[Removing Duplication by Extracting a Function 2](#_Toc506718461)

[Generic Data Types 5](#_Toc506718462)

[In Function Definitions 5](#_Toc506718463)

[In Struct Definitions 8](#_Toc506718464)

[In Enum Definitions 11](#_Toc506718465)

[In Method Definitions 11](#_Toc506718466)

[Performance of Code Using Generics 13](#_Toc506718467)

[Traits: Defining Shared Behavior 15](#_Toc506718468)

[Defining a Trait 15](#_Toc506718469)

[Implementing a Trait on a Type 16](#_Toc506718470)

[Default Implementations 19](#_Toc506718471)

[Trait Bounds 21](#_Toc506718472)

[Fixing the largest Function with Trait Bounds 22](#_Toc506718473)

[Validating References with Lifetimes 25](#_Toc506718474)

[Lifetimes Prevent Dangling References 25](#_Toc506718475)

[Uninitialized Variables Cannot Be Used 26](#_Toc506718476)

[The Borrow Checker 27](#_Toc506718477)

[Generic Lifetimes in Functions 28](#_Toc506718478)

[Lifetime Annotation Syntax 29](#_Toc506718479)

[Lifetime Annotations in Function Signatures 30](#_Toc506718480)

[Thinking in Terms of Lifetimes 33](#_Toc506718481)

[Lifetime Annotations in Struct Definitions 35](#_Toc506718482)

[Lifetime Elision 35](#_Toc506718483)

[Lifetime Annotations in Method Definitions 38](#_Toc506718484)

[The Static Lifetime 39](#_Toc506718485)

[Generic Type Parameters, Trait Bounds, and Lifetimes Together 40](#_Toc506718486)

[Summary 40](#_Toc506718487)

Chapter 10

Generic Types, Traits, and Lifetimes

Every programming language has tools for dealing with duplication of concepts. In Rust, one such tool is generics, which we can use as abstract stand-ins for concrete types or other properties. For example, generics lets us express their behavior or how they relate to other generics without needing to know what will actually be in their place when writing and compiling 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: confirm xrefs

First, we’ll review how to extract a function to reduce code duplication. Then we’ll use the same technique to make a generic function out of two functions that only differ in the types of their parameters. We’ll go over how to use generic types in struct and enum definitions too.

After that, 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 generic that gives the compiler information about how references are related 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 review 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 small program that finds the largest number in a list, as shown in Listing 10-1:

Filename: src/main.rs

fn main() {

let numbers = vec![34, 50, 25, 100, 65];

let mut largest = numbers[0];

for number in numbers {

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 numbers and places the first item in the list in a variable named largest. Then it iterates through all the numbers in the list, and if the current value is greater than the number stored in largest, it replaces the value in that variable. If the current value is smaller than the largest value seen so far, however, the variable doesn’t change and the code moves on to the next item in the list. After all the items in the list have been considered, largest should hold the largest value, 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 in Listing 10-2:

Filename: src/main.rs

fn main() {

let numbers = vec![34, 50, 25, 100, 65];

let mut largest = numbers[0];

for number in numbers {

if number > largest {

largest = number;

}

}

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

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

let mut largest = numbers[0];

for number in numbers {

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

While this code works, duplicating code is tedious and error-prone. It also means that we 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 makes our code clear and lets us express the concept of finding the largest number in a list abstractly.

In Listing 10-3, we’ve 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:

Filename: 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 numbers = vec![34, 50, 25, 100, 65];

let result = largest(&numbers);

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

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

let result = largest(&numbers);

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. This means that when we call the function, the code runs on the specific values that we pass in.

In sum, here are the steps we used to get 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 get rid of that duplication? Let’s find out!

Generic Data Types

We can use generics to create definitions for many different concrete data types, like in function signatures or structs. 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. This 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:

Filename: 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 numbers = vec![34, 50, 25, 100, 65];

let result = largest\_i32(&numbers);

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

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

let result = largest\_char(&chars);

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. Both functions have the exact same code, so let’s get rid of the duplication by introducing a generic type parameter in a single function.

To parameterize the types in the new function we’re going to 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 using T because by convention parameter names in Rust tend to be 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 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, and 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 shortly!

Filename: 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 numbers = vec![34, 50, 25, 100, 65];

let result = largest(&numbers);

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

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

let result = largest(&chars);

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`

|

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’re going to talk about traits in the next section. For now, this error is saying 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 D for more on this trait). You’ll learn how to specify that a generic type has a particular trait in the XXX section, but let’s first explore other ways of using generic type parameters.

