[Removing Duplication by Extracting a Function 2](#_Toc508292470)

[Generic Data Types 5](#_Toc508292471)

[In Function Definitions 5](#_Toc508292472)

[In Struct Definitions 9](#_Toc508292473)

[In Enum Definitions 11](#_Toc508292474)

[In Method Definitions 12](#_Toc508292475)

[Performance of Code Using Generics 14](#_Toc508292476)

[Traits: Defining Shared Behavior 16](#_Toc508292477)

[Defining a Trait 16](#_Toc508292478)

[Implementing a Trait on a Type 17](#_Toc508292479)

[Default Implementations 19](#_Toc508292480)

[Trait Bounds 21](#_Toc508292481)

[Fixing the largest Function with Trait Bounds 22](#_Toc508292482)

[Using Trait Bounds to Conditionally Implement Methods 25](#_Toc508292483)

[Validating References with Lifetimes 27](#_Toc508292484)

[Lifetimes Prevent Dangling References 27](#_Toc508292485)

[The Borrow Checker 28](#_Toc508292486)

[Generic Lifetimes in Functions 30](#_Toc508292487)

[Lifetime Annotation Syntax 31](#_Toc508292488)

[Lifetime Annotations in Function Signatures 32](#_Toc508292489)

[Thinking in Terms of Lifetimes 35](#_Toc508292490)

[Lifetime Annotations in Struct Definitions 37](#_Toc508292491)

[Lifetime Elision 38](#_Toc508292492)

[Lifetime Annotations in Method Definitions 41](#_Toc508292493)

[The Static Lifetime 42](#_Toc508292494)

[Generic Type Parameters, Trait Bounds, and Lifetimes Together 42](#_Toc508292495)

[Summary 43](#_Toc508292496)

10

Generic Types, Traits, and Lifetimes

Every programming language has tools for effectively handling the duplication of concepts. In Rust, one such tool is generics. Generics are abstract stand-ins for concrete types or other properties. When we’re writing code, 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.

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

prod: check xrefs

First, we’ll review how to extract a function to reduce code duplication. Next, we’ll use the same technique to make a generic function from two functions that only differ 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 then combine traits with generic types to constrain a generic type to 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 borrow values in many situations while still enabling the compiler to check that the references are valid.

Removing Duplication by Extracting a Function

Before diving into generics syntax, let’s first look at how to remove duplication that doesn’t involve generic types by extracting a function. Then we’ll apply this technique to extract a generic function! In the same way that you recognize duplicated code to extract into a function, you’ll start to recognize duplicated code that can use generics.

Consider a short program that finds the largest number in a list, as shown in Listing 10-1:

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: Code to find the largest number in a list of numbers

This code stores a list of integers in the variable number\_list  and places the first number in the list in a variable named largest . Then it iterates through all the numbers in the list , and if the current number is greater than the number stored in largest , it replaces the number in that variable . However, if the current number is less than 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 hold the largest number, which in this case is 100.

To find the largest number in two different lists of numbers, we can 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:

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 update the code in multiple places to change it.

To eliminate this duplication, we can create an abstraction by defining a function that operates on any list of integers given to it 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 extracted the code that finds the largest number into a function named largest. Unlike the code in Listing 10-1, which can find the largest number in only one particular list, this program can find the largest number in two different lists:

src/main.rs

fn largest(list: &[i32]) -> i32 {

let mut largest = list[0];

for &item in list.iter() {

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

Identify duplicate code.

Extract the duplicate code into the body of the function, and specify the inputs and return values of that code in the function signature.

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 different ways. 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 can 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:

src/main.rs

fn largest\_i32(list: &[i32]) -> i32 {

let mut largest = list[0];

for &item in list.iter() {

if item > largest {

largest = item;

}

}

largest

}

fn largest\_char(list: &[char]) -> char {

let mut largest = list[0];

for &item in list.iter() {

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 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 the new function we’ll define, we need to name the type parameter, just like 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, parameter names in Rust are short, often just a letter, and Rust’s type naming convention is CamelCase. Short for “type,” T is the default choice of most Rust programmers.

