16 but with annotations showing the lifetimes of the variables.

```
fn main() {
    let r;                // ---------+-- 'a
                          //          |
    {                     //          |
        let x = 5;        // -+-- 'b  |
        r = &x;           //  |       |
    }                     // -+       |
                          //          |
    println!("r: {r}");   //          |
}                         // ---------+
```

Listing 10-17: Annotations of the lifetimes of `r` and `x`, named `'a` and `'b`, respectively

Here, we've annotated the lifetime of `r` with `'a` and the lifetime of `x` with `'b`. As you can see, the inner `'b` block is much smaller than the outer `'a` lifetime block. At compile time, Rust compares the size of the two lifetimes and sees that `r` has a lifetime of `'a` but that it refers to memory with a lifetime of `'b`. The program is rejected because `'b` is shorter than `'a`: the subject of the reference doesn't live as long as the reference.

Listing 10-18 fixes the code so it doesn't have a dangling reference and it compiles without any errors.

```
fn main() {
    let x = 5;            // ----------+-- 'b
                          //           |
    let r = &x;           // --+-- 'a  |
                          //   |       |
    println!("r: {r}");   //   |       |
                          // --+       |
}                         // ----------+
```

Listing 10-18: A valid reference because the data has a longer lifetime than the reference

Here, `x` has the lifetime `'b`, which in this case is larger than `'a`. This means `r` can reference `x` because Rust knows that the reference in `r` will always be valid while `x` is valid.

Now that you know what the lifetimes of references are and how Rust analyzes lifetimes to ensure references will always be valid, let's explore generic lifetimes of parameters and return values in the context of functions.

Generic Lifetimes in Functions

We'll write a function that returns the longer of two string slices. This function will take two string slices and return a single string slice. After we've implemented the `longest` function, the code in Listing 10-19 should print `The longest string is abcd`.

Filename: src/main.rs

```
fn main() {
    let string1 = String::from("abcd");
    let string2 = "xyz";

    let result = longest(string1.as_str(), string2);
    println!("The longest string is {result}");
}
```

Listing 10-19: A `main` function that calls the `longest` function to find the longer of two string slices

Note that we want the function to take string slices, which are references, rather than strings, because we don't want the `longest` function to take ownership of its parameters. Refer to the "String Slices as Parameters" section in Chapter 4 for more discussion about why the parameters we use in Listing 10-19 are the ones we want.

If we try to implement the `longest` function as shown in Listing 10-20, it won't compile.

Filename: src/main.rs

```
fn longest(x: &str, y: &str) -> &str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

Listing 10-20: An implementation of the `longest` function that returns the longer of two string slices but does not yet compile

Instead, we get the following error that talks about lifetimes:

```
$ cargo run
   Compiling chapter10 v0.1.0 (file:///projects/chapter10)
error[E0106]: missing lifetime specifier
 --> src/main.rs:9:33
  |
9 | fn longest(x: &str, y: &str) -> &str {
  |               ----     ----     ^ expected named lifetime parameter
  |
  = help: this function's return type contains a borrowed value, but the signature
does not say whether it is borrowed from `x` or `y`
help: consider introducing a named lifetime parameter
  |
9 | fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
  |           ++++     ++          ++          ++

error: lifetime may not live long enough
  --> src/main.rs:11:9
   |
9  | fn longest(x: &str, y: &str) -> &str {
   |               - let's call the lifetime of this reference `'1`
10 |     if x.len() > y.len() {
11 |         x
   |         ^ returning this value requires that `'1` must outlive `'static`

error: lifetime may not live long enough
  --> src/main.rs:13:9
   |
9  | fn longest(x: &str, y: &str) -> &str {
   |                         - let's call the lifetime of this reference `'2`
...
13 |         y
   |         ^ returning this value requires that `'2` must outlive `'static`