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 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 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, this says that the Point 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 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. Then when we specify 4.0 for y, which we’ve defined to have the same type as x, we should get a type mismatch error, like this:

error[E0308]: mismatched types

-->

|

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

| ^^^ expected integral variable, found

floating-point variable

|

= note: expected type `{integer}`

= note: 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:

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 };

}

Listing 10-8: A Point generic over two types so that x and y may be values of different types

Now all instances of Point 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 find yourself needing lots of generic types, it may indicate that your code needs restructuring into smaller pieces.

In Enum Definitions

Like we did with structs, we can use enums to hold generic data types in their variants. Let’s take another look at the Option<T> enum provided by the standard library we used in Chapter 6:

enum Option<T> {

Some(T),

None,

}

This definition should now make more sense to you. As you can see, Option<T> is an enum generic of type T, which has two variants: Some, that holds one value of type T, and a None variant that doesn’t hold any value. This definition lets us use the more abstract concept of “an optional value” to create values of the Option enum that have any data type.

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),

}

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 9-2 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 iusing generic types in function definitions instead.

In Method Definitions

Like 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.

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 that we can use it to specify that we’re implementing methods on the type Point<T>.

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 10-10 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):

Filename: 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 10-10: 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, since x came from p1. The p3 variable will have a char for y, since y came from p2. The println! macro will print p3.x = 5, p3.y = c.

Note that the generic parameters T and U are declared after impl, since they go with the struct definition. The generic parameters V and W are declared after fn mixup, since they are only relevant to the method.

Performance of Code Using Generics

You may be wondering whether there’s a run-time cost to 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's using generics at compile time. Monomorphization is the process of turning generic code into specific code by filling in the concrete types that are actually used when compiled.

In this process the compiler it doing the opposite of the steps we performed to create the generic function in Listing 10-5: the compiler looks at all the places that generic code is called and generates code for the concrete types the generic code is called with.

Let’s see how this works with an example that uses the standard library’s Option 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 passed to Option and sees that we have two kinds of Option<T>: one is i32, and one 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 this, with the generic Option replaced with the specific definitions created by the compiler:

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);

}

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 particular definition by hand. As you can see, 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 state that a generic can be any type that has certain behavior: In situations where we use generic type parameters, we can use trait bounds to specify, at compile time, that the generic type may be any type that implements a particular trait and therefore has the behavior we want to use in that situation.

Note: Traits are similar to a feature often called ‘interfaces’ in other languages, though 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 in order 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 140 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 each struct to be summarizable, and we need to be able to ask for that summary by calling a summary method on an instance. Listing 10-11 shows the definition of a Summarizable trait that expresses this behavior:

Filename: lib.rs

pub trait Summarizable {

fn summary(&self) -> String;

}

Listing 10-11: Definition of a Summarizable trait that consists of the behavior provided by a summary method

Here, we declare a trait with the trait keyword, and then the trait’s name, which is Summarizable in this case. Inside the curly braces we declare the method signatures that describe the behaviors of the types that implement this trait , which in this case is fn summary(&self) -> String.

After the method signature, instead of providing an implementation within curly braces, we put a semicolon. This means that each type implementing this trait must provide its own custom behavior for the body of the method, but the compiler will enforce that any type that has the Summarizable trait will have the method summary defined with this signature exactly.

In short, a trait can have multiple methods in its body, with the method signatures listed one per line and each line ending in a semicolon.

Implementing a Trait on a Type

Now that we’ve defined our desired behavior using the Summarizable trait, we can implement it on the types in our media aggregator. Listing 10-12 shows an implementation of the Summarizable trait on the NewsArticle struct that uses the headline, the author, and the location to create the return value of summary. For the Tweet struct, we define summary as the username followed by the whole text of the tweet, assuming that tweet content is already limited to 140 characters.

Filename: lib.rs

pub struct NewsArticle {

pub headline: String,

pub location: String,

pub author: String,

pub content: String,

}

impl Summarizable for NewsArticle {

fn summary(&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 Summarizable for Tweet {

fn summary(&self) -> String {

format!("{}: {}", self.username, self.content)

}

}

Listing 10-12: Implementing the Summarizable 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 for to 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, but instead of adding a semicolon after each signature, we put curly braces 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 manner that 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.summary());

This should print one new tweet: horse\_ebooks: of course, as you probably already know, people.

