[Unsafe Rust 2](#_Toc508803063)

[Unsafe Superpowers 2](#_Toc508803064)

[Dereferencing a Raw Pointer 3](#_Toc508803065)

[Calling an Unsafe Function or Method 5](#_Toc508803066)

[Creating a Safe Abstraction Over Unsafe Code 6](#_Toc508803067)

[Calling Rust Functions from Other Languages 10](#_Toc508803068)

[Accessing or Modifying a Mutable Static Variable 11](#_Toc508803069)

[Implementing an Unsafe Trait 13](#_Toc508803070)

[When to Use Unsafe Code 13](#_Toc508803071)

[Advanced Lifetimes 13](#_Toc508803072)

[Lifetime Subtyping Ensures One Lifetime Outlives Another 14](#_Toc508803073)

[Lifetime Bounds on References to Generic Types 20](#_Toc508803074)

[Inference of Trait Object Lifetimes 22](#_Toc508803075)

[Advanced Traits 23](#_Toc508803076)

[Associated Types Specify Placeholder Types in Trait Definitions 23](#_Toc508803077)

[Associated Types Versus Generics 24](#_Toc508803078)

[Default Generic Type Parameters and Operator Overloading 25](#_Toc508803079)

[Fully Qualified Syntax for Disambiguation: Calling Methods with the Same Name 27](#_Toc508803080)

[Using Supertraits to Require One Trait’s Functionality Within Another Trait 32](#_Toc508803081)

[The Newtype Pattern to Implement External Traits on External Types 34](#_Toc508803082)

[Advanced Types 36](#_Toc508803083)

[Using the Newtype Pattern for Type Safety and Abstraction 36](#_Toc508803084)

[Type Aliases Create Type Synonyms 37](#_Toc508803085)

[The ! Never Type that Never Returns 39](#_Toc508803086)

[Dynamically Sized Types & Sized 41](#_Toc508803087)

[The Sized Trait 42](#_Toc508803088)

[Advanced Functions & Closures 43](#_Toc508803089)

[Function Pointers 43](#_Toc508803090)

[Returning Closures 45](#_Toc508803091)

[Summary 46](#_Toc508803092)

19

Advanced Features

By now, you’ve learned the most commonly used parts of the Rust programming language. Before we do one more project in Chapter 20, we’ll look at a few aspects of the language you might run into every once in a while. You can use this chapter as a reference for when you encounter any unknowns when using Rust. The features you’ll learn to use in this chapter are useful in very specific situations. Although you might not reach for them often, we want to make sure you have a grasp of all the features Rust has to offer.

In this chapter, we’ll cover:

Unsafe Rust: Opt out of some of Rust’s guarantees and manually uphold the guarantees

Advanced lifetimes: Syntax for complex lifetime situations

Advanced traits: Associated types, default type parameters, fully qualified syntax, supertraits, and the newtype pattern in relation to traits

Advanced types: More about the newtype pattern, type aliases, the never type, and dynamically sized types

Advanced functions and closures: Function pointers and returning closures

It’s a panoply of Rust features with something for everyone! Let’s dive in!

Unsafe Rust

All the code we’ve discussed so far has had Rust’s memory safety guarantees enforced at compile time. However, Rust has a second language hidden inside it that doesn’t enforce these memory safety guarantees: it’s called unsafe Rust and works just like regular Rust, but gives us extra superpowers.

Unsafe Rust exists because, by nature, static analysis is conservative. When the compiler tries to determine whether or not code upholds the guarantees, it’s better for it to reject some valid programs rather than accepting some invalid programs. Although the code might be okay, as far as Rust is able to tell, it’s not! In these cases, we can use unsafe code to tell the compiler, “trust me, I know what I’m doing.” The downside is that we use it at our own risk: if we use unsafe code incorrectly, problems due to memory unsafety, such as null pointer dereferencing, can occur.

Another reason Rust has an unsafe alter ego is that the underlying computer hardware is inherently unsafe. If Rust didn’t let us do unsafe operations, we couldn’t do certain tasks. Rust needs to allow us to do low-level systems programming, such as directly interacting with the operating system or even writing our own operating system. Working with low-level systems programming is one of the goals of the language. Let’s explore what we can do with unsafe Rust and how to do it.

Unsafe Superpowers

To switch to unsafe Rust, we use the unsafe keyword, and then start a new block that holds the unsafe code. We can take four actions in unsafe Rust, which we call unsafe superpowers, that we can’t in safe Rust. Those superpowers include the ability to:

Dereference a raw pointer

Call an unsafe function or method

Access or modify a mutable static variable

Implement an unsafe trait

It’s important to understand that unsafe doesn’t turn off the borrow checker or disable any other of Rust’s safety checks: if you use a reference in unsafe code, it will still be checked. The unsafe keyword only gives us access to these four features that are then not checked by the compiler for memory safety. We still get some degree of safety inside of an unsafe block.

In addition, unsafe does not mean the code inside the block is necessarily dangerous or that it will definitely have memory safety problems: the intent is that as the programmer, we’ll ensure the code inside an unsafe block will access memory in a valid way.

People are fallible, and mistakes will happen, but by requiring these four unsafe operations to be inside blocks annotated with unsafe we’ll know that any errors related to memory safety must be within an unsafe block. Keep unsafe blocks small; you’ll be thankful later when you investigate memory bugs.

To isolate unsafe code as much as possible, it’s best to enclose unsafe code within a safe abstraction and provide a safe API, which we’ll discuss later in the chapter when we examine unsafe functions and methods. Parts of the standard library are implemented as safe abstractions over unsafe code that has been audited. Wrapping unsafe code in a safe abstraction prevents uses of unsafe from leaking out into all the places that you or your users might want to use the functionality implemented with unsafe code, because using a safe abstraction is safe.

Let’s look at each of the four unsafe superpowers in turn: we’ll also look at some abstractions that provide a safe interface to unsafe code.

Dereferencing a Raw Pointer

In Chapter 4, in “Dangling References” on page XX, we mentioned that the compiler ensures references are always valid. Unsafe Rust has two new types called raw pointers that are similar to references. As with references, raw pointers can be immutable or mutable and are written as \*const T and \*mut T, respectively. The asterisk isn’t the dereference operator; it’s part of the type name. In the context of raw pointers, “immutable” means that the pointer can’t be directly assigned to after being dereferenced.

prod: check & Fill xref (ch4)

Different from references and smart pointers, keep in mind that raw pointers:

Are allowed to ignore the borrowing rules by having both immutable and mutable pointers or multiple mutable pointers to the same location

Aren’t guaranteed to point to valid memory

Are allowed to be null

Don’t implement any automatic cleanup

By opting out of having Rust enforce these guarantees, we can make the trade-off of giving up guaranteed safety to gain performance or the ability to interface with another language or hardware where Rust’s guarantees don’t apply.

Listing 19-1 shows how to create an immutable and a mutable raw pointer from references:

let mut num = 5;

let r1 = &num as \*const i32;

let r2 = &mut num as \*mut i32;

Listing 19-1: Creating raw pointers from references

Notice that we don’t include the unsafe keyword in this code. We can create raw pointers in safe code; we just can’t dereference raw pointers outside an unsafe block, as you’ll see in a bit.

We’ve created raw pointers by using as to cast an immutable and a mutable reference into their corresponding raw pointer types. Because we created them directly from references guaranteed to be valid, we know these particular raw pointers are valid, but we can’t make that assumption about just any raw pointer.

Next, we’ll create a raw pointer whose validity we can’t be so certain of. Listing 19-2 shows how to create a raw pointer to an arbitrary location in memory. Trying to use arbitrary memory is undefined: there might be data at that address or there might not, the compiler might optimize the code so there is no memory access, or the program might error with a segmentation fault. Usually, there is no good reason to write code like this, but it is possible:

let address = 0x012345usize;

let r = address as \*const i32;

Listing 19-2: Creating a raw pointer to an arbitrary memory address

Recall that we can create raw pointers in safe code, but we can’t dereference raw pointers and read the data being pointed to. In Listing 19-3, we use the dereference operator \* on a raw pointer that requires an unsafe block:

let mut num = 5;

let r1 = &num as \*const i32;

let r2 = &mut num as \*mut i32;

unsafe {

println!("r1 is: {}", \*r1);

println!("r2 is: {}", \*r2);

}

Listing 19-3: Dereferencing raw pointers within an unsafe block

Creating a pointer does no harm; it’s only when we try to access the value that it points at that we might end up dealing with an invalid value.

