[Defining and Instantiating Structs 1](#_Toc489718915)

[Field Init Shorthand when Variables Have the Same Name as Fields 3](#_Toc489718916)

[Creating Instances From Other Instances With Struct Update Syntax 4](#_Toc489718917)

[Tuple Structs without Named Fields to Create Different Types 4](#_Toc489718918)

[Unit-Like Structs without Any Fields 5](#_Toc489718919)

[An Example Program Using Structs 6](#_Toc489718920)

[Refactoring with Tuples 7](#_Toc489718921)

[Refactoring with Structs: Adding More Meaning 8](#_Toc489718922)

[Adding Useful Functionality with Derived Traits 10](#_Toc489718923)

[Method Syntax 12](#_Toc489718924)

[Defining Methods 12](#_Toc489718925)

[Methods with More Parameters 14](#_Toc489718926)

[Associated Functions 16](#_Toc489718927)

[Multiple impl Blocks 17](#_Toc489718928)

[Summary 17](#_Toc489718929)

5

Using Structs to Structure Related Data

A struct, or structure, is a custom data type that lets us name and package together multiple related values that make up a meaningful group. If you’re familiar with an object-oriented language, a struct is like an object’s data attributes. In this chapter, we’ll compare and contrast tuples with structs, demonstrate how to use structs, and discuss how to define methods and associated functions to specify behavior associated with a struct’s data. The struct and enum (which is discussed in Chapter 6) concepts are the building blocks for creating new types in your program’s domain to take full advantage of Rust’s compile time type checking.

Prod: link xref

Defining and Instantiating Structs

Structs are similar to tuples, which were discussed in Chapter 3. Like tuples, the pieces of a struct can be different types. Unlike tuples, we name each piece of data so it’s clear what the values mean. As a result of these names, structs are more flexible than tuples: we don’t have to rely on the order of the data to specify or access the values of an instance.

Prod: link xref

To define a struct, we enter the keyword struct and name the entire struct. A struct’s name should describe the significance of the pieces of data being grouped together. Then, inside curly braces, we define the names and types of the pieces of data, which we call fields. For example, Listing 5-1 shows a struct to store information about a user account:

struct User {

username: String,

email: String,

sign\_in\_count: u64,

active: bool,

}

Listing 5-1: A User struct definition

To use a struct after we’ve defined it, we create an instance of that struct by specifying concrete values for each of the fields. We create an instance by stating the name of the struct, and then add curly braces containing key: value pairs where the keys are the names of the fields and the values are the data we want to store in those fields. We don’t have to specify the fields in the same order in which we declared them in the struct. In other words, the struct definition is like a general template for the type, and instances fill in that template with particular data to create values of the type. For example, we can declare a particular user as shown in Listing 5-2:

let user1 = User {

email: String::from("someone@example.com"),

username: String::from("someusername123"),

active: true,

sign\_in\_count: 1,

};

Listing 5-2: Creating an instance of the User struct

To get a specific value from a struct, we can use dot notation. If we wanted just this user’s email address, we can use user1.email wherever we want to use this value. If the struct is mutable, we can change a value using the dot notation and assigning into a particular field. Listing 5-3 shows how to change the value in the email field of a mutable User instance:

let mut user1 = User {

email: String::from("someone@example.com"),

username: String::from("someusername123"),

active: true,

sign\_in\_count: 1,

};

user1.email = String::from("anotheremail@example.com");

Listing 5-3: Changing the value in the email field of a User instance

As with any expression, we can construct a new instance of the struct as the last expression in the function body to implicitly return that new instance.

Listing 5-4 shows a build\_user function that returns a User instance with the given email and username. The active field gets the value of true, and the sign\_in\_count gets a value of 1.

fn build\_user(email: String, username: String) -> User {

User {

email: email,

username: username,

active: true,

sign\_in\_count: 1,

}

}

Listing 5-4: A build\_user function that takes an email and username and returns a User instance

It makes sense to name the function arguments with the same name as the struct fields, but having to Repeat the email and username field names and variables is a bit tedious. If the struct had more fields, repeating each name would get even more annoying. Luckily, there's a convenient shorthand!