Note that because we defined the Summarizable trait and the NewsArticle and Tweet types all in the same lib.rs in Listing 10-12, they’re all in the same scope. But if this lib.rs is for a crate we’ve called aggregator, and someone else wants to use our crate’s functionality to implement the Summarizable trait on their WeatherForecast struct, they would need to import the trait into their scope first before they can implement it, as shown in Listing 10-13:

Filename: lib.rs

extern crate aggregator;

use aggregator::Summarizable;

struct WeatherForecast {

high\_temp: f64,

low\_temp: f64,

chance\_of\_precipitation: f64,

}

impl Summarizable for WeatherForecast {

fn summary(&self) -> String {

format!("The high will be {}, and the low will be {}. The chance of

precipitation is {}%.", self.high\_temp, self.low\_temp,

self.chance\_of\_precipitation)

}

}

Listing 10-13: Bringing the Summarizable trait from our aggregator crate into scope in another crate

This code correctly assumes Summarizable is a public trait, which we made so by putting the pub keyword before trait in Listing 10-11.

One restriction to note with trait implementations is that we can implement a trait on a type as long as 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. We can also implement Summarizable on Vec in our aggregator crate, since we defined Summarizable there.

In other words, we can’t implement external traits on external types. We can’t implement the Display trait on Vec, for example, because both Display and Vec are defined in the standard library. This restriction is part of what’s called 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 it, 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 the methods in a trait, instead implementing all behavior on every type. Then, as we implement the trait on a particular type, we can choose to keep or override each method’s default behavior.

Listing 10-14 shows how to specify a default string for the summary method of the Summarize trait instead of simply defining the method signature like we did in Listing 10-11:

Filename: lib.rs

pub trait Summarizable {

fn summary(&self) -> String {

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

}

}

Listing 10-14: Definition of a Summarizable trait with a default implementation of the summary 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 Summarizable for NewsArticle {}.

Even though we’re no longer choosing to define the summary method on NewsArticle directly, we've provided a default implementation and specified that NewsArticle implements the Summarizable trait, so we can still call the summary 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.summary());

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

Creating a default implementation for summary does not require us to change anything about the implementations of Summarizable on Tweet in Listing 10-12 or WeatherForecast in Listing 10-13, because the syntax for overriding a default implementation is exactly 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 requires implementers to specify a small part of it. For example, we could define the Summarizable trait to have an author\_summary method whose implementation is required, then a summary method that has a default implementation that calls the author\_summary method:

pub trait Summarizable {

fn author\_summary(&self) -> String;

fn summary(&self) -> String {

format!("(Read more from {}...)", self.author\_summary())

}

}

To use this version of Summarizable, we just define author\_summary when we implement the trait on a type:

impl Summarizable for Tweet {

fn author\_summary(&self) -> String {

format!("@{}", self.username)

}

}

Once we define author\_summary, we can call summary on instances of the Tweet struct, and the default implementation of summary will call the definition of author\_summary that we’ve provided.

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.summary());

This should print one new tweet: (Read more from @horse\_ebooks...).

Note that it is not possible to call the default implementation from an overridden implementation.

Trait Bounds

Now that you've learned how to defined traits and implemented those traits on types, we can see how to use traits with generic type parameters. As mentioned earlier, 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 10-12, we implemented the Summarizable trait on the types NewsArticle and Tweet. We can define a function notify that calls the summary method on its parameter item, which is of the generic type T. To be able to call summary on item without getting an error, we can use trait bounds on T to specify that item must be of a type that implements the Summarizable trait:

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

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

}

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 call notify and pass in any instance of NewsArticle or Tweet. This allows the external code from Listing 10-13 that’s using our aggregator crate to call our notify function and pass in an instance of WeatherForecast, since Summarizable is implemented for WeatherForecast as well. Code that calls the function with any other type, like a String or an i32, won’t compile, since those don’t implement Summarizable.

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 summary method, we can use T: Summarizable + Display to say T can be any type that implements both Summarizable and Display.

There are downsides to using too many trait bounds, however. Because each generic has its own trait bounds, functions with multiple generic type parameters can have lots of trait bound information between a function’s name and its parameter list, making them 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’ve learned how to specify the behavior you want to use using the generic type parameter’s bounds, you can now return to Listing 10-5 to fix the definition of the largest function that uses a generic type parameter! Last time we were trying out that code, we were getting this error:

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

|

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 that the largest function can work on slices of any type that can be compared. We don’t need to bring PartialOrd into scope because it’s in the prelude.

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

This time, when we compile this, we’ll get a different set of errors:

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

--> src/main.rs:4:23

|

4 | let mut largest = list[0];

| ----------- ^^^^^^^ cannot move out of here

| |

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

error[E0507]: cannot move out of borrowed content

--> src/main.rs:6:9

|

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

| ^----

| ||

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

| cannot move out of borrowed content

The key thing to note about this error is the line: cannot move out of type [T], a non-copy array. With our non-generic versions of the largest function, we were only trying to find the largest i32 or char. As we discussed in Chapter 4, 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, it became possible that the list parameter could have types in it that don’t implement the Copy trait, which would mean we wouldn’t be able to move the value out of list[0] and into the largest variable, resulting in this error.

