**Understanding Structs in Rust**

In Rust, a struct is a custom data type that lets you group together related data under one name. Structs are similar to tuples in that they can hold multiple values of different types. However, unlike tuples, each field in a struct is named. This makes structs more readable and flexible, because you do not have to remember the order of the values.

**Defining a Struct**

To define a struct, you use the struct keyword followed by the struct name. Inside curly brackets, you list the fields with their names and types.

A screenshot of a computer code

AI-generated content may be incorrect.

In this example, the User struct represents a user account. It has four fields: a boolean to show if the user is active, a username and email as strings, and a counter that tracks how many times the user has signed in.

**Creating an Instance**

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 brackets 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.

A computer code with text

AI-generated content may be incorrect.

Here, user1 is an instance of the User struct with actual values assigned to its fields.

**Accessing and Modifying Fields**

You can access fields using dot notation. If the instance is mutable, you can also change its fields.

A computer code with text

AI-generated content may be incorrect.

Note that you must make the entire instance mutable. Rust does not allow individual fields to be marked as mutable.

A screenshot of a computer code

AI-generated content may be incorrect.

**Field Init Shorthand**

When the function parameters have the same names as the struct fields, you can use shorthand to simplify the code:

A screen shot of a computer

AI-generated content may be incorrect.

This makes the code cleaner and easier to write.

**Struct Update Syntax**

Sometimes, you want to create a new instance that copies most values from an existing one. Instead of repeating all fields, you can use the struct update syntax.

Without update syntax:

A computer screen shot of a computer code

AI-generated content may be incorrect.

With update syntax:

A computer code on a white background

AI-generated content may be incorrect.

The double-dot syntax copies the remaining fields from user1, user2 now has a different value for email but has the same values for the username, active, and sign\_in\_count fields from user1.

 The ..user1 must come last to specify that any remaining fields should get their values from the corresponding fields in user1, but we can choose to specify values for as many fields as we want in any order, regardless of the order of the fields in the struct’s definition.

Note that the struct update syntax uses = like an assignment; this is because it moves the data. In this example, we can no longer use user1 after creating user2 because the String in the username field of user1 was moved into user2. If we had given user2 new String values for both email and username, and thus only used the active and sign\_in\_count values from user1, then user1 would still be valid after creating user2. We can still use user1.email in this example because its value was *not* moved out.

**Unit-Like Structs**

You can also define structs that have no fields. These are called unit-like structs and are useful when you want to implement a trait without storing data.

A white screen with black text

AI-generated content may be incorrect.

Why use?:

1. **Marker types**: To tag or label data with specific behavior, often used with traits.
2. **Trait implementation**: To implement a trait on a type without storing any fields.

A white background with text

AI-generated content may be incorrect.

1. **Type differentiation**: This means you can use unit structs to create types that look different to the compiler, even if they store no data.

A close-up of a sign

AI-generated content may be incorrect.

Even though both Celsius and Fahrenheit have no fields, you can use them to label or wrap values:

A computer code with black text

AI-generated content may be incorrect.

Now, if someone accidentally passes Fahrenheit where Celsius is expected, the compiler will show an error. This protects your code from confusing different types that use the same data format.

1. **Zero-cost abstraction**: They take no space at runtime but still provide type-level features (you can: Implement traits, use them in type comparisons, add behaviors).

**Note on the Debug Trait in Rust:**

In Rust, the Debug trait allows custom types such as structs to be formatted using the {:?} or {:#?} specifiers. This is primarily intended for developer-facing output during debugging.

By default, user-defined types do not implement Debug. Attempting to print them with println!("{:?}", ...) will result in a compile-time error. To enable debug formatting, you must explicitly opt in by adding the #[derive(Debug)] attribute above the struct definition.

For example:

A screenshot of a computer code

AI-generated content may be incorrect.

Once the trait is derived, the instance can be printed with:

A group of symbols on a white background

AI-generated content may be incorrect.

The dbg! macro is another useful debugging tool. It prints both the expression and its value, along with the source file and line number. Unlike println!, dbg! outputs to standard error (stderr) and takes ownership of the expression, returning its value.

For example:

A close-up of a number

AI-generated content may be incorrect.

The dbg! macro is another useful debugging tool. It prints both the expression and its value, along with the source file and line number. Unlike println!, dbg! outputs to standard error (stderr) and takes ownership of the expression, returning its value so we must use the reference like &rect1.

**Methods and** impl **Blocks in Rust:**

In Rust, methods are functions associated with a specific type, typically defined within an impl (implementation) block. Unlike standalone functions, methods must take self, &self, or &mut self as their first parameter, representing the instance the method is called on.

**Method Definition and Usage**

Methods are declared using the fn keyword inside an impl block. For example:

A computer code with text

AI-generated content may be incorrect.

We chose &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 you want to prevent the caller from using the original instance after the transformation.

**Why Use Methods**

* **Encapsulation**: Related behavior is grouped with the type definition.
* **Clarity**: Method syntax (instance.method()) improves readability over function-style calls.
* **Reuse and Maintenance**: The impl block keeps functionality centralized, enhancing maintainability.

**Methods Sharing Names with Fields**

Rust permits methods to have the same name as struct fields. The compiler differentiates between them based on whether parentheses are used (rect.width() vs. rect.width). Such methods are often used as getters, especially when exposing read-only access to private fields.

A computer screen shot of a code

AI-generated content may be incorrect.

**Multiple Parameters**

Methods can accept additional parameters beyond self, as demonstrated in a can\_hold method that takes another Rectangle by reference:

A black and white text

AI-generated content may be incorrect.

**Associated Functions**

Functions within an impl block that do not take self are called associated functions. They are often used as constructors and are called with the :: syntax (e.g., Rectangle::square(10)). These do not require an instance to be called.

A screen shot of a computer code

AI-generated content may be incorrect.

Associated functions that aren’t methods are often used for constructors that will return a new instance of the struct. These are often called new, but new isn’t a special name and isn’t built into this language.

**Multiple impl Blocks**

Rust allows defining multiple impl blocks for the same type. This is syntactically valid and can be useful for separating logically distinct sets of functionality, such as trait implementations.

A computer screen shot of text

AI-generated content may be incorrect.

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