Using the Field Init Shorthand when Variables and Fields Have the Same Name

Because the parameter names and the struct field names are exactly the same in Listing 5-4, we can use the field init shorthand syntax to rewrite build\_user so that it behaves exactly the same but doesn’t have the repetition of email and username in the way shown in Listing 5-5.

fn build\_user(email: String, username: String) -> User {

User {

email,

username,

active: true,

sign\_in\_count: 1,

}

}

Listing 5-5: A build\_user function that uses field init shorthand since the email and username parameters have the same name as struct fields

Here, we’re creating a new instance of the User struct, which has a field named email. We want to set the email field’s value to the value in the email parameter of the build\_user function. Because the email field and the email parameter have the same name, we only need to write email rather than email: email.

Creating Instances From Other Instances With Struct Update

It’s often useful to create a new instance that uses most of the old instance’s values, but changes some. We do this using the struct update syntax.

First, Listing 5-6, shows how we create a new User instance in user2 without the update syntax, by setting new values for email and username, but otherwise using the same values from Listing 5-2:

let user2 = User {

email: String::from("another@example.com"),

username: String::from("anotherusername567"),

active: user1.active,

sign\_in\_count: user1.sign\_in\_count,

};

Listing 5-6: Creating a new User instance using some of the values from user1Author having trouble with libreoffice, the second line should read: email: String::from([another@example.com](mailto:another@example.com)”),

Now using the update syntax, we can achieve the same effect with less code, shown in Listing 5-7. The syntax .. specifies that the remaining fields not explicitly set should have the same value as the fields in the given instance.

let user2 = User {

email: String::from("another@example.com"),

username: String::from("anotherusername567"),

..user1

};

Listing 5-7: Using struct update syntax to set a new email and username values for a User instance but use the rest of the values from the fields of the instance in the user1 variable

from .

Tuple Structs without Named Fields to Create Different Types

We can also define structs that look similar to tuples, called tuple structs, that have the added meaning the struct name provides, but don’t have names associated with their fields, just the types of the fields. Tuple structs are useful when you want to give the whole tuple a name and make the tuple be a different type than other tuples, but naming each field as in a regular struct would be verbose or redundant.

To define a tuple struct you start with the struct keyword and the struct name followed by the types in the tuple. For example, here are definitions and usages of two tuple structs named Color and Point:

struct Color(i32, i32, i32);

struct Point(i32, i32, i32);

let black = Color(0, 0, 0);

let origin = Point(0, 0, 0);

Note that the black and origin values are different types, since they’re instances of different tuple structs. Each struct we define is its own type, even though the fields within the struct have the same types. For example, a function that takes a parameter of type Color cannot take a Point as an argument, even though both types are made up of three i32 values. Otherwise, tuple struct instances behave like tuples, which we covered in Chapter 3: you can destructure them into their individual pieces, you can use a . followed by the index to access an individual value, and so on.

Prod: Check xref

Unit-Like Structs without Any Fields

We can also define structs that don’t have any fields! These are called unit-like structs since they behave similarly to (), the unit type. Unit-like structs can be useful in situations such as when you need to implement a trait on some type, but you don’t have any data that you want to store in the type itself. We’ll be discussing traits in Chapter 10.

Prod: Check xref

Start box

Ownership of Struct Data

In the User struct definition in Listing 5-1, we used the owned String type rather than the &str string slice type. This is a deliberate choice because we want instances of this struct to own all of its data and for that data to be valid for as long as the entire struct is valid.