To call this code with only those types that implement the trait Copy, we can simply 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 both the PartialOrd and Copy traits, like i32 and char:

Filename: src/main.rs

use std::cmp::PartialOrd;

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 numbers = vec![34, 50, 25, 100, 65];

let result = largest(&numbers);

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

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

let result = largest(&chars);

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 our largest function to the types that implement the Copy trait, we could specify that T has the trait bound Clone instead of Copy and 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, though, 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. For example, 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 either the Clone or Copy trait bounds and we could avoid heap allocations altogether. Try implementing these alternate solutions on your own!

Traits and trait bounds let us write code that uses generic type parameters to not only reduce duplication, but also specify to the compiler exactly the behavior we want the generic type to have. The compiler can then use the trait bound information to check that all the concrete types used with our code provide the right behavior. Unlike in dynamically typed languages, where we’d get an error at runtime if we tried to call a method on a type that the type didn’t implement, Rust moves these errors to compile time so that 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 since we’ve already checked at compile time, which improves performance without having to give up the flexibility of generics.

There’s another kind of generic that we’ve already been using 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 learn how lifetimes do that.

Validating References with Lifetimes

One thing we didn’t discuss in our discussions on references 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. Similar to to the way we annotate types when multiple types are possible, in cases where 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.

This concept is somewhat different from tools in other programming languages, arguably making lifetimes Rust’s most distinctive feature. Although we won’t cover lifetimes in its entirety in this chapter, we’ll cover common ways you might encounter lifetime syntax to get you familiar with the concepts. See Chapter 19 for more detailed information about lifetimes.

Lifetimes Prevent Dangling References

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

{

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

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 out the value in r. This code won't compile, therefore, before r has gone out of scope when we try to use it the final time.

Uninitialized Variables Cannot Be Used

The next few examples declare variables without giving them an initial value, so that the variable name exists in the outer scope. This might appear to be in conflict with Rust having no null value. However, if we try to use a variable before giving it a value, we’ll get a compile-time error. Let’s start by examining the error we get when we compile the code in Listing 10-16:

error: `x` does not live long enough

|

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.” This is because 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?

The Borrow Checker

The Rust compiler has a borrow checker, which compares scopes to determine that all borrows are valid. Listing 10-17 shows , but with annotations showing the lifetimes of the variables:

{

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

// |

{ // |

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

r = &x; // | |

} // -+ |

// |

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

// |

// -------+

}

Listing 10-17: Annotations of the lifetimes of x and r, 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 an object 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 this so it doesn't have any dangling reference and compiles without any errors:

{

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 we’ve seen where the lifetimes of references are and how Rust analyzes lifetimes to ensure references will always be valid, let’s talk about generic lifetimes of parameters and return values in the context of functions.

Generic Lifetimes in Functions

Let’s write a function that returns the longest of two string slices. This function will take two string slices, and return a string slice. The code in Listing 10-19 should print The longest string is abcd once we’ve implemented the longest function:

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 longest of two string slices

Note that we want the function to take string slices, which are references, since we don’t want the longest function to take ownership of its arguments. 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 Arguments” section of Chapter 4 for more discussion about why these are the arguments 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 longest of two string slices, but does not yet compile

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

error[E0106]: missing lifetime specifier

|

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 says 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, since 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 10-17 and 10-18 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 get around this, we’re going to add generic lifetime parameters that define the relationship between the references so that the borrow checker can perform its analysis.

Lifetime Annotation Syntax

Lifetime annotations don’t actually change how long any of the references live. Just like we let functions accept any type when the signature specifies a generic type parameter, we can let functions can accept references with any lifetime by specifying a generic lifetime parameter. You can use lifetime annotations to relate the lifetimes of multiple references to each other without affecting the lifetimes themselves.

Lifetime annotations have a slightly unusual syntax: the names of lifetime parameters must start with an apostrophe ' and are usually all lowercase very short, like generic types. 'a is the name most people use as a default. 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 since the annotations are meant to tell Rust how generic lifetime parameters of multiple references relate to each other. For example, if we have a function with the parameter first that is a reference to an i32 with lifetime 'a, and the function has another parameter named second that is another reference to an i32 that also has the lifetime 'a, the same lifetime annotations indicate that the references first and second must both live as long as the generic lifetime.