Note also that in Listing 19-1 and 19-3 we created \*const i32 and \*mut i32 raw pointers that both pointed to the same memory location, where num is stored. If we instead tried to create an immutable and a mutable reference to num, the code would not have compiled because Rust’s ownership rules don’t allow a mutable reference at the same time as any immutable references. With raw pointers, we can create a mutable pointer and an immutable pointer to the same location, and change data through the mutable pointer, potentially creating a data race. Be careful!

With all of these dangers, why would we ever use raw pointers? One major use case is when interfacing with C code, as you’ll see in the next section, “Calling an Unsafe Function or Method.” Another case is when building up safe abstractions that the borrow checker doesn’t understand. We’ll introduce unsafe functions and then look at an example of a safe abstraction that uses unsafe code.

Calling an Unsafe Function or Method

The second type of operation that requires an unsafe block is calls to unsafe functions. Unsafe functions and methods look exactly like regular functions and methods, but they have an extra unsafe before the rest of the definition. The unsafe keyword in this context indicates the function has requirements we need to uphold when we call this function, because Rust can’t guarantee we’ve met these requirements. By calling an unsafe function within an unsafe block, we’re saying that we’ve read this function’s documentation and take responsibility for upholding the function’s contracts.

Here is an unsafe function named dangerous that doesn’t do anything in its body:

unsafe fn dangerous() {}

unsafe {

dangerous();

}

We must call the dangerous function within a separate unsafe block. If we try to call dangerous without the unsafe block, we’ll get an error:

error[E0133]: call to unsafe function requires unsafe function or block

-->

|

4 | dangerous();

| ^^^^^^^^^^^ call to unsafe function

By inserting the unsafe block around our call to dangerous, we’re asserting to Rust that we’ve read the function’s documentation, we understand how to use it properly, and we’ve verified that we’re fulfilling the contract of the function.

Bodies of unsafe functions are effectively unsafe blocks, so to perform other unsafe operations within an unsafe function, we don’t need to add another unsafe block.

Creating a Safe Abstraction over Unsafe Code

Just because a function contains unsafe code doesn’t mean we need to mark the entire function as unsafe. In fact, wrapping unsafe code in a safe function is a common abstraction. As an example, let’s study a function from the standard library, split\_at\_mut, that requires some unsafe code and explore how we might implement it. This safe method is defined on mutable slices: it takes one slice and makes it two by splitting the slice at the index given as an argument. Listing 19-4 shows how to use split\_at\_mut:

let mut v = vec![1, 2, 3, 4, 5, 6];

let r = &mut v[..];

let (a, b) = r.split\_at\_mut(3);

assert\_eq!(a, &mut [1, 2, 3]);

assert\_eq!(b, &mut [4, 5, 6]);

Listing 19-4: Using the safe split\_at\_mut function

We can’t implement this function using only safe Rust. An attempt might look something like Listing 19-5, which won’t compile. For simplicity, we’ll implement split\_at\_mut as a function rather than a method and only for slices of i32 values rather than for a generic type T.

fn split\_at\_mut(slice: &mut [i32], mid: usize) -> (&mut [i32], &mut [i32]) {

let len = slice.len();

assert!(mid <= len);

(&mut slice[..mid],

&mut slice[mid..])

}

Listing 19-5: An attempted implementation of split\_at\_mut using only safe Rust

This function first gets the total length of the slice, then it asserts that the index given as a parameter is within the slice by checking that it’s less than or equal to the length. The assertion means that if we pass an index that is greater than the index to split the slice at, the function will panic before it attempts to use that index.

Then we return two mutable slices in a tuple: one from the start of the original slice to the mid index and another from mid to the end of the slice.

When we try to compile the code in Listing 19-5, we’ll get an error:

error[E0499]: cannot borrow `\*slice` as mutable more than once at a time

-->

|

6 | (&mut slice[..mid],

| ----- first mutable borrow occurs here

7 | &mut slice[mid..])

| ^^^^^ second mutable borrow occurs here

8 | }

| - first borrow ends here

Rust’s borrow checker can’t understand that we’re borrowing different parts of the slice; it only knows that we’re borrowing from the same slice twice. Borrowing different parts of a slice is fundamentally okay because the two slices aren’t overlapping, but Rust isn’t smart enough to know this. When we know code is okay, but Rust doesn’t, it’s time to reach for unsafe code.

Listing 19-6 shows how to use an unsafe block, a raw pointer, and some calls to unsafe functions to make the implementation of split\_at\_mut work:

use std::slice;

fn split\_at\_mut(slice: &mut [i32], mid: usize) -> (&mut [i32], &mut [i32]) {

let len = slice.len();

let ptr = slice.as\_mut\_ptr();

assert!(mid <= len);

unsafe {

(slice::from\_raw\_parts\_mut(ptr, mid),

slice::from\_raw\_parts\_mut(ptr.offset(mid as isize), len - mid))

}

}

Listing 19-6: Using unsafe code in the implementation of the split\_at\_mut function

Recall from “Slices” in Chapter 4 on page XX that slices are a pointer to some data and the length of the slice. We use the len method to get the length of a slice and the as\_mut\_ptr method to access the raw pointer of a slice. In this case, because we have a mutable slice to i32 values, as\_mut\_ptr returns a raw pointer with the type \*mut i32, which we’ve stored in the variable ptr.

prod: check & fill xref (ch4)

We keep the assertion that the mid index is within the slice. Then we get to the unsafe code: the slice::from\_raw\_parts\_mut function takes a raw pointer and a length, and creates a slice. We use this function to create a slice that starts from ptr and is mid items long. Then we call the offset method on ptr with mid as an argument to get a raw pointer that starts at mid, and we create a slice using that pointer and the remaining number of items after mid as the length.

The function slice::from\_raw\_parts\_mut is unsafe because it takes a raw pointer and must trust that this pointer is valid. The offset method on raw pointers is also unsafe, because it must trust that the offset location is also a valid pointer. Therefore, we had to put an unsafe block around our calls to slice::from\_raw\_parts\_mut and offset so we could call them. By looking at the code and by adding the assertion that mid must be less than or equal to len, we can tell that all the raw pointers used within the unsafe block will be valid pointers to data within the slice. This is an acceptable and appropriate use of unsafe.

Note that we don’t need to mark the resulting split\_at\_mut function as unsafe, and we can call this function from safe Rust. We’ve created a safe abstraction to the unsafe code with an implementation of the function that uses unsafe code in a safe way, because it creates only valid pointers from the data this function has access to.

In contrast, the use of slice::from\_raw\_parts\_mut in Listing 19-7 would likely crash when the slice is used. This code takes an arbitrary memory location and creates a slice ten thousand items long:

use std::slice;

let address = 0x012345usize;

let r = address as \*mut i32;

let slice = unsafe {

slice::from\_raw\_parts\_mut(r, 10000)

};

Listing 19-7: Creating a slice from an arbitrary memory location

We don’t own the memory at this arbitrary location, and there is no guarantee that the slice this code creates contains valid i32 values. Attempting to use slice as though it’s a valid slice results in undefined behavior.

Using extern Functions to Call External Code

Sometimes, your Rust code might need to interact with code written in another language. For this, Rust has a keyword, extern, that facilitates the creation and use of a Foreign Function Interface (FFI). An FFI is a way for a programming language to define functions and enable a different (foreign) programming language to call those functions.

Listing 19-8 demonstrates how to set up an integration with the abs function from the C standard library. Functions declared within extern blocks are always unsafe to call from Rust code. The reason is that other languages don`t enforce Rust’s rules and guarantees, and Rust can’t check them, so responsibility falls on the programmer to ensure safety.

src/main.rs

extern "C" {

fn abs(input: i32) -> i32;

}

fn main() {

unsafe {

println!("Absolute value of -3 according to C: {}", abs(-3));

}

}

Listing 19-8: Declaring and calling an extern function defined in another language

Within the extern "C" block, we list the names and signatures of external functions from another language we want to call. The "C" part defines which application binary interface (ABI) the external function uses: the ABI defines how to call the function at the assembly level. The "C" ABI is the most common and follows the C programming language’s ABI.