It’s possible for structs to store references to data owned by something else, but to do so requires the use of lifetimes, a Rust feature that is discussed in Chapter 10. Lifetimes ensure that the data referenced by a struct is valid for as long as the struct is. Let’s say you try to store a reference in a struct without specifying lifetimes, like this:

Filename: src/main.rs

Prod: Check xref

struct User {

username: &str,

email: &str,

sign\_in\_count: u64,

active: bool,

}

fn main() {

let user1 = User {

email: "someone@example.com",

username: "someusername123",

active: true,

sign\_in\_count: 1,

};

}

The compiler will complain that it needs lifetime specifiers:

error[E0106]: missing lifetime specifier

-->

|

2 | username: &str,

| ^ expected lifetime parameter

error[E0106]: missing lifetime specifier

-->

|

3 | email: &str,

| ^ expected lifetime parameter

We’ll discuss how to fix these errors so you can store references in structs in Chapter 10, but for now, we’ll fix errors like these using owned types like String instead of references like &str.

End box

An Example Program Using Structs

To understand when we might want to use structs, let’s write a program that calculates the area of a rectangle. We’ll start with single variables, and then refactor the program until we’re using structs instead.

Let’s make a new binary project with Cargo called rectangles that will take the length and width of a rectangle specified in pixels and will calculate the area of the rectangle. Listing 5-8 shows a short program with one way of doing just that in our project’s src/main.rs:

Filename: src/main.rs

fn main() {

let length1 = 50;

let width1 = 30;

println!(

"The area of the rectangle is {} square pixels.",

area(length1, width1)

);

}

fn area(length: u32, width: u32) -> u32 {

length \* width

}

Listing 5-8: Calculating the area of a rectangle specified by its length and width in separate variables

Now, run this program using cargo run:

The area of the rectangle is 1500 square pixels.

Refactoring with Tuples

Even though Listing 5-8 works and figures out the area of the rectangle by calling the area function with each dimension, we can do better. The length and the width are related to each other because together they describe one rectangle.

The issue with this method is evident in the signature of area:

fn area(length: u32, width: u32) -> u32 {

The area function is supposed to calculate the area of one rectangle, but the function we wrote has two parameters. The parameters are related, but that’s not expressed anywhere in our program. It would be more readable and more manageable to group length and width together. We’ve already discussed one way we might do that in the Grouping Values into Tuples section of Chapter 3 on page XX: by using tuples. Listing 5-9 shows another version of our program that uses tuples:

Prod: link xref

Filename: src/main.rs

fn main() {

let rect1 = (50, 30);

println!(

"The area of the rectangle is {} square pixels.",

area(rect1) 

);

}

fn area(dimensions: (u32, u32)) -> u32 {

dimensions.0 \* dimensions.1 

}

Listing 5-8: Specifying the length and width of the rectangle with a tuple

In one way, this program is better. Tuples let us add a bit of structure, and we’re now passing just one argument . But in another way this version is less clear: tuples don’t name their elements, so our calculation has become more confusing because we have to index into the parts of the tuple .

It doesn’t matter if we mix up length and width for the area calculation, but if we want to draw the rectangle on the screen, it would matter! We would have to keep in mind that length is the tuple index 0 and width is the tuple index 1. If someone else worked on this code, they would have to figure this out and keep it in mind as well. It would be easy to forget or mix up these values and cause errors, because we haven’t conveyed the meaning of our data in our code.

Refactoring with Structs: Adding More Meaning

We use structs to add meaning by labeling the data. We can transform the tuple we’re using into a data type with a name for the whole as well as names for the parts, as shown in Listing 5-10:

Filename: src/main.rs

struct Rectangle {

 length: u32,

width: u32,

}

fn main() {

 let rect1 = Rectangle { length: 50, width: 30 };

println!(

"The area of the rectangle is {} square pixels.",

area(&rect1)

);

}

 fn area(rectangle: &Rectangle) -> u32 {

 rectangle.length \* rectangle.width

}

Listing 5-10: Defining a Rectangle struct

Here we’ve defined a struct and named it Rectangle . Inside the {}  we defined the fields as length and width, both of which have type u32. Then in main we create a particular instance of a Rectangle that has a length of 50 and a width of 30 .