When we use a parameter in the body of the function, we have to declare the parameter name in the signature so that 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, place type name declarations inside angle brackets (<>) between the name of the function and the parameter list, like this:

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 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.

src/main.rs

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

let mut largest = list[0];

for &item in list.iter() {

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: A definition of the largest function that uses generic type parameters but doesn’t compile yet

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

error[E0369]: binary operation `>` cannot be applied to type `T`

--> src/main.rs:5:12

|

5 | if item > largest {

| ^^^^^^^^^^^^^^

|

= note: an implementation of `std::cmp::PartialOrd` might be missing for `T`

The note mentions std::cmp::PartialOrd, which is a trait. We’ll talk about traits in the next section. For now, 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, “Derivable Traits,” for more on this trait). You’ll learn how to specify that a generic type has a particular trait in the “Trait Bounds” section on page XX, but let’s first explore other ways of using generic type parameters.

Prod: Fill Xref

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 shows how to define a Point<T> struct to hold x and y coordinate values of any type:

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 can use the generic type in the struct definition where we would otherwise specify concrete data types .

Note that because we’ve only used one generic type to define Point<T>, this 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. This means that 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:

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:

error[E0308]: mismatched types

--> src/main.rs:7:38

|

7 | let wont\_work = Point { x: 5, y: 4.0 };

| ^^^ expected integral variable, found floating-point variable

|

= note: expected type `{integer}`

found type `{float}`

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 can 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:

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 };

}

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. When 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 that we used in Chapter 6:

prod: check xref

enum Option<T> {

Some(T),

None,

}

This definition should now make more sense to you. As you can see, Option<T> is an enum that 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 having 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:

prod: check xref

enum Result<T, E> {

Ok(T),

Err(E),

}

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.

prod: check xref

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

As we did in Chapter 5, we can implement methods on structs and enums that have generic types in their definitions. Listing 10-9 shows the Point<T> struct we defined in Listing 10-6 with a method named x implemented on it:

prod: check xref

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 it 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, 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:

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 method named distance\_from\_origin, and 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 only available for floating point types.

Generic type parameters in a struct definition aren’t always the same as those you use in that struct’s method signatures. For example, Listing 10-11 defines the method mixup on the Point<T, U> struct from Listing 10-8. The method takes another Point as a parameter, which might have different types than the self Point we’re calling mixup on. The method creates a new Point instance with the x value from the self Point (of type T) and the y value from the passed-in Point (of type W):

src/main.rs

struct Point<T, U> {

x: T,

y: U,

}

impl<T, U> Point<T, U> {

fn mixup<V, W>(self, other: Point<V, W>) -> Point<T, W> {

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: Methods that use different generic types than their 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 T and U are declared after impl, because they go with the struct definition. The generic parameters V and W 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 you’re using generic type parameters. The good news is that Rust implements generics in such a way that your code doesn’t run any slower using generic types than it would with concrete types.

Rust accomplishes this by performing monomorphization of the code that is 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 with an example that uses the standard library’s 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 the instances of Option<T> 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 Option\_i32 and Option\_f64, thereby replacing the generic definition with the specific ones.

The monomorphized version of the code looks like the following. The generic Option<T> is replaced with the specific definitions created by the compiler:

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);

}

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 like it would if we had duplicated each definition by hand. The process of monomorphization makes Rust’s generics extremely efficient at runtime.

Traits: Defining Shared Behavior

A trait tells the Rust compiler about 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 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 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 need to request that summary by calling a summarize method on an instance. Listing 10-12 shows the definition of a Summary trait that expresses this behavior:

src/lib.rs

pub trait Summary {

fn summarize(&self) -> String;

}

Listing 10-12: Definition of 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. 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 behavior using the Summary trait, 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 tweet content is already limited to 280 characters.

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 that 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.

After implementing the trait, we can call the methods on instances of NewsArticle and Tweet in the same way we call regular methods, like this:

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.