For more information about this error, try `rustc --explain E0106`.
error: could not compile `chapter10` (bin "chapter10") due to 3 previous errors
```

The help text reveals that the return type needs a generic lifetime parameter on it because Rust can't tell whether the reference being returned refers to `x` or `y`. Actually, we don't know either, because the `if` block in the body of this function returns a reference to `x` and the `else` block returns a reference to `y`!

When we're defining this function, we don't know the concrete values that will be passed into this function, so we don't know whether the `if` case or the `else` case will execute. We also don't know the concrete lifetimes of the references that will be passed in, so we can't look at the scopes as we did in Listings 10-17 and 10-18 to determine whether the reference we return will always be valid. The borrow checker can't determine this either, because it doesn't know how the lifetimes of `x` and `y` relate to the lifetime of the return value. To fix this error, we'll add generic lifetime parameters that define the relationship between the references so the borrow checker can perform its analysis.

Lifetime Annotation Syntax

Lifetime annotations don't change how long any of the references live. Rather, they describe the relationships of the lifetimes of multiple references to each other without affecting the lifetimes. Just as functions can accept any type when the signature specifies a generic type parameter, functions can accept references with any lifetime by specifying a generic lifetime parameter.

Lifetime annotations have a slightly unusual syntax: the names of lifetime parameters must start with an apostrophe (`'`) and are usually all lowercase and very short, like generic types. Most people use the name `'a` for the first lifetime annotation. We place lifetime parameter annotations after the `&` of a reference, using a space to separate the annotation from the reference's type.

Here are some examples: a reference to an `i32` without a lifetime parameter, a reference to an `i32` that has a lifetime parameter named `'a` , and a mutable reference to an `i32` that also has the lifetime `'a` .

```
&i32        // a reference
&'a i32     // a reference with an explicit lifetime
&'a mut i32 // a mutable reference with an explicit lifetime
```

One lifetime annotation by itself doesn't have much meaning because the annotations are meant to tell Rust how generic lifetime parameters of multiple references relate to each other. Let's examine how the lifetime annotations relate to each other in the context of the `longest` function.

Lifetime Annotations in Function Signatures

To use lifetime annotations in function signatures, we need to declare the generic *lifetime* parameters inside angle brackets between the function name and the parameter list, just as we did with generic *type* parameters.

We want the signature to express the following constraint: the returned reference will be valid as long as both the parameters are valid. This is the relationship between lifetimes of the parameters and the return value. We'll name the lifetime `'a` and then add it to each reference, as shown in Listing 10-21.

Filename: src/main.rs

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

Listing 10-21: The `longest` function definition specifying that all the references in the signature must have the same lifetime `'a`

This code should compile and produce the result we want when we use it with the `main` function in Listing 10-19.

The function signature now tells Rust that for some lifetime `'a`, the function takes two parameters, both of which are string slices that live at least as long as lifetime `'a`. The function signature also tells Rust that the string slice returned from the function will live at least as long as lifetime `'a`. In practice, it means that the lifetime of the reference returned by the `longest` function is the same as the smaller of the lifetimes of the values referred to by the function arguments. These relationships are what we want Rust to use when analyzing this code.

Remember, when we specify the lifetime parameters in this function signature, we're not changing the lifetimes of any values passed in or returned. Rather, we're specifying that the borrow checker should reject any values that don't adhere to these constraints. Note that the `longest` function doesn't need to know exactly how long `x` and `y` will live, only that some scope can be substituted for `'a` that will satisfy this signature.

When annotating lifetimes in functions, the annotations go in the function signature, not in the function body. The lifetime annotations become part of the contract of the function, much like the types in the signature. Having function signatures contain the lifetime contract means the analysis the Rust compiler does can be simpler. If there's a problem with the way a function is annotated or the way it is called, the compiler errors can point to the part of our code and the constraints more precisely. If, instead, the Rust compiler made more inferences about what we intended the relationships of the lifetimes to be, the compiler might only be able to point to a use of our code many steps away from the cause of the problem.

When we pass concrete references to `longest`, the concrete lifetime that is substituted for `'a` is the part of the scope of `x` that overlaps with the scope of `y`. In other words, the generic lifetime `'a` will get the concrete lifetime that is equal to the smaller of the lifetimes of `x` and `y`. Because we've annotated the returned reference with the same lifetime parameter `'a`, the returned reference will also be valid for the length of the smaller of the lifetimes of `x` and `y`.

Let's look at how the lifetime annotations restrict the `longest` function by passing in references that have different concrete lifetimes. Listing 10-22 is a straightforward example.

Filename: src/main.rs