Calling Rust Functions from Other Languages

We can also use extern to create an interface that allows other languages to call Rust functions. Instead of an extern block, we add the extern keyword and specify the ABI to use just before the fn keyword. We also need to add a #[no\_mangle] annotation to tell the Rust compiler not to mangle the name of this function. Mangling is when a compiler changes the name we’ve given a function to a different name that contains more information for other parts of the compilation process to consume but is less human readable. Every programming language compiler mangles names slightly differently, so for a Rust function to be nameable by other languages, we must disable the Rust compiler’s name mangling.

In the following example, we make the call\_from\_c function accessible from C code, after it’s compiled to a shared library and linked from C:

#[no\_mangle]

pub extern "C" fn call\_from\_c() {

println!("Just called a Rust function from C!");

}

This usage of extern does not require unsafe.

Accessing or Modifying a Mutable Static Variable

Until now, we’ve not talked about global variables, which Rust does support but can be problematic with Rust’s ownership rules. If two threads are accessing the same mutable global variable, it can cause a data race.

In Rust, global variables are called static variables. Listing 19-9 shows an example declaration and use of a static variable with a string slice as a value:

src/main.rs

static HELLO\_WORLD: &str = "Hello, world!";

fn main() {

println!("name is: {}", HELLO\_WORLD);

}

Listing 19-9: Defining and using an immutable static variable

Static variables are similar to constants, which we discussed in Chapter 3 in the section “Differences Between Variables and Constants.” The names of static variables are in SCREAMING\_SNAKE\_CASE by convention, and we must annotate the variable’s type, which is &'static str in this example. Static variables can only store references with the 'static lifetime, which means the Rust compiler can figure out the lifetime; we don’t need to annotate it explicitly. Accessing an immutable static variable is safe.

prod: check xref

Constants and immutable static variables might seem similar, but a subtle difference is that values in a static variable have a fixed address in memory. Using the value will always access the same data. Constants, on the other hand, are allowed to duplicate their data whenever they’re used.

Another difference between constants and static variables is that static variables can be mutable. Accessing and modifying mutable static variables is unsafe. Listing 19-10 shows how to declare, access, and modify a mutable static variable named COUNTER:

src/main.rs

static mut COUNTER: u32 = 0;

fn add\_to\_count(inc: u32) {

unsafe {

COUNTER += inc;

}

}

fn main() {

add\_to\_count(3);

unsafe {

println!("COUNTER: {}", COUNTER);

}

}

Listing 19-10: Reading from or writing to a mutable static variable is unsafe

As with regular variables, we specify mutability using the mut keyword. Any code that reads or writes from COUNTER must be within an unsafe block. This code compiles and prints COUNTER: 3 as we would expect because it’s single threaded. Having multiple threads access COUNTER would likely result in data races.

With mutable data that is globally accessible, it’s difficult to ensure there are no data races, which is why Rust considers mutable static variables to be unsafe. Where possible, it’s preferable to use the concurrency techniques and thread-safe smart pointers we discussed in Chapter 16, so the compiler checks that data accessed from different threads is done safely.

prod: check xref

Implementing an Unsafe Trait

The final action that only works with unsafe is implementing an unsafe trait. A trait is unsafe when at least one of its methods has some invariant that the compiler can’t verify. We can declare that a trait is unsafe by adding the unsafe keyword before trait; then implementation of the trait must be marked as unsafe too, as shown in Listing 19-11:

unsafe trait Foo {

// methods go here

}

unsafe impl Foo for i32 {

// method implementations go here

}

Listing 19-11: Defining and implementing an unsafe trait

By using unsafe impl, we’re promising that we’ll uphold the invariants that the compiler can’t verify.

As an example, recall the Sync and Send marker traits we discussed in the section “Extensible Concurrency with the Sync and Send Traits” in Chapter 16: the compiler implements these traits automatically if our types are composed entirely of Send and Sync types. If we implement a type that contains a type that is not Send or Sync, such as raw pointers, and we want to mark that type as Send or Sync, we must use unsafe. Rust can’t verify that our type upholds the guarantees that it can be safely sent across threads or accessed from multiple threads; therefore, we need to do those checks manually and indicate as such with unsafe.

prod: check xref

When to Use Unsafe Code

Using unsafe to take one of the four actions (superpowers) just discussed isn’t wrong or even frowned upon. But it is trickier to get unsafe code correct because the compiler can’t help uphold memory safety. When you have a reason to use unsafe code, you can do so, and having the explicit unsafe annotation makes it easier to track down the source of problems if they occur.

Advanced Lifetimes

In Chapter 10 in the section “Validating References with Lifetimes,” you learned how to annotate references with lifetime parameters to tell Rust how lifetimes of different references relate. You saw how every reference has a lifetime, but most of the time, Rust will let you elide lifetimes. Now we’ll look at three advanced features of lifetimes that we haven’t covered yet:

prod: check xref

Lifetime subtyping: Ensures that one lifetime outlives another lifetime

Lifetime bounds: Specifies a lifetime for a reference to a generic type

Inference of trait object lifetimes: How the compiler infers trait object lifetimes and when they need to be specified

Lifetime Subtyping Ensures One Lifetime Outlives Another

Lifetime subtyping specifies that one lifetime should outlive another lifetime. To explore lifetime subtyping, imagine we want to write a parser. We’ll use a structure called Context that holds a reference to the string we’re parsing. We’ll write a parser that will parse this string and return success or failure. The parser will need to borrow the Context to do the parsing. Listing 19-12 implements this parser code, except the code doesn’t have the required lifetime annotations, so it won’t compile:

src/lib.rs

struct Context(&str);

struct Parser {

context: &Context,

}

impl Parser {

fn parse(&self) -> Result<(), &str> {

Err(&self.context.0[1..])

}

}

Listing 19-12: Defining a parser without lifetime annotations

Compiling the code results in errors because Rust expects lifetime parameters on the string slice in Context and the reference to a Context in Parser.

For simplicity’s sake, the parse function returns Result<(), &str>. That is, the function will do nothing on success, and on failure will return the part of the string slice that didn’t parse correctly. A real implementation would provide more error information and would return a structured data type when parsing succeeds. We won’t be discussing those details because they aren’t relevant to the lifetimes part of this example.

To keep this code simple, we won’t write any parsing logic. However, it’s very likely that somewhere in the parsing logic we would handle invalid input by returning an error that references the part of the input that is invalid; this reference is what makes the code example interesting in regard to lifetimes. Let’s pretend that the logic of our parser is that the input is invalid after the first byte. Note that this code might panic if the first byte is not on a valid character boundary; again, we’re simplifying the example to focus on the lifetimes involved.

To get this code to compile, we need to fill in the lifetime parameters for the string slice in Context and the reference to the Context in Parser. The most straightforward way to do this is to use the same lifetime everywhere, as shown in Listing 19-13:

src/lib.rs

struct Context<'a>(&'a str);

struct Parser<'a> {

context: &'a Context<'a>,

}

impl<'a> Parser<'a> {

fn parse(&self) -> Result<(), &str> {

Err(&self.context.0[1..])

}

}

Listing 19-13: Annotating all references in Context and Parser with the same lifetime parameter

This code compiles just fine. It tells Rust that a Parser holds a reference to a Context with lifetime 'a, and that Context holds a string slice that also lives as long as the reference to the Context in Parser. Rust’s compiler error message stated that lifetime parameters were required for these references, and we’ve now added lifetime parameters.

Next, in Listing 19-14, we’ll add a function that takes an instance of Context, uses a Parser to parse that context, and returns what parse returns. This code doesn’t quite work:

src/lib.rs

fn parse\_context(context: Context) -> Result<(), &str> {

Parser { context: &context }.parse()

}

Listing 19-14: An attempt to add a parse\_context function that takes a Context and uses a Parser

We get two verbose errors when we try to compile the code with the addition of the parse\_context function:

error[E0597]: borrowed value does not live long enough

--> src/lib.rs:14:5

|

14 | Parser { context: &context }.parse()

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

15 | }

| - temporary value only lives until here

|

note: borrowed value must be valid for the anonymous lifetime #1 defined on the function body at 13:1...

--> src/lib.rs:13:1

|

13 | / fn parse\_context(context: Context) -> Result<(), &str> {

14 | | Parser { context: &context }.parse()

15 | | }

| |\_^

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

--> src/lib.rs:14:24

|

14 | Parser { context: &context }.parse()

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

15 | }

| - borrowed value only lives until here

|

note: borrowed value must be valid for the anonymous lifetime #1 defined on the function body at 13:1...