Our area function is now defined with one parameter, which we’ve named rectangle, whose type is an immutable borrow of a struct Rectangle instance . As mentioned in Chapter 4, we want to borrow the struct rather than take ownership of it. This way, main retains its ownership and can continue using rect1, which is the reason we use the & in the function signature and where we call the function.

Prod: link xref

The area function accesses the length and width fields of the Rectangle instance . Our function signature for area now indicates exactly what we mean: calculate the area of a Rectangle using its length and width fields. This conveys that the length and width are related to each other, and gives descriptive names to the values rather than using the tuple index values of 0 and 1—a win for clarity.

Adding Useful Functionality with Derived Traits

It would be helpful to be able to print out an instance of the Rectangle while we’re debugging our program in order to see the values for all its fields. Listing 5-11 uses the println! macro as we have been in earlier chapters:

Filename: src/main.rs

struct Rectangle {

length: u32,

width: u32,

}

fn main() {

let rect1 = Rectangle { length: 50, width: 30 };

println!("rect1 is {}", rect1);

}

Listing 5-11: Attempting to print a Rectangle instance

When we run this code, we get an error with this core message:

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

The println! macro can do many kinds of formatting, and by default, {} tells println! to use formatting known as Display: output intended for direct end user consumption. The primitive types we’ve seen so far implement Display by default, because there’s only one way you’d want to show a 1 or any other primitive type to a user. But with structs, the way println! should format the output is less clear because there are more display possibilities: do you want commas or not? Do you want to print the curly braces? Should all the fields be shown? Due to this ambiguity, Rust doesn’t try to guess what we want and structs don’t have a provided implementation of Display.

If we continue reading the errors, we’ll find this helpful note:

note: `Rectangle` cannot be formatted with the default formatter; try using

`:?` instead if you are using a format string

Let’s try it! The println! macro call will now look like println!("rect1 is {:?}", rect1);. Putting the specifier :? inside the {} tells println! we want to use an output format called Debug. Debug is a trait that enables us to print out our struct in a way that is useful for developers so we can see its value while we’re debugging our code.

Run the code with this change. Drat! We still get an error:

error: the trait bound `Rectangle: std::fmt::Debug` is not satisfied

But again, the compiler gives us a helpful note:

note: `Rectangle` cannot be formatted using `:?`; if it is defined in your

crate, add `#[derive(Debug)]` or manually implement it

Rust does include functionality to print out debugging information, but we have to explicitly opt-in to make that functionality available for our struct. To do that, we add the annotation #[derive(Debug)] just before the struct definition, as shown in Listing 5-12:

Filename: src/main.rs

#[derive(Debug)]

struct Rectangle {

length: u32,

width: u32,

}

fn main() {

let rect1 = Rectangle { length: 50, width: 30 };

println!("rect1 is {:?}", rect1);

}

Listing 5-12: Adding the annotation to derive the Debug trait and printing the Rectangle instance using debug formatting

Now when we run the program, we won’t get any errors and we’ll see the following output:

rect1 is Rectangle { length: 50, width: 30 }

Nice! It’s not the prettiest output, but it shows the values of all the fields for this instance, which would definitely help during debugging. When we have larger structs, it’s useful to have output that’s a bit easier to read; in those cases, we can use {:#?} instead of {:?} in the println! string. When we use the {:#?} style in the example, the output will look like this:

rect1 is Rectangle {

length: 50,

width: 30

}

Rust has provided a number of traits for us to use with the derive annotation that can add useful behavior to our custom types. Those traits and their behaviors are listed in Appendix C. We’ll cover how to implement these traits with custom behavior as well as how to create your own traits in Chapter 10.