Note that because we defined the Summary trait and the NewsArticle and Tweet types in the same lib.rs in Listing 10-13, they’re all in the same scope. Let’s say this lib.rs is for a crate we’ve called aggregator, and someone else wants to use our crate’s functionality to implement the Summary trait on a struct defined within their library’s scope. They would need to import the trait into their scope first. They would do so by specifying use aggregator::Summary;, which then enables them to implement Summary for their type. The Summary trait would also need to be a public trait for another crate to implement it, which it is because we put the pub keyword before trait in Listing 10-12.

One restriction to note with trait implementations is that we can implement a trait on a type only if either the trait or the type is local to your 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 defined in the standard library and aren’t local to our aggregator crate. This restriction is part of a property of programs 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.

Listing 10-14 shows how to specify a default string for the summarize method of the Summary trait instead of only defining the method signature, like we did in Listing 10-12:

src/lib.rs

pub trait Summary {

fn summarize(&self) -> String {

String::from("(Read more...)")

}

}

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

To use a default implementation to summarize instances of NewsArticle instead of defining a custom implementation, 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 for summarize 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.

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.

Trait Bounds

Now that you know how to define traits and implement those traits on types, we can explore how to use traits with generic type parameters. We can use trait bounds to constrain generic types to ensure the type will be limited to those that implement a particular trait and behavior.

For example, in Listing 10-13, we implemented the Summary trait on the types NewsArticle and Tweet. We can define a function notify that calls the summarize method on its parameter item, which is of the generic type T. To be able to call summarize on item without getting an error telling us that the generic type T doesn’t implement the method summarize, we can use trait bounds on T to specify that item must be of a type that implements the Summary trait:

pub fn notify<T: Summary>(item: T) {

println!("Breaking news! {}", item.summarize());

}

We place trait bounds with the declaration of the generic type parameter, after a colon and inside angle brackets. Because of the trait bound on T, we can call notify and pass in any instance of NewsArticle or Tweet. Code that calls the function with any other type, like a String or an i32, won’t compile, because those types don’t implement Summary.

We can specify multiple trait bounds on a generic type using the + syntax. For example, to use display formatting on the type T in a function as well as the summarize method, we can use T: Summary + Display to say T can be any type that implements Summary and Display.

However, there are downsides to using too many trait bounds. Each generic has its own trait bounds; so functions with multiple generic type parameters can have lots of trait bound information between a 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 in that the function name, parameter list, and return type are close together, similar to a function without lots of trait bounds.

Fixing the largest Function with Trait Bounds

Now that you know how to specify the behavior you want to use using the generic type parameter’s bounds, let’s return to Listing 10-5 to fix the definition of the largest function that uses a generic type parameter! Last time we tried to run that code, we received this error:

error[E0369]: binary operation `>` cannot be applied to type `T`

--> src/main.rs:5:12

|

5 | if item > largest {

| ^^^^^^^^^^^^^^

|

= note: an implementation of `std::cmp::PartialOrd` might be missing for `T`

In the body of largest we wanted to compare two values of type T using the greater-than (>) operator. Because that operator is defined as a default method on the standard library trait std::cmp::PartialOrd, we need to specify PartialOrd in the trait bounds for T so the largest function can work on slices of any type that we can compare. We don’t need to bring PartialOrd into scope because it’s in the prelude. Change the signature of largest to look like this:

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

This time when we compile the code, we get a different set of errors:

error[E0508]: cannot move out of type `[T]`, a non-copy slice

--> src/main.rs:2:23

|

2 | let mut largest = list[0];

| ^^^^^^^

| |

| cannot move out of here

| help: consider using a reference instead: `&list[0]`

error[E0507]: cannot move out of borrowed content

--> src/main.rs:4:9

|

4 | for &item in list.iter() {

| ^----

| ||

| |hint: to prevent move, use `ref item` or `ref mut item`

| cannot move out of borrowed content

The key line in this error is cannot move out of type [T], a non-copy slice. With our non-generic versions of the largest function, we were only trying to find the largest i32 or char. As discussed in “Stack-Only Data: Copy” on page XX, types like i32 and char that have a known size can be stored on the stack, so they implement the Copy trait. But when we made the largest function generic, the list parameter could have types in it that don’t implement the Copy trait. Consequently, we wouldn’t be able to move the value out of list[0] and into the largest variable, resulting in this error.