Lifetime Annotations in Function Signatures

Now let’s examine lifetime annotations in the context of the longest function. Just like with generic type parameters, we need to declare generic lifetime parameters inside angle brackets between the function name and the parameter list. The constraint is that all the references in the parameters and the return value must have the same lifetime, which we’ll name 'a, and then add 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 should compile and produce the result we want when used with the main function in Listing 10-19.

The function signature now tells Rust that for some lifetime 'a, the function should get two parameters, both of which are string slices that live at least as long as the lifetime 'a. It also instructs the function to return a string slice that will also last at least as long as the lifetime 'a. This is the rule we want Rust to enforce.

As discussed, by specifying the lifetime parameters in this function signature, we’re not changing the lifetimes of any values passed in or returned, but we’re specifying that the borrow checker reject any values that don’t adhere to this rule. Note that this function doesn’t need to know exactly how long x and y will live, only that there is some scope that 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. This is because while Rust can analyze the code within the function without any help, 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 arguments or return values on its own, which might be different each time the function is called. In this case, we need to annotate the lifetimes ourselves.

When we pass concrete references to longest, the concrete lifetime that substitutes '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 be valid as long as the shorter of the lifetimes of x and y.

Let’s see how we can 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 outer scope. Run this 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 when run.

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. Next, we’ll move the println! that uses result outside of the inner scope, after it has ended. The code in Listing 10-23 will not compile yet, but we’ll fix it soon:

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 won’t compile

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

error: `string2` does not live long enough

|

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

| ------- borrow occurs here

7 | }

| ^ `string2` dropped here while still borrowed

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

9 | }

| - borrowed value needs to live until here

The error says that in order 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, 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!. The compiler, however, cannot. 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 some 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 your experiments will pass the borrow checker or not 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 argument 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

}

In this example, we’ve specified a lifetime parameter 'a for the parameter x and the return type, but not for the parameter y, since 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 arguments. If the reference returned does not refer to one of the arguments, then it must refer to a value created within this function, which would be a dangling reference since 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’s the error message we get:

error: `result` does not live long enough

|

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 block

at 1:44...

|

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

| ^

The problem is that result goes out of scope and gets cleaned up at the end of the longest function, and we’re trying to return a reference to result from the function. There’s 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 that the calling function is then responsible for cleaning up the value.

Ultimately, lifetime syntax is about connecting the lifetimes of various arguments 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 is 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 10-24 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()

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

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

}

Listing 10-24: 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. Just like with generic data types, we declare the name of the generic lifetime parameter inside angle brackets after the name of the struct so that we can use the lifetime parameter in the body of the struct definition.

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.

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 the “String Slices” section, 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 Chapter 4 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 of pre-1.0 Rust, this would not 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 that the borrow checker can infer the lifetimes in these situations without needing explicit annotations.

I mention this piece of Rust history 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. For example, if Rust deterministically applies the rules but there’s still ambiguity as to what lifetimes the references have, it 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 two 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.

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 arguments gets two separate lifetime parameters: fn foo<'a, 'b>(x: &'a i32, y: &'b i32), and so on.

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.

If there are multiple input lifetime parameters, but one of them is &self or &mut self because this is a method, then the lifetime of self is assigned to all output lifetime parameters. This makes writing methods much nicer.

Let’s pretend we’re the compiler and apply these rules to figure out what the lifetimes of the references in the signature of the first\_word function in Listing 10-25 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 says each parameter gets its own lifetime. We’re going to 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. Because the second rule says the lifetime of the one input parameter gets assigned to the output lifetime, 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 do another example, this time with the longest function that had no lifetime parameters when we started working with 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 {

We can see that the second rule doesn’t apply since there is more than one input lifetime. The third rule does not apply either, because this is a function rather than a method, so none of the parameters are self. After going 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 from 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 only really applies in method signatures, we'll ook 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 10-10. Where we declare and use the lifetime parameters depends on whether they’re related to the struct fields or the method arguments 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, since 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, here’s 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’s 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 may 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 is one that 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 which case 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 10-21 that returns the longest of two string slices, but now with an extra argument named ann of the generic type T, which may be filled in by any type that implements the Display trait as specified by the where clause. This extra argument will be printed out 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 both 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 unduplicated code 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 will 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 happens at compile time, which doesn’t affect run-time performance!

Believe it or not, there’s much more to learn on these topics: Chapter 17 will discuss trait objects, which are another way to use traits. Chapter 19 will cover more complex scenarios involving lifetime annotations. Chapter 20 will explore some advanced type system features. But in the next chapter, you’ll learn how to write tests in Rust so that we can make sure our code is working the way it should.