Prod: confirm xrefs to App C and 10

Our area function is very specific: it only computes the area of rectangles. It would be helpful to tie this behavior more closely to our Rectangle struct, because it won’t work with any other type. Let’s look at how we can continue to refactor this code by turning the area function into an area method defined on our Rectangle type.

Method Syntax

Methods are similar to functions: they’re declared with the fn keyword and their name, they can have parameters and a return value, and they contain some code that is run when they’re called from somewhere else. However, methods are different from functions in that they’re defined within the context of a struct (or an enum or a trait object, which we cover in Chapters 6 and 17, respectively), and their first parameter is always self, which represents the instance of the struct the method is being called on.

Prod: Check xref to 17, link 6

Defining Methods

Let’s change the area function that has a Rectangle instance as a parameter and instead make an area method defined on the Rectangle struct, as shown in Listing 5-13:

Filename: src/main.rs

#[derive(Debug)]

struct Rectangle {

length: u32,

width: u32,

}

 impl Rectangle {

 fn area(&self) -> u32 {

self.length \* self.width

}

}

fn main() {

let rect1 = Rectangle { length: 50, width: 30 };

println!(

"The area of the rectangle is {} square pixels.",

 rect1.area()

);

}

Listing 5-13: Defining an area method on the Rectangle struct

To define the function within the context of Rectangle, we start an impl (implementation) block . Then we move the area function within the impl curly braces  and change the first (and in this case, only) parameter to be self in the signature and everywhere within the body. In main where we called the area function and passed rect1 as an argument, we can instead use method syntax to call the area method on our Rectangle instance . The method syntax goes after an instance: we add a dot followed by the method name, parentheses, and any arguments.

In the signature for area, we use &self instead of rectangle: &Rectangle because Rust knows the type of self is Rectangle due to this method being inside the impl Rectangle context. Note that we still need to use the & before self, just like we did in &Rectangle. Methods can take ownership of self, borrow self immutably as we’ve done here, or borrow self mutably, just like any other parameter.

We’ve chosen &self here for the same reason we used &Rectangle in the function version: we don’t want to take ownership, and we just want to read the data in the struct, not write to it. If we wanted to change the instance that we’ve called the method on as part of what the method does, we’d use &mut self as the first parameter. Having a method that takes ownership of the instance by using just self as the first parameter is rare; this technique is usually used when the method transforms self into something else and we want to prevent the caller from using the original instance after the transformation.

The main benefit of using methods instead of functions, in addition to using method syntax and not having to repeat the type of self in every method’s signature, is for organization. We’ve put all the things we can do with an instance of a type in one impl block rather than making future users of our code search for capabilities of Rectangle in various places in the library we provide.

PROD: START BOX

Where’s the -> Operator?

In languages like C++, two different operators are used for calling methods: you use . if you’re calling a method on the object directly and -> if you’re calling the method on a pointer to the object and need to dereference the pointer first. In other words, if object is a pointer, object->something() is similar to (\*object).something().

Rust doesn’t have an equivalent to the -> operator; instead, Rust has a feature called automatic referencing and dereferencing. Calling methods is one of the few places in Rust that has this behavior.

Here’s how it works: when you call a method with object.something(), Rust automatically adds in &, &mut, or \* so object matches the signature of the method. In other words, the following are the same:

p1.distance(&p2);

(&p1).distance(&p2);

The first one looks much cleaner. This automatic referencing behavior works because methods have a clear receiver—the type of self. Given the receiver and name of a method, Rust can figure out definitively whether the method is reading (&self), mutating (&mut self), or consuming (self). The fact that Rust makes borrowing implicit for method receivers is a big part of making ownership ergonomic in practice.