prod: check/fill xref (ch4)

To call this code with only those types that implement the Copy trait, we can add Copy to the trait bounds of T! Listing 10-15 shows the complete code of a generic largest function that will compile as long as the types of the values in the slice that we pass into the function implement the PartialOrd and Copy traits, like i32 and char do:

src/main.rs

fn largest<T: PartialOrd + Copy>(list: &[T]) -> T {

let mut largest = list[0];

for &item in list.iter() {

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 10-15: A working definition of the largest function that works on any generic type that implements the PartialOrd and Copy traits

If we don’t want to restrict the largest function to the types that implement the Copy trait, we could specify that T has the trait bound Clone instead of Copy. Then we could clone each value in the slice when we want the largest function to have ownership. Using the clone function means we’re potentially making more heap allocations in the case of types that own heap data like String, and heap allocations can be slow if we’re working with large amounts of data.

Another way we could implement largest is for the function to return a reference to a T value in the slice. If we change the return type to &T instead of T, thereby changing the body of the function to return a reference, we wouldn’t need the Clone or Copy trait bounds and we could avoid heap allocations. Try implementing these alternate solutions on your own!

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 10-16 always implements the new function. But 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:

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 10-16: Conditionally implement 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 extensively used 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 that the type didn’t implement. 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.

Another kind of generic that we’ve already been using is called lifetimes. 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. Let’s look at how lifetimes do that.

Validating References with Lifetimes

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 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.

prod: check xref

The concept of lifetimes is somewhat different from tools in other programming languages, arguably making lifetimes Rust’s most distinctive feature. 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 become familiar with the concepts. See “Advanced Lifetimes” on page XX for more detailed information.

prod: check/fill xref (ch 19)

Lifetimes Prevent Dangling References

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 10-17, which has an outer scope and an inner scope:

{

 let r;

{

 let x = 5;

 r = &x;

 }

 println!("r: {}", r);

}

Listing 10-17: An attempt to use a reference whose value has gone out of scope

Note The example in Listing 10-17 and the next few examples 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 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 r is referring to has gone out of scope before we try to use it. Here is the error message:

error[E0597]: `x` does not live long enough

--> src/main.rs:7:5

|

6 | r = &x;

| - borrow occurs here

7 | }

| ^ `x` dropped here while still borrowed

...

10 | }

| - borrowed value needs to live until here

The variable x doesn’t “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 that all borrows are valid. Listing 10-18 shows the same code as Listing 10-17 but with annotations showing the lifetimes of the variables:

{

let r; // ---------+-- 'a

// |

{ // |

let x = 5; // -+-- 'b |

r = &x; // | |

} // -+ |

// |

println!("r: {}", r); // |

} // ---------+

Listing 10-18: 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 10-19 fixes the code so it doesn’t have a dangling reference and compiles without any errors:

{

let x = 5; // ----------+-- 'b

// |

let r = &x; // --+-- 'a |

// | |

println!("r: {}", r); // | |

// --+ |

} // ----------+

Listing 10-19: 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 where 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

Let’s write a function that returns the longer of two string slices. This function will take two string slices and return a string slice. After we’ve implemented the longest function, the code in Listing 10-20 should print The longest string is abcd:

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 10-20: 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, because we don’t want the longest function to take ownership of its parameters. We want to allow the function to accept slices of a String (the type stored in the variable string1) as well as string literals (which is what variable string2 contains).

Refer to the “String Slices as Parameters” section in Chapter 4 for more discussion about why the parameters we use in Listing 10-20 are the ones we want.

prod: check xref

If we try to implement the longest function as shown in Listing 10-21, it won’t compile:

src/main.rs

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

if x.len() > y.len() {

x

} else {

y

}

}

Listing 10-21: 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:

error[E0106]: missing lifetime specifier

--> src/main.rs:1:33

|

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

| ^ expected 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`

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 like we did in Listings 10-18 and 10-19 to determine that 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. Just like 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 describe the relationships of the lifetimes of multiple references to each other without affecting the lifetimes.