--> src/lib.rs:13:1

|

13 | / fn parse\_context(context: Context) -> Result<(), &str> {

14 | | Parser { context: &context }.parse()

15 | | }

| |\_^

These errors state that the Parser instance that is created and the context parameter live only until the end of the parse\_context function. But they both need to live for the entire lifetime of the function.

In other words, Parser and context need to outlive the entire function and be valid before the function starts as well as after it ends for all the references in this code to always be valid. The Parser we’re creating and the context parameter go out of scope at the end of the function, because parse\_context takes ownership of context.

To figure out why these errors occur, let’s look at the definitions in Listing 19-13 again, specifically the references in the signature of the parse method:

fn parse(&self) -> Result<(), &str> {

Remember the elision rules? If we annotate the lifetimes of the references rather than eliding, the signature would be as follows:

fn parse<'a>(&'a self) -> Result<(), &'a str> {

That is, the error part of the return value of parse has a lifetime that is tied to the lifetime of the Parser instance (that of &self in the parse method signature). That makes sense: the returned string slice references the string slice in the Context instance held by the Parser, and the definition of the Parser struct specifies that the lifetime of the reference to Context and the lifetime of the string slice that Context holds should be the same.

The problem is that the parse\_context function returns the value returned from parse, so the lifetime of the return value of parse\_context is tied to the lifetime of the Parser as well. But the Parser instance created in the parse\_context function won’t live past the end of the function (it’s temporary), and context will go out of scope at the end of the function (parse\_context takes ownership of it).

Rust thinks we’re trying to return a reference to a value that goes out of scope at the end of the function, because we annotated all the lifetimes with the same lifetime parameter. The annotations told Rust the lifetime of the string slice that Context holds is the same as that of the lifetime of the reference to Context that Parser holds.

The parse\_context function can’t see that within the parse function, the string slice returned will outlive Context and Parser, and that the reference parse\_context returns refers to the string slice, not to Context or Parser.

By knowing what the implementation of parse does, we know that the only reason the return value of parse is tied to the Parser is because it’s referencing the Parser’s Context, which is referencing the string slice. So, it’s really the lifetime of the string slice that parse\_context needs to care about. We need a way to tell Rust that the string slice in Context and the reference to the Context in Parser have different lifetimes and that the return value of parse\_context is tied to the lifetime of the string slice in Context.

First, we’ll try giving Parser and Context different lifetime parameters, as shown in Listing 19-15. We’ll use 's and 'c as lifetime parameter names to clarify which lifetime goes with the string slice in Context and which goes with the reference to Context in Parser. Note that this solution won’t completely fix the problem, but it’s a start. We’ll look at why this fix isn’t sufficient when we try to compile.

src/lib.rs

struct Context<'s>(&'s str);

struct Parser<'c, 's> {

context: &'c Context<'s>,

}

impl<'c, 's> Parser<'c, 's> {

fn parse(&self) -> Result<(), &'s str> {

Err(&self.context.0[1..])

}

}

fn parse\_context(context: Context) -> Result<(), &str> {

Parser { context: &context }.parse()

}

Listing 19-15: Specifying different lifetime parameters for the references to the string slice and to Context

We’ve annotated the lifetimes of the references in all the same places that we annotated them in Listing 19-13. But this time we used different parameters depending on whether the reference goes with the string slice or with Context. We’ve also added an annotation to the string slice part of the return value of parse to indicate that it goes with the lifetime of the string slice in Context.

When we try to compile now, we get the following error:

error[E0491]: in type `&'c Context<'s>`, reference has a longer lifetime than the data it references

--> src/lib.rs:4:5

|

4 | context: &'c Context<'s>,

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

|

note: the pointer is valid for the lifetime 'c as defined on the struct at 3:1

--> src/lib.rs:3:1

|

3 | / struct Parser<'c, 's> {

4 | | context: &'c Context<'s>,

5 | | }

| |\_^

note: but the referenced data is only valid for the lifetime 's as defined on the struct at 3:1

--> src/lib.rs:3:1

|

3 | / struct Parser<'c, 's> {

4 | | context: &'c Context<'s>,

5 | | }

| |\_^

Rust doesn’t know of any relationship between 'c and 's. To be valid, the referenced data in Context with lifetime 's needs to be constrained to guarantee that it lives longer than the reference with lifetime 'c. If 's is not longer than 'c, the reference to Context might not be valid.

Now we get to the point of this section: the Rust feature lifetime subtyping specifies that one lifetime parameter lives at least as long as another one. In the angle brackets where we declare lifetime parameters, we can declare a lifetime 'a as usual and declare a lifetime 'b that lives at least as long as 'a by declaring 'b using the syntax 'b: 'a.

In our definition of Parser, to say that 's (the lifetime of the string slice) is guaranteed to live at least as long as 'c (the lifetime of the reference to Context), we change the lifetime declarations to look like this:

src/lib.rs

struct Parser<'c, 's: 'c> {

context: &'c Context<'s>,

}

Now the reference to Context in the Parser and the reference to the string slice in the Context have different lifetimes; we’ve ensured that the lifetime of the string slice is longer than the reference to the Context.

That was a very long-winded example, but as we mentioned at the start of this chapter, Rust’s advanced features are very specific. You won’t often need the syntax we described in this example, but in such situations, you’ll know how to refer to something you have a reference to.

Lifetime Bounds on References to Generic Types

In the “Trait Bounds” section in Chapter 10, we discussed using trait bounds on generic types. We can also add lifetime parameters as constraints on generic types; these are called lifetime bounds. Lifetime bounds help Rust verify that references in generic types won’t outlive the data they’re referencing.

prod: check xref

As an example, consider a type that is a wrapper over references. Recall the RefCell<T> type from “RefCell<T> and the Interior Mutability Pattern” on page XX in Chapter 15: its borrow and borrow\_mut methods return the types Ref and RefMut, respectively. These types are wrappers over references that keep track of the borrowing rules at runtime. The definition of the Ref struct is shown in Listing 19-16, without lifetime bounds for now:

prod: check/link xref (ch 15)

src/lib.rs

struct Ref<'a, T>(&'a T);

Listing 19-16: Defining a struct to wrap a reference to a generic type, without lifetime bounds to start

Without explicitly constraining the lifetime 'a in relation to the generic parameter T, Rust will error because it doesn’t know how long the generic type T will live:

error[E0309]: the parameter type `T` may not live long enough

--> src/lib.rs:1:19

|

1 | struct Ref<'a, T>(&'a T);

| ^^^^^^

|

= help: consider adding an explicit lifetime bound `T: 'a`...

note: ...so that the reference type `&'a T` does not outlive the data it points at

--> src/lib.rs:1:19

|

1 | struct Ref<'a, T>(&'a T);

| ^^^^^^

Because T can be any type, T could be a reference or a type that holds one or more references, each of which could have their own lifetimes. Rust can’t be sure T will live as long as 'a.

Fortunately, the error provides helpful advice on how to specify the lifetime bound in this case:

consider adding an explicit lifetime bound `T: 'a` so that the reference type

`&'a T` does not outlive the data it points at

Listing 19-17 shows how to apply this advice by specifying the lifetime bound when we declare the generic type T:

struct Ref<'a, T: 'a>(&'a T);

Listing 19-17: Adding lifetime bounds on T to specify that any references in T live at least as long as 'a

This code now compiles because the T: 'a syntax specifies that T can be any type, but if it contains any references, the references must live at least as long as 'a.

We could solve this problem in a different way, as shown in the definition of a StaticRef struct in Listing 19-18, by adding the 'static lifetime bound on T. This means if T contains any references, they must have the 'static lifetime:

struct StaticRef<T: 'static>(&'static T);

Listing 19-18: Adding a 'static lifetime bound to T to constrain T to types that have only 'static references or no references

Because 'static means the reference must live as long as the entire program, a type that contains no references meets the criteria of all references living as long as the entire program (because there are no references). For the borrow checker concerned about references living long enough, there is no real distinction between a type that has no references and a type that has references that live forever: both are the same for determining whether or not a reference has a shorter lifetime than what it refers to.