```
fn main() {
    let string1 = String::from("long string is long");

    {
        let string2 = String::from("xyz");
        let result = longest(string1.as_str(), string2.as_str());
        println!("The longest string is {result}");
    }
}
```

Listing 10-22: Using the `longest` function with references to `String` values that have different concrete lifetimes

In this example, `string1` is valid until the end of the outer scope, `string2` is valid until the end of the inner scope, and `result` references something that is valid until the end of the inner scope. Run this code and you'll see that the borrow checker approves; it will compile and print `The longest string is long string is long`.

Next, let's try an example that shows that the lifetime of the reference in `result` must be the smaller lifetime of the two arguments. We'll move the declaration of the `result` variable outside the inner scope but leave the assignment of the value to the `result` variable inside the scope with `string2`. Then we'll move the `println!` that uses `result` to outside the inner scope, after the inner scope has ended. The code in Listing 10-23 will not compile.

Filename: src/main.rs

```
fn main() {
    let string1 = String::from("long string is long");
    let result;
    {
        let string2 = String::from("xyz");
        result = longest(string1.as_str(), string2.as_str());
    }
    println!("The longest string is {result}");
}
```

Listing 10-23: Attempting to use `result` after `string2` has gone out of scope

When we try to compile this code, we get this error:

```
$ cargo run
   Compiling chapter10 v0.1.0 (file:///projects/chapter10)
error[E0597]: `string2` does not live long enough
 --> src/main.rs:6:44
  |
5 |         let string2 = String::from("xyz");
  |             ------- binding `string2` declared here
6 |         result = longest(string1.as_str(), string2.as_str());
  |                                            ^^^^^^^ borrowed value does not
live long enough
7 |     }
  |     - `string2` dropped here while still borrowed
8 |     println!("The longest string is {result}");
  |                                     -------- borrow later used here

For more information about this error, try `rustc --explain E0597`.
error: could not compile `chapter10` (bin "chapter10") due to 1 previous error
```

The error shows that for `result` to be valid for the `println!` statement, `string2` would need to be valid until the end of the outer scope. Rust knows this because we annotated the lifetimes of the function parameters and return values using the same lifetime parameter `'a`.

As humans, we can look at this code and see that `string1` is longer than `string2`, and therefore, `result` will contain a reference to `string1`. Because `string1` has not gone out of scope yet, a reference to `string1` will still be valid for the `println!` statement. However, the compiler can't see that the reference is valid in this case. We've told Rust that the lifetime of the reference returned by the `longest` function is the same as the smaller of the lifetimes of the references passed in. Therefore, the borrow checker disallows the code in Listing 10-23 as possibly having an invalid reference.

Try designing more experiments that vary the values and lifetimes of the references passed in to the `longest` function and how the returned reference is used. Make hypotheses about whether or not your experiments will pass the borrow checker before you compile; then check to see if you're right!

Thinking in Terms of Lifetimes

The way in which you need to specify lifetime parameters depends on what your function is doing. For example, if we changed the implementation of the `longest` function to always return the first parameter rather than the longest string slice, we wouldn't need to specify a lifetime on the `y` parameter. The following code will compile:

Filename: src/main.rs