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. 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. For example, let’s say we have a function with the parameter first that is a reference to an i32 with lifetime 'a. The function also has another parameter named second that is another reference to an i32 that also has the lifetime 'a. The lifetime annotations indicate that the references first and second must both live as long as that generic lifetime.

Lifetime Annotations in Function Signatures

Now let’s examine lifetime annotations in the context of the longest function. As with generic type parameters, we need to declare generic lifetime parameters inside angle brackets between the function name and the parameter list. The constraint we want to express in this signature is that all the references in the parameters and the return value must have the same lifetime. We’ll name the lifetime 'a, and then add it to each reference, as shown in Listing 10-22:

src/main.rs

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

if x.len() > y.len() {

x

} else {

y

}

}

Listing 10-22: 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 10-20.

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. These constraints are what we want Rust to enforce.

As discussed earlier, by specifying the lifetime parameters in this function signature, we’re not changing the lifetimes of any values passed in or returned. Instead, 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. Rust can analyze the code within the function without any help. However, when a function has references to or from code outside that function, it becomes almost impossible for Rust to figure out the lifetimes of the parameters or return values on its own. The lifetimes might be different each time the function is called. This is why we need to annotate the lifetimes manually.

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 10-23 is a straightforward example:

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 10-23: 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 of this code; 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 outside the inner scope, after the inner scope has ended. The code in Listing 10-24 will not compile:

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 10-24: Attempting to use result after string2 has gone out of scope; the code won’t compile

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

error[E0597]: `string2` does not live long enough

--> src/main.rs:15:5

|

14 | result = longest(string1.as\_str(), string2.as\_str());

| ------- borrow occurs here

15 | }

| ^ `string2` dropped here while still borrowed

16 | println!("The longest string is {}", result);

17 | }

| - borrowed value needs to live until here

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 10-24 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:

src/main.rs

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

x

}

In this example, 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, which 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:

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:

error[E0597]: `result` does not live long enough

--> src/main.rs:3:5

|

3 | result.as\_str()

| ^^^^^^ does not live long enough

4 | }

| - borrowed value only lives until here

|

note: borrowed value must be valid for the lifetime 'a as defined on the

function body at 1:1...

--> src/main.rs:1:1

|

1 | / fn longest<'a>(x: &str, y: &str) -> &'a str {

2 | | let result = String::from("really long string");

3 | | result.as\_str()

4 | | }

| |\_^

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, we’ve only defined structs to hold owned types. It’s possible for 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 10-25 has a struct named ImportantExcerpt that holds a string slice:

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()

.expect("Could not find a '.'");

 let i = ImportantExcerpt { part: first\_sentence };

}

Listing 10-25: A struct that holds a reference, so its definition needs a lifetime annotation

This struct has one 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, in Chapter 4 we had a function in “String Slices” on page XX, which is shown again in Listing 10-26, that compiled without lifetime annotations:

prod: check/fill xref (ch4)

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 10-26: A function we defined in Chapter 4 that compiled without lifetime annotations, even though the parameter and return type are references

prod: check xref

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 repeatedly 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 not 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 Rust deterministically applies the rules but there is still ambiguity as to what lifetimes the references have, the compiler won’t guess what the lifetime of the remaining references should be. In this case, instead of guessing, the compiler will give you an error that you can resolve by adding the lifetime annotations that specify how the references relate to each other.

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 what lifetimes references have 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.

The first rule is that each parameter that is a reference gets its own lifetime parameter. 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 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 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 what the lifetimes of the references in the signature of the first\_word function in Listing 10-26 are. 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:

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 10-21:

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 10-21: 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 10-25.

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 use after the type name is 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 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 binary of your program, 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. You might consider whether you want it to live that long, even if it could. Most of the time, the problem results from attempting to create a dangling reference or a mismatch of the available lifetimes. In such cases, the solution is fixing those problems, not specifying 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 10-22 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 before the function compares the lengths of the string slices, 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 yet 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. Chapter 19 covers more complex scenarios involving lifetime annotations as well as some advanced type system features. But in the next chapter, you’ll learn how to write tests in Rust so you can make sure your code is working the way it should.

prod: check xrefs