Inference of Trait Object Lifetimes

In Chapter 17 in the section “Using Trait Objects that Allow for Values of Different Types,” we discussed trait objects, consisting of a trait behind a reference, that allow us to use dynamic dispatch. We haven’t yet discussed what happens if the type implementing the trait in the trait object has a lifetime of its own. Consider Listing 19-19 where we have a trait Red and a struct Ball. The Ball struct holds a reference (and thus has a lifetime parameter) and also implements trait Red. We want to use an instance of Ball as the trait object Box<Red>:

prod: check xref

src/main.rs

trait Red { }

struct Ball<'a> {

diameter: &'a i32,

}

impl<'a> Red for Ball<'a> { }

fn main() {

let num = 5;

let obj = Box::new(Ball { diameter: &num }) as Box<Red>;

}

Listing 19-19: Using a type that has a lifetime parameter with a trait object

This code compiles without any errors, even though we haven’t explicitly annotated the lifetimes involved in obj. This code works because there are rules for working with lifetimes and trait objects:

The default lifetime of a trait object is 'static.

With &'a Trait or &'a mut Trait, the default lifetime is 'a.

With a single T: 'a clause, the default lifetime is 'a.

With multiple T: 'a-like clauses, there is no default; we must be explicit.

When we must be explicit, we can add a lifetime bound on a trait object like Box<Red> using the syntax Box<Red + 'static> or Box<Red + 'a>, depending on whether the reference lives for the entire program or not. As with the other bounds, the syntax adding a lifetime bound means that any implementor of the Red trait that has references inside the type must have the same lifetime specified in the trait object bounds as those references.

Next, let’s look at some other advanced features that manage traits.

Advanced Traits

We first covered traits in “Traits: Defining Shared Behavior” on page XX in Chapter 10, but as with lifetimes, we didn’t discuss the more advanced details. Now that you know more about Rust, we can get into the nitty-gritty.

prod: check/link xref (ch 10)

Associated Types Specify Placeholder Types in Trait Definitions

Associated types connect a type placeholder with a trait such that the trait method definitions can use these placeholder types in their signatures. The implementor of a trait will specify the concrete type to be used in this type’s place for the particular implementation. That way, we can define a trait that uses some types without needing to know exactly what those types are until the trait is implemented.

We’ve described most of the advanced features in this chapter as being rarely needed. Associated types are somewhere in the middle: they’re used more rarely than features explained in the rest of the book, but more commonly than many of the other features discussed in this chapter.

One example of a trait with an associated type is the Iterator trait that the standard library provides. This trait has an associated type named Item that stands in for the type of the values the type implementing the Iterator trait is iterating over. In “The Iterator Trait and the next Method” on page XX in Chapter 13, we mentioned that the definition of the Iterator trait is as shown in Listing 19-20:

prod: check/link xref (ch13)

pub trait Iterator {

type Item;

fn next(&mut self) -> Option<Self::Item>;

}

Listing 19-20: The definition of the Iterator trait that has an associated type Item

The type Item is a placeholder type, and the next method’s definition shows that it will return values of type Option<Self::Item>. Implementors of the Iterator trait will specify the concrete type for Item, and the next method will return an Option containing a value of that concrete type.

Associated Types vs. Generics

Associated types might seem like a similar concept to generics, in that they allow us to define a function without specifying what types it can handle. So why use associated types?

Let’s examine the difference between the two concepts with an example from Chapter 13 that implements the Iterator trait on the Counter struct. In Listing 13-21 on page XX, we specified that the Item type was u32:

prod: check xref/link listing 13-21

src/lib.rs

impl Iterator for Counter {

type Item = u32;

fn next(&mut self) -> Option<Self::Item> {

// --snip--

This syntax seems comparable to generics. So why not just define the Iterator trait with generics, as shown in Listing 19-21?

pub trait Iterator<T> {

fn next(&mut self) -> Option<T>;

}

Listing 19-21: A hypothetical definition of the Iterator trait using generics

The difference is that when using generics, as in Listing 19-21, we must annotate the types in each implementation. The reason is that we can also implement Iterator<String> for Counter or any other type, which would give us multiple implementations of Iterator for Counter. In other words, when a trait has a generic parameter, it can be implemented for a type multiple times, changing the concrete types of the generic type parameters each time. When we use the next method on Counter, we would have to provide type annotations to indicate which implementation of Iterator we want to use.

With associated types, we don’t need to annotate types because we can’t implement a trait on a type multiple times. In Listing 19-20 with the definition that uses associated types, we can only choose what the type of Item will be once, because there can only be one impl Iterator for Counter. We don’t have to specify that we want an iterator of u32 values everywhere that we call next on Counter.

Default Generic Type Parameters and Operator Overloading

When we use generic type parameters, we can specify a default concrete type for the generic type. This eliminates the need for implementors of the trait to specify a concrete type if the default type works. The syntax for specifying a default type for a generic type is <PlaceholderType=ConcreteType> when declaring the generic type.

A great example of a situation where this technique is useful is with operator overloading. Operator overloading is customizing the behavior of an operator (such as +) in particular situations.

Rust doesn’t allow you to create your own operators or overload arbitrary operators. But you can overload the operations and corresponding traits listed in std::ops by implementing the traits associated with the operator. For example, in Listing 19-22 we overload the + operator to add two Point instances together. We do this by implementing the Add trait on a Point struct:

src/main.rs

use std::ops::Add;

#[derive(Debug,PartialEq)]

struct Point {

x: i32,

y: i32,

}

impl Add for Point {

type Output = Point;

fn add(self, other: Point) -> Point {

Point {

x: self.x + other.x,

y: self.y + other.y,

}

}

}

fn main() {

assert\_eq!(Point { x: 1, y: 0 } + Point { x: 2, y: 3 },

Point { x: 3, y: 3 });

}

Listing 19-22: Implementing the Add trait to overload the + operator for Point instances

The add method adds the x values of two Point instances and the y values of two Point instances to create a new Point. The Add trait has an associated type named Output that determines the type returned from the add method.

The default generic type in this code is within the Add trait. Here is its definition:

trait Add<RHS=Self> {

type Output;

fn add(self, rhs: RHS) -> Self::Output;

}

This code should look generally familiar: a trait with one method and an associated type. The new part is RHS=Self in the angle brackets: this syntax is called default type parameters. The RHS generic type parameter (short for “right hand side”) defines the type of the rhs parameter in the add method. If we don’t specify a concrete type for RHS when we implement the Add trait, the type of RHS will default to Self, which will be the type we’re implementing Add on.

When we implemented Add for Point, we used the default for RHS because we wanted to add two Point instances. Let’s look at an example of implementing the Add trait where we want to customize the RHS type rather than using the default.

We have two structs holding values in different units, Millimeters and Meters. We want to add values in millimeters to values in meters and have the implementation of Add do the conversion correctly. We can implement Add for Millimeters with Meters as the RHS, as shown in Listing 19-23:

src/lib.rs

use std::ops::Add;

struct Millimeters(u32);

struct Meters(u32);

impl Add<Meters> for Millimeters {

type Output = Millimeters;

fn add(self, other: Meters) -> Millimeters {

Millimeters(self.0 + (other.0 \* 1000))

}

}

Listing 19-23: Implementing the Add trait on Millimeters to add Millimeters to Meters

To add Millimeters and Meters, we specify impl Add<Meters> to set the value of the RHS type parameter instead of using the default of Self.

We use default type parameters in two main ways:

To extend a type without breaking existing code

To allow customization in specific cases most users won’t need

The standard library’s Add trait is an example of the second purpose: usually, you’ll add two like types, but the Add trait provides the ability for customizing beyond that. Using a default type parameter in the Add trait definition means you don’t have to specify the extra parameter most of the time. In other words, a bit of implementation boilerplate isn’t needed, making it easier to use the trait.

The first purpose is similar to the second but in reverse: if we want to add a type parameter to an existing trait, we can give it a default to let us extend the functionality of the trait without breaking the existing implementation code.

Fully Qualified Syntax for Disambiguation: Calling Methods with the Same Name

Nothing in Rust prevents a trait from having a method with the same name as another trait’s method, nor does Rust prevent us from implementing both traits on one type. It’s also possible to implement a method directly on the type with the same name as methods from traits.