PROD: END BOX

Methods with More Parameters

Let’s practice using methods by implementing a second method on the Rectangle struct. This time, we want an instance of Rectangle to take another instance of Rectangle and return true if the second Rectangle can fit completely within self; otherwise it should return false. That is, we want to be able to write the program shown in Listing 5-14, once we’ve defined the can\_hold method:

Filename: src/main.rs

fn main() {

let rect1 = Rectangle { length: 50, width: 30 };

let rect2 = Rectangle { length: 40, width: 10 };

let rect3 = Rectangle { length: 45, width: 60 };

println!("Can rect1 hold rect2? {}", rect1.can\_hold(&rect2));

println!("Can rect1 hold rect3? {}", rect1.can\_hold(&rect3));

}

Listing 5-14: Demonstration of using the as-yet-unwritten can\_hold method

And the expected output would look like the following, because both dimensions of rect2 are smaller than the dimensions of rect1, but rect3 is wider than rect1:

Can rect1 hold rect2? true

Can rect1 hold rect3? false

We know we want to define a method, so it will be within the impl Rectangle block. The method name will be can\_hold, and it will take an immutable borrow of another Rectangle as a parameter. We can tell what the type of the parameter will be by looking at the code that calls the method: rect1.can\_hold(&rect2) passes in &rect2, which is an immutable borrow to rect2, an instance of Rectangle. This makes sense because we only need to read rect2 (rather than write, which would mean we’d need a mutable borrow), and we want main to retain ownership of rect2 so we can use it again after calling the can\_hold method. The return value of can\_hold will be a boolean, and the implementation will check whether the length and width of self are both greater than the length and width of the other Rectangle, respectively. Let’s add the new can\_hold method to the impl block from Listing 5-13, shown in Listing 5-15:

Filename: src/main.rs

impl Rectangle {

fn area(&self) -> u32 {

self.length \* self.width

}

fn can\_hold(&self, other: &Rectangle) -> bool {

self.length > other.length && self.width > other.width

}

}

Listing 5-15: Implementing the can\_hold method on Rectangle that takes another Rectangle instance as a parameter

When we run this code with the main function in Listing 5-14, we’ll get our desired output. Methods can take multiple parameters that we add to the signature after the self parameter, and those parameters work just like parameters in functions.

Associated Functions

Another useful feature of impl blocks is that we’re allowed to define functions within impl blocks that don’t take self as a parameter. These are called associated functions because they’re associated with the struct. They’re still functions, not methods, because they don’t have an instance of the struct to work with. You’ve already used the String::from associated function.

Associated functions are often used for constructors that will return a new instance of the struct. For example, we could provide an associated function that would have one dimension parameter and use that as both length and width, thus making it easier to create a square Rectangle rather than having to specify the same value twice:

Filename: src/main.rs

impl Rectangle {

fn square(size: u32) -> Rectangle {

Rectangle { length: size, width: size }

}

}

To call this associated function, we use the :: syntax with the struct name, like let sq = Rectangle::square(3);, for example. This function is namespaced by the struct: the :: syntax is used for both associated functions and namespaces created by modules, which we'll discuss in Chapter 7.

de/au: to revisist the namespace explanation in Ch 7

Prod: check xref

Multiple impl Blocks

Each struct is allowed to have multiple impl blocks. For example, Listing 5-15 is equivalent to the code shown in Listing 5-16, which has each method in its own impl block:

impl Rectangle {

fn area(&self) -> u32 {

self.length \* self.width

}

}

impl Rectangle {

fn can\_hold(&self, other: &Rectangle) -> bool {

self.length > other.length && self.width > other.width

}

}

Listing 5-16: Rewriting Listing 5-15 using multiple impl blocks

There’s no reason to separate these methods into multiple impl blocks here, but it’s valid syntax. We will see a case when multiple impl blocks are useful in Chapter 10 when we discuss generic types and traits.

Prod: check xref

Summary

Structs let us create custom types that are meaningful for our domain. By using structs, we can keep associated pieces of data connected to each other and name each piece to make our code clear. Methods let us specify the behavior that instances of our structs have, and associated functions let us namespace functionality that is particular to our struct without having an instance available.

But structs aren’t the only way we can create custom types: let’s turn to Rust’s enum feature to add another tool to our toolbox.