```
fn longest<'a>(x: &'a str, y: &str) -> &'a str {
    x
}
```

We've specified a lifetime parameter `'a` for the parameter `x` and the return type, but not for the parameter `y`, because the lifetime of `y` does not have any relationship with the lifetime of `x` or the return value.

When returning a reference from a function, the lifetime parameter for the return type needs to match the lifetime parameter for one of the parameters. If the reference returned does *not* refer to one of the parameters, it must refer to a value created within this function. However, this would be a dangling reference because the value will go out of scope at the end of the function. Consider this attempted implementation of the `longest` function that won't compile:

Filename: src/main.rs

```
fn longest<'a>(x: &str, y: &str) -> &'a str {
    let result = String::from("really long string");
    result.as_str()
}
```

Here, even though we've specified a lifetime parameter `'a` for the return type, this implementation will fail to compile because the return value lifetime is not related to the lifetime of the parameters at all. Here is the error message we get:

```
$ cargo run
   Compiling chapter10 v0.1.0 (file:///projects/chapter10)
error[E0515]: cannot return value referencing local variable `result`
  --> src/main.rs:11:5
   |
11 |     result.as_str()
   |     ------^^^^^^^^^
   |     |
   |     returns a value referencing data owned by the current function
   |     `result` is borrowed here

For more information about this error, try `rustc --explain E0515`.
error: could not compile `chapter10` (bin "chapter10") due to 1 previous error
```

The problem is that `result` goes out of scope and gets cleaned up at the end of the `longest` function. We're also trying to return a reference to `result` from the function. There is no way we can specify lifetime parameters that would change the dangling reference, and Rust won't let us create a dangling reference. In this case, the best fix would be to return an owned data type rather than a reference so the calling function is then responsible for cleaning up the value.

Ultimately, lifetime syntax is about connecting the lifetimes of various parameters and return values of functions. Once they're connected, Rust has enough information to allow memory-safe operations and disallow operations that would create dangling pointers or otherwise violate memory safety.

Lifetime Annotations in Struct Definitions

So far, the structs we've defined all hold owned types. We can define structs to hold references, but in that case we would need to add a lifetime annotation on every reference in the struct's definition. Listing 10-24 has a struct named `ImportantExcerpt` that holds a string slice.

Filename: src/main.rs

```
struct ImportantExcerpt<'a> {
    part: &'a str,
}

fn main() {
    let novel = String::from("Call me Ishmael. Some years ago...");
    let first_sentence = novel.split('.').next().unwrap();
    let i = ImportantExcerpt {
        part: first_sentence,
    };
}
```

Listing 10-24: A struct that holds a reference, requiring a lifetime annotation

This struct has the single field `part` that holds a string slice, which is a reference. As with generic data types, we declare the name of the generic lifetime parameter inside angle brackets after the name of the struct so we can use the lifetime parameter in the body of the struct definition. This annotation means an instance of `ImportantExcerpt` can't outlive the reference it holds in its `part` field.

The `main` function here creates an instance of the `ImportantExcerpt` struct that holds a reference to the first sentence of the `String` owned by the variable `novel`. The data in `novel` exists before the `ImportantExcerpt` instance is created. In addition, `novel` doesn't go out of scope until after the `ImportantExcerpt` goes out of scope, so the reference in the `ImportantExcerpt` instance is valid.

Lifetime Elision

You've learned that every reference has a lifetime and that you need to specify lifetime parameters for functions or structs that use references. However, we had a function in Listing

4-9, shown again in Listing 10-25, that compiled without lifetime annotations.

Filename: src/lib.rs

```
fn first_word(s: &str) -> &str {
    let bytes = s.as_bytes();

    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return &s[0..i];
        }
    }

    &s[..]
}
```

Listing 10-25: A function we defined in Listing 4-9 that compiled without lifetime annotations, even though the parameter and return type are references

The reason this function compiles without lifetime annotations is historical: in early versions (pre-1.0) of Rust, this code wouldn't have compiled because every reference needed an explicit lifetime. At that time, the function signature would have been written like this:

```
fn first_word<'a>(s: &'a str) -> &'a str {
```

After writing a lot of Rust code, the Rust team found that Rust programmers were entering the same lifetime annotations over and over in particular situations. These situations were predictable and followed a few deterministic patterns. The developers programmed these patterns into the compiler's code so the borrow checker could infer the lifetimes in these situations and wouldn't need explicit annotations.

This piece of Rust history is relevant because it's possible that more deterministic patterns will emerge and be added to the compiler. In the future, even fewer lifetime annotations might be required.

The patterns programmed into Rust's analysis of references are called the *lifetime elision rules*. These aren't rules for programmers to follow; they're a set of particular cases that the compiler will consider, and if your code fits these cases, you don't need to write the lifetimes explicitly.

The elision rules don't provide full inference. If there is still ambiguity as to what lifetimes the references have after Rust applies the rules, the compiler won't guess what the lifetime of the remaining references should be. Instead of guessing, the compiler will give you an error that you can resolve by adding the lifetime annotations.

Lifetimes on function or method parameters are called *input lifetimes*, and lifetimes on return values are called *output lifetimes*.

The compiler uses three rules to figure out the lifetimes of the references when there aren't explicit annotations. The first rule applies to input lifetimes, and the second and third rules apply to output lifetimes. If the compiler gets to the end of the three rules and there are still references for which it can't figure out lifetimes, the compiler will stop with an error. These rules apply to `fn` definitions as well as `impl` blocks.

The first rule is that the compiler assigns a lifetime parameter to each parameter that's a reference. In other words, a function with one parameter gets one lifetime parameter: `fn foo<'a>(x: &'a i32)`; a function with two parameters gets two separate lifetime parameters: `fn foo<'a, 'b>(x: &'a i32, y: &'b i32)`; and so on.

The second rule is that, if there is exactly one input lifetime parameter, that lifetime is assigned to all output lifetime parameters: `fn foo<'a>(x: &'a i32) -> &'a i32`.

The third rule is that, if there are multiple input lifetime parameters, but one of them is `&self` or `&mut self` because this is a method, the lifetime of `self` is assigned to all output lifetime parameters. This third rule makes methods much nicer to read and write because fewer symbols are necessary.

Let's pretend we're the compiler. We'll apply these rules to figure out the lifetimes of the references in the signature of the `first_word` function in Listing 10-25. The signature starts without any lifetimes associated with the references:

```
fn first_word(s: &str) -> &str {
```

Then the compiler applies the first rule, which specifies that each parameter gets its own lifetime. We'll call it `'a` as usual, so now the signature is this:

```
fn first_word<'a>(s: &'a str) -> &str {
```

The second rule applies because there is exactly one input lifetime. The second rule specifies that the lifetime of the one input parameter gets assigned to the output lifetime, so the signature is now this:

```
fn first_word<'a>(s: &'a str) -> &'a str {
```

Now all the references in this function signature have lifetimes, and the compiler can continue its analysis without needing the programmer to annotate the lifetimes in this function signature.

Let's look at another example, this time using the `longest` function that had no lifetime parameters when we started working with it in Listing 10-20:

```
fn longest(x: &str, y: &str) -> &str {
```

Let's apply the first rule: each parameter gets its own lifetime. This time we have two parameters instead of one, so we have two lifetimes:

```
fn longest<'a, 'b>(x: &'a str, y: &'b str) -> &str {
```

You can see that the second rule doesn't apply because there is more than one input lifetime. The third rule doesn't apply either, because `longest` is a function rather than a method, so none of the parameters are `self`. After working through all three rules, we still haven't figured out what the return type's lifetime is. This is why we got an error trying to compile the code in Listing 10-20: the compiler worked through the lifetime elision rules but still couldn't figure out all the lifetimes of the references in the signature.

Because the third rule really only applies in method signatures, we'll look at lifetimes in that context next to see why the third rule means we don't have to annotate lifetimes in method signatures very often.

Lifetime Annotations in Method Definitions

When we implement methods on a struct with lifetimes, we use the same syntax as that of generic type parameters shown in Listing 10-11. Where we declare and use the lifetime parameters depends on whether they're related to the struct fields or the method parameters and return values.

Lifetime names for struct fields always need to be declared after the `impl` keyword and then used after the struct's name because those lifetimes are part of the struct's type.

In method signatures inside the `impl` block, references might be tied to the lifetime of references in the struct's fields, or they might be independent. In addition, the lifetime elision rules often make it so that lifetime annotations aren't necessary in method signatures. Let's look at some examples using the struct named `ImportantExcerpt` that we defined in Listing 10-24.

First we'll use a method named `level` whose only parameter is a reference to `self` and whose return value is an `i32`, which is not a reference to anything:

```
impl<'a> ImportantExcerpt<'a> {
    fn level(&self) -> i32 {
        3
    }
}
```

The lifetime parameter declaration after `impl` and its use after the type name are required, but we're not required to annotate the lifetime of the reference to `self` because of the first elision rule.

Here is an example where the third lifetime elision rule applies:

```
impl<'a> ImportantExcerpt<'a> {
    fn announce_and_return_part(&self, announcement: &str) -> &str {
        println!("Attention please: {announcement}");
        self.part
    }
}
```

There are two input lifetimes, so Rust applies the first lifetime elision rule and gives both `&self` and `announcement` their own lifetimes. Then, because one of the parameters is `&self`, the return type gets the lifetime of `&self`, and all lifetimes have been accounted for.

The Static Lifetime

One special lifetime we need to discuss is `'static`, which denotes that the affected reference *can* live for the entire duration of the program. All string literals have the `'static` lifetime, which we can annotate as follows:

```
let s: &'static str = "I have a static lifetime.";
```

The text of this string is stored directly in the program's binary, which is always available. Therefore, the lifetime of all string literals is `'static`.

You might see suggestions to use the `'static` lifetime in error messages. But before specifying `'static` as the lifetime for a reference, think about whether the reference you have actually lives the entire lifetime of your program or not, and whether you want it to. Most of the time, an error message suggesting the `'static` lifetime results from attempting to create a dangling reference or a mismatch of the available lifetimes. In such cases, the solution is to fix those problems, not to specify the `'static` lifetime.

Generic Type Parameters, Trait Bounds, and Lifetimes Together

Let's briefly look at the syntax of specifying generic type parameters, trait bounds, and lifetimes all in one function!

```
use std::fmt::Display;

fn longest_with_an_announcement<'a, T>(
    x: &'a str,
    y: &'a str,
    ann: T,
) -> &'a str
where
    T: Display,
{
    println!("Announcement! {ann}");
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

This is the `longest` function from Listing 10-21 that returns the longer of two string slices. But now it has an extra parameter named `ann` of the generic type `T`, which can be filled in by any type that implements the `Display` trait as specified by the `where` clause. This extra parameter will be printed using `{}`, which is why the `Display` trait bound is necessary. Because lifetimes are a type of generic, the declarations of the lifetime parameter `'a` and the generic type parameter `T` go in the same list inside the angle brackets after the function name.

Summary

We covered a lot in this chapter! Now that you know about generic type parameters, traits and trait bounds, and generic lifetime parameters, you're ready to write code without repetition that works in many different situations. Generic type parameters let you apply the code to different types. Traits and trait bounds ensure that even though the types are generic, they'll have the behavior the code needs. You learned how to use lifetime annotations to ensure that this flexible code won't have any dangling references. And all of this analysis happens at compile time, which doesn't affect runtime performance!

Believe it or not, there is much more to learn on the topics we discussed in this chapter: Chapter 17 discusses trait objects, which are another way to use traits. There are also more complex scenarios involving lifetime annotations that you will only need in very advanced scenarios; for those, you should read the Rust Reference. But next, you'll learn how to write tests in Rust so you can make sure your code is working the way it should.