When calling methods with the same name, we need to tell Rust which one we want to use. Consider the code in Listing 19-24 where we’ve defined two traits, Pilot and Wizard, that both have a method called fly. We then implement both traits on a type Human that already has a method named fly implemented on it. Each fly method does something different:

src/main.rs

trait Pilot {

fn fly(&self);

}

trait Wizard {

fn fly(&self);

}

struct Human;

impl Pilot for Human {

fn fly(&self) {

println!("This is your captain speaking.");

}

}

impl Wizard for Human {

fn fly(&self) {

println!("Up!");

}

}

impl Human {

fn fly(&self) {

println!("\*waving arms furiously\*");

}

}

Listing 19-24: Two traits defined to have a fly method and implementations of those traits on the Human type in addition to a fly method on Human directly

When we call fly on an instance of Human, the compiler defaults to calling the method that is directly implemented on the type, as shown in Listing 19-25:

src/main.rs

fn main() {

let person = Human;

person.fly();

}

Listing 19-25: Calling fly on an instance of Human

Running this code will print \*waving arms furiously\*, which shows that Rust called the fly method implemented on Human directly.

To call the fly methods from either the Pilot trait or the Wizard trait, we need to use more explicit syntax to specify which fly method we mean. Listing 19-26 demonstrates this syntax:

src/main.rs

fn main() {

let person = Human;

Pilot::fly(&person);

Wizard::fly(&person);

person.fly();

}

Listing 19-26: Specifying which trait’s fly method we want to call

Specifying the trait name before the method name clarifies to Rust which implementation of fly we want to call. We could also write Human::fly(&person), which is equivalent to person.fly() that we used in Listing 19-26 but is a bit longer to write if we don’t need to disambiguate.

Running this code prints the following:

This is your captain speaking.

Up!

\*waving arms furiously\*

Because the fly method takes a self parameter, if we had two types that both implement one trait, Rust can figure out which implementation of a trait to use based on the type of self.

However, associated functions that are part of traits don’t have a self parameter. When two types in the same scope implement that trait, Rust can’t figure out which type we mean unless we use fully qualified syntax. For example, the Animal trait in Listing 19-27 has the associated function baby\_name, the implementation of Animal for the struct Dog, and the associated function baby\_name defined on Dog directly:

src/main.rs

trait Animal {

fn baby\_name() -> String;

}

struct Dog;

impl Dog {

fn baby\_name() -> String {

String::from("Spot")

}

}

impl Animal for Dog {

fn baby\_name() -> String {

String::from("puppy")

}

}

fn main() {

println!("A baby dog is called a {}", Dog::baby\_name());

}

Listing 19-27: A trait with an associated function and a type that has an associated function with the same name that also implements the trait

This code is for an animal shelter that wants to name all puppies Spot, which is implemented in the baby\_name associated function that is defined on Dog. The Dog type also implements the trait Animal, which describes characteristics that all animals have. Baby dogs are called puppies, and that is expressed in the implementation of the Animal trait on Dog in the baby\_name function associated with the Animal trait.

In main, we call the Dog::baby\_name function, which calls the associated function defined on Dog directly. This code prints the following:

A baby dog is called a Spot

This output isn’t what we wanted. We want to call the baby\_name function that is part of the Animal trait that we implemented on Dog so the code prints A baby dog is called a puppy. The technique of specifying the trait name that we used in Listing 19-26 doesn’t help here; if we change main to the code in Listing 19-28, we’ll get a compilation error:

src/main.rs

fn main() {

println!("A baby dog is called a {}", Animal::baby\_name());

}

Listing 19-28: Attempting to call the baby\_name function from the Animal trait, but Rust doesn’t know which implementation to use

Because Animal::baby\_name is an associated function rather than a method, and thus doesn’t have a self parameter, Rust can’t figure out which implementation of Animal::baby\_name we want. We’ll get this compiler error:

error[E0283]: type annotations required: cannot resolve `\_: Animal`

--> src/main.rs:20:43

|

20 | println!("A baby dog is called a {}", Animal::baby\_name());

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

|

= note: required by `Animal::baby\_name`

To disambiguate and tell Rust that we want to use the implementation of Animal for Dog, we need to use fully qualified syntax, which is the most specific we can be when calling a function. Listing 19-29 demonstrates how to use fully qualified syntax:

src/main.rs

fn main() {

println!("A baby dog is called a {}", <Dog as Animal>::baby\_name());

}

Listing 19-29: Using fully qualified syntax to specify that we want to call the baby\_name function from the Animal trait as implemented on Dog

We’re providing Rust with a type annotation within the angle brackets, which indicates we want to call the baby\_name method from the Animal trait as implemented on Dog by saying that we want to treat the Dog type as an Animal for this function call. This code will now print what we want:

A baby dog is called a puppy

In general, fully qualified syntax is defined as follows:

<Type as Trait>::function(receiver\_if\_method, next\_arg, ...);

For associated functions, there would not be a receiver: there would only be the list of other arguments. We could use fully qualified syntax everywhere that we call functions or methods. However, we’re allowed to omit any part of this syntax that Rust can figure out from other information in the program. We only need to use this more verbose syntax in cases where there are multiple implementations that use the same name and Rust needs help to identify which implementation we want to call.

Using Supertraits to Require One Trait’s Functionality Within Another Trait

Sometimes, we might need one trait to use another trait’s functionality. In this case, we need to rely on the dependent trait also being implemented. The trait we’re relying on is a supertrait of the trait we’re implementing.

For example, let’s say we want to make an OutlinePrint trait with an outline\_print method that will print a value framed in asterisks. That is, given a Point struct that implements Display to result in (x, y), when we call outline\_print on a Point instance that has 1 for x and 3 for y, it should print the following:

\*\*\*\*\*\*\*\*\*\*

\* \*

\* (1, 3) \*

\* \*

\*\*\*\*\*\*\*\*\*\*

In the implementation of outline\_print, we want to use the Display trait’s functionality. Therefore, we need to specify that the OutlinePrint trait will only work for types that also implement Display and provide the functionality that OutlinePrint needs. We can do that in the trait definition by specifying OutlinePrint: Display. This technique is similar to adding a trait bound to the trait. Listing 19-30 shows an implementation of the OutlinePrint trait:

src/main.rs

use std::fmt;

trait OutlinePrint: fmt::Display {

fn outline\_print(&self) {

let output = self.to\_string();

let len = output.len();

println!("{}", "\*".repeat(len + 4));

println!("\*{}\*", " ".repeat(len + 2));

println!("\* {} \*", output);

println!("\*{}\*", " ".repeat(len + 2));

println!("{}", "\*".repeat(len + 4));

}

}

Listing 19-30: Implementing the OutlinePrint trait that requires the functionality from Display

Because we’ve specified that OutlinePrint requires the Display trait, we can use the to\_string function that is automatically implemented for any type that implements Display. If we tried to use to\_string without adding: Display after the trait name, we’d get an error saying that no method named to\_string was found for the type &Self in the current scope.

Let’s see what happens when we try to implement OutlinePrint on a type that doesn’t implement Display, such as the Point struct:

src/main.rs

struct Point {

x: i32,

y: i32,

}

impl OutlinePrint for Point {}

We get an error saying that Display is required but not implemented:

error[E0277]: the trait bound `Point: std::fmt::Display` is not satisfied

--> src/main.rs:20:6

|

20 | impl OutlinePrint for Point {}

| ^^^^^^^^^^^^ `Point` cannot be formatted with the default formatter; try using `:?` instead if you are using a format string

|

= help: the trait `std::fmt::Display` is not implemented for `Point`

To fix this, we implement Display on Point and satisfy the constraint that OutlinePrint requires, like so:

src/main.rs

use std::fmt;

impl fmt::Display for Point {

fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {

write!(f, "({}, {})", self.x, self.y)

}

}

Then implementing the OutlinePrint trait on Point will compile successfully, and we can call outline\_print on a Point instance to display it within an outline of asterisks.

The Newtype Pattern to Implement External Traits on External Types

In “Implementing a Trait on a Type” on page XX in Chapter 10, we mentioned the orphan rule that states we’re allowed to implement a trait on a type as long as either the trait or the type are local to our crate. It’s possible to get around this restriction using the newtype pattern, which involves creating a new type in a tuple struct. (We covered tuple structs in “Tuple Structs Without Named Fields to Create Different Types” on page XX in Chapter 5.) The tuple struct will have one field and be a thin wrapper around the type we want to implement a trait for. Then the wrapper type is local to our crate, and we can implement the trait on the wrapper. Newtype is a term that originates from the Haskell programming language. There is no runtime performance penalty for using this pattern, and the wrapper type is elided at compile time.

prod: check/link xrefs (ch 10 & ch 5)

As an example, let’s say we want to implement Display on Vec, which the orphan rule prevents us from doing directly because the Display trait and the Vec type are defined outside our crate. We can make a Wrapper struct that holds an instance of Vec; then we can implement Display on Wrapper and use the Vec value, as shown in Listing 19-31:

src/main.rs

use std::fmt;

struct Wrapper(Vec<String>);

impl fmt::Display for Wrapper {

fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {

write!(f, "[{}]", self.0.join(", "))

}

}

fn main() {

let w = Wrapper(vec![String::from("hello"), String::from("world")]);

println!("w = {}", w);

}

Listing 19-31: Creating a Wrapper type around Vec<String> to implement Display

The implementation of Display uses self.0 to access the inner Vec, because Wrapper is a tuple struct and Vec is the item at index 0 in the tuple. Then we can use the functionality of the Display type on Wrapper.

The downside of using this technique is that Wrapper is a new type, so it doesn’t have the methods of the value it’s holding. We would have to implement all the methods of Vec directly on Wrapper so it can delegate to self.0, allowing us to treat Wrapper exactly like a Vec. If we wanted the new type to have every method the inner type has, implementing the Deref trait (discussed in Chapter 15 in “Treating Smart Pointers Like Regular References with the Deref Trait” on page XX) on the Wrapper to return the inner type would be a solution. If we don’t want the Wrapper type to have all the methods of the inner type, in order to restrict the Wrapper type’s behavior for example, we would have to implement just the methods we do want manually.

prod: check xref/link xref (ch15)

Now you know how the newtype pattern is used in relation to traits; it’s also a useful pattern even when traits are not involved. Let’s switch focus and look at some advanced ways to interact with Rust’s type system.

Advanced Types

The Rust type system has some features that we’ve mentioned in this book but haven’t yet discussed. We’ll start by discussing newtypes in general as we examine why newtypes are useful as types. Then we’ll move on to type aliases, a feature similar to newtypes but with slightly different semantics. We’ll also discuss the ! type and dynamically sized types.

Using the Newtype Pattern for Type Safety and Abstraction

Note: This section assumes you’ve read the earlier section “The Newtype Pattern to Implement External Traits on External Types” on page XX.

prod: fill xref (this chapter)

The newtype pattern is useful for other tasks beyond what we’ve discussed so far, including statically enforcing that values are never confused and as an indication of the units of a value. You saw an example of using newtypes to indicate units in Listing 19-23: recall that the Millimeters and Meters structs wrapped u32 values in a newtype. If we wrote a function with a parameter of type Millimeters, we couldn’t compile a program that accidentally tried to call that function with a value of type Meters or a plain u32.

Another use of the newtype pattern is in abstracting away some implementation details of a type: the new type can expose a public API that is different from the API of the private inner type if we used the new type directly to restrict the available functionality, for example.

Newtypes can also hide internal implementation. For example, we could provide a People type to wrap a HashMap<i32, String> that stores a person’s ID associated with their name. Code using People would only interact with the public API we provide, such as a method to add a name string to the People collection; that code wouldn’t need to know that we assign an i32 ID to names internally. The newtype pattern is a lightweight way to achieve encapsulation to hide implementation details, which we discussed in “Encapsulation that Hides Implementation Details” on page XX in Chapter 17.

prod: check xref/link xref (ch17)

Type Aliases Create Type Synonyms

Along with the newtype pattern, Rust provides the ability to declare a type alias to give an existing type another name. For this we use the type keyword. For example, we can create the alias Kilometers to i32 like so:

type Kilometers = i32;

Now, the alias Kilometers is a synonym for i32; unlike the Millimeters and Meters types we created in Listing 19-23, Kilometers is not a separate, new type. Values that have the type Kilometers will be treated the same as values of type i32:

type Kilometers = i32;

let x: i32 = 5;

let y: Kilometers = 5;

println!("x + y = {}", x + y);

Because Kilometers and i32 are the same type, we can add values of both types and we can pass Kilometers values to functions that take i32 parameters. However, using this method, we don’t get the type checking benefits that we get from the newtype pattern discussed earlier.

The main use case for type synonyms is to reduce repetition. For example, we might have a lengthy type like this:

Box<Fn() + Send + 'static>

Writing this lengthy type in function signatures and as type annotations all over the code can be tiresome and error prone. Imagine having a project full of code like that in Listing 19-32:

let f: Box<Fn() + Send + 'static> = Box::new(|| println!("hi"));

fn takes\_long\_type(f: Box<Fn() + Send + 'static>) {

// --snip--

}

fn returns\_long\_type() -> Box<Fn() + Send + 'static> {

// --snip--

}

Listing 19-32: Using a long type in many places

A type alias makes this code more manageable by reducing the repetition. In Listing 19-33, we’ve introduced an alias named Thunk for the verbose type and can replace all uses of the type with the shorter alias Thunk:

type Thunk = Box<Fn() + Send + 'static>;

let f: Thunk = Box::new(|| println!("hi"));

fn takes\_long\_type(f: Thunk) {

// --snip--

}

fn returns\_long\_type() -> Thunk {

// --snip--

}

Listing 19-33: Introducing a type alias Thunk to reduce repetition

This code is much easier to read and write! Choosing a meaningful name for a type alias can help communicate your intent as well (thunk is a word for code to be evaluated at a later time, so it’s an appropriate name for a closure that gets stored).

Type aliases are also commonly used with the Result<T, E> type for reducing repetition. Consider the std::io module in the standard library. I/O operations often return a Result<T, E> to handle situations when operations fail to work. This library has a std::io::Error struct that represents all possible I/O errors. Many of the functions in std::io will be returning Result<T, E> where the E is std::io::Error, such as these functions in the Write trait:

use std::io::Error;

use std::fmt;

pub trait Write {

fn write(&mut self, buf: &[u8]) -> Result<usize, Error>;

fn flush(&mut self) -> Result<(), Error>;

fn write\_all(&mut self, buf: &[u8]) -> Result<(), Error>;

fn write\_fmt(&mut self, fmt: fmt::Arguments) -> Result<(), Error>;

}

The Result<..., Error> is repeated a lot. As such, std::io has this type of alias declaration:

type Result<T> = Result<T, std::io::Error>;

Because this declaration is in the std::io module, we can use the fully qualified alias std::io::Result<T>; that is, a Result<T, E> with the E filled in as std::io::Error. The Write trait function signatures end up looking like this:

pub trait Write {

fn write(&mut self, buf: &[u8]) -> Result<usize>;

fn flush(&mut self) -> Result<()>;

fn write\_all(&mut self, buf: &[u8]) -> Result<()>;

fn write\_fmt(&mut self, fmt: Arguments) -> Result<()>;

}

The type alias helps in two ways: it makes code easier to write and it gives us a consistent interface across all of std::io. Because it’s an alias, it’s just another Result<T, E>, which means we can use any methods that work on Result<T, E> with it, as well as special syntax like ?.

The ! Never Type that Never Returns

Rust has a special type named ! that’s known in type theory lingo as the empty type because it has no values. We prefer to call it the never type because it stands in the place of the return type when a function will never return. Here is an example:

fn bar() -> ! {

// --snip--

}

This code is read as “the function bar returns never.” Functions that return never are called diverging functions. We can’t create values of the type ! so bar can never possibly return.

But what use is a type you can never create values for? Recall the code in Chapter 2 that we added in the “Handling Invalid Input” section; we’ve reproduced it here in Listing 19-34:

prod: check xref

let guess: u32 = match guess.trim().parse() {

Ok(num) => num,

Err(\_) => continue,

};

Listing 19-34: A match with an arm that ends in continue

At the time, we skipped over some details in this code. In Chapter 6 in the section “The match Control Flow Operator,” we discussed that match arms must all return the same type. So, for example, the following code doesn’t work:

let guess = match guess.trim().parse() {

Ok(\_) => 5,

Err(\_) => "hello",

}

The type of guess in this code would have to be an integer and a string, and Rust requires that guess can only have one type. So what does continue return? How were we allowed to return a u32 from one arm and have another arm that ends with continue in Listing 19-34?

As you might have guessed, continue has a ! value. That is, when Rust computes the type of guess, it looks at both match arms, the former with a value of u32 and the latter with a ! value. Because ! can never have a value, Rust decides that the type of guess is u32.

The formal way of describing this behavior is that expressions of type ! can be coerced into any other type. We’re allowed to end this match arm with continue because continue doesn’t return a value; instead, it moves control back to the top of the loop, so in the Err case, we never assign a value to guess.

The never type is useful with the panic! macro as well. Remember the unwrap function that we call on Option<T> values to produce a value or panic? Here is its definition:

impl<T> Option<T> {

pub fn unwrap(self) -> T {

match self {

Some(val) => val,

None => panic!("called `Option::unwrap()` on a `None` value"),

}

}

}

In this code, the same thing happens as in the match in Listing 19-34: Rust sees that val has the type T and panic! has the type ! so the result of the overall match expression is T. This code works because panic! doesn’t produce a value; it ends the program. In the None case, we won’t be returning a value from unwrap, so this code is valid.

One final expression that has the type ! is a loop:

print!("forever ");

loop {

print!("and ever ");

}

Here, the loop never ends, so ! is the value of the expression. However, this wouldn’t be true if we included a break, because the loop would terminate when it got to the break.

Dynamically Sized Types and Sized

Due to Rust’s need to know certain details, such as how much space to allocate for a value of a particular type, there is a corner of its type system that can be confusing: the concept of dynamically sized types. Sometimes referred to as DSTs or unsized types, these types let us write code using values whose size we can only know at runtime.

Let’s dig into the details of a dynamically sized type called str, which we’ve been using throughout the book. That’s right, not &str, but str on its own, is a DST. We can’t know how long the string is until runtime, meaning we can’t create a variable of type str, nor can we take an argument of type str. Consider the following code, which does not work:

let s1: str = "Hello there!";

let s2: str = "How's it going?";

Rust needs to know how much memory to allocate for any value of a particular type, and all values of a type must use the same amount of memory. If Rust allowed us to write this code, these two str values would need to take up the same amount of space. But they have different lengths: s1 needs 12 bytes of storage and s2 needs 15. This is why it’s not possible to create a variable holding a dynamically sized type.

So what do we do? In this case, you already know the answer: we make the types of s1 and s2 a &str rather than a str. Recall that in “String Slices” on page XX in Chapter 4 we said the slice data structure stores the starting position and the length of the slice.

prod: check/link xref (ch 4)

So although a &T is a single value that stores the memory address of where the T is located, a &str is two values: the address of the str and its length. As such, we can know the size of a &str value at compile time: it’s two times the size of a usize in length. That is, we always know the size of a &str, no matter how long the string it refers to is. In general, this is the way in which dynamically sized types are used in Rust: they have an extra bit of metadata that stores the size of the dynamic information. The golden rule of dynamically sized types is that we must always put values of dynamically sized types behind a pointer of some kind.

We can combine str with all kinds of pointers: for example, Box<str> or Rc<str>. In fact, you’ve seen this before but with a different dynamically sized type: traits. Every trait is a dynamically sized type we can refer to by using the name of the trait. In Chapter 17 in “Using Trait Objects that Allow for Values of Different Types” on page XX, we mentioned that to use traits as trait objects, we must put them behind a pointer, such as &Trait or Box<Trait> (Rc<Trait> would work too).

prod: check/link xref (ch17)

To work with DSTs, Rust has a particular trait called the Sized trait to determine whether or not a type’s size is known at compile time. This trait is automatically implemented for everything whose size is known at compile time. In addition, Rust implicitly adds a bound on Sized to every generic function. That is, a generic function definition like this:

fn generic<T>(t: T) {

// --snip--

}

is actually treated as though we had written this:

fn generic<T: Sized>(t: T) {

// --snip--

}

By default, generic functions will only work on types that have a known size at compile time. However, you can use the following special syntax to relax this restriction:

fn generic<T: ?Sized>(t: &T) {

// --snip--

}

A trait bound on ?Sized is the opposite of a trait bound on Sized: we would read this as “T may or may not be Sized.” This syntax is only available for Sized, not any other traits.

Also note that we switched the type of the t parameter from T to &T. Because the type might not be Sized, we need to use it behind some kind of pointer. In this case, we’ve chosen a reference.

Next, we’ll talk about functions and closures!

Advanced Functions and Closures

Finally, we’ll explore some advanced features related to functions and closures, which include function pointers and returning closures.

Function Pointers

We’ve talked about how to pass closures to functions; you can also pass regular functions to functions! This technique is useful when we want to pass a function we’ve already defined rather than defining a new closure. We do this using function pointers to allow us to use functions as arguments to other functions. Functions coerce to the type fn (with a lowercase f), not to be confused with the Fn closure trait. The fn type is called a function pointer. The syntax for specifying that a parameter is a function pointer is similar to that of closures, as shown in Listing 19-35:

src/main.rs

fn add\_one(x: i32) -> i32 {

x + 1

}

fn do\_twice(f: fn(i32) -> i32, arg: i32) -> i32 {

f(arg) + f(arg)

}

fn main() {

let answer = do\_twice(add\_one, 5);

println!("The answer is: {}", answer);

}

Listing 19-35: Using the fn type to accept a function pointer as an argument

This code prints The answer is: 12. We specify that the parameter f in do\_twice is an fn that takes one parameter of type i32 and returns an i32. We can then call f in the body of do\_twice. In main, we can pass the function name add\_one as the first argument to do\_twice.

Unlike closures, fn is a type rather than a trait, so we specify fn as the parameter type directly rather than declaring a generic type parameter with one of the Fn traits as a trait bound.

Function pointers implement all three of the closure traits (Fn, FnMut, and FnOnce), so we can always pass a function pointer as an argument for a function that expects a closure. It’s best to write functions using a generic type and one of the closure traits so your functions can accept either functions or closures.

An example of where you would want to only accept fn and not closures is when interfacing with external code that doesn’t have closures: C functions can accept functions as arguments, but C doesn’t have closures.

As an example of where we can use either a closure defined inline or a named function, let’s look at a use of map. To use the map function to turn a vector of numbers into a vector of strings, we could use a closure, like this:

let list\_of\_numbers = vec![1, 2, 3];

let list\_of\_strings: Vec<String> = list\_of\_numbers

.iter()

.map(|i| i.to\_string())

.collect();

Or we could name a function as the argument to map instead of the closure, like this:

let list\_of\_numbers = vec![1, 2, 3];

let list\_of\_strings: Vec<String> = list\_of\_numbers

.iter()

.map(ToString::to\_string)

.collect();

Note that we must use the fully qualified syntax that we talked about earlier in “Advanced Traits” on page XX because there are multiple functions available named to\_string. Here, we’re using the to\_string function defined in the ToString trait, which the standard library has implemented for any type that implements Display.

Prod: check and link xref (this chapter)

Some people prefer this style, and some people prefer to use closures. They end up compiling to the same code, so use whichever style is clearer to you.

Returning Closures

Closures are represented by traits, which means we can’t return closures directly. In most cases where we might want to return a trait, we can instead use the concrete type that implements the trait as the return value of the function. But we can’t do that with closures because they don’t have a concrete type that is returnable; we’re not allowed to use the function pointer fn as a return type, for example.

The following code tries to return a closure directly, but it won’t compile:

fn returns\_closure() -> Fn(i32) -> i32 {

|x| x + 1

}

The compiler error is as follows:

error[E0277]: the trait bound `std::ops::Fn(i32) -> i32 + 'static:

std::marker::Sized` is not satisfied

-->

|

1 | fn returns\_closure() -> Fn(i32) -> i32 {

| ^^^^^^^^^^^^^^ `std::ops::Fn(i32) -> i32 + 'static` does not have a constant size known at compile-time

|

= help: the trait `std::marker::Sized` is not implemented for `std::ops::Fn(i32) -> i32 + 'static`

= note: the return type of a function must have a statically known size

The error references the Sized trait again! Rust doesn’t know how much space it will need to store the closure. We saw a solution to this problem earlier. We can use a trait object:

fn returns\_closure() -> Box<Fn(i32) -> i32> {

Box::new(|x| x + 1)

}

This code will compile just fine. For more about trait objects, refer to “Trait Objects” on page XX in Chapter 17.

prod: check/link xref (ch17)

Summary

Whew! Now you have some features of Rust in your toolbox that you won’t use often, but you’ll know they’re available in very particular circumstances. We’ve introduced several complex topics so that when you encounter them in error message suggestions or in other peoples’ code, you’ll be able to recognize these concepts and syntax. Use this chapter as a reference to guide you to solutions.

Next, we’ll put everything we’ve discussed throughout the book into practice and do one more project!