5_Structures_en

May 2, 2023

1 Structures

A struct, or structure, is a custom data type that lets you name and package together multiple related values that make up a meaningful group. Structs and enums are the building blocks for creating new types in your program's domain to take full advantage of Rust's compile time type checking.

Like tuples, the pieces of a struct can be different types. Unlike with tuples, you'll 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: you don't have to rely on the order of the data to specify or access the values of an instance.

[ ]:
```
fn main() {
    struct User {
        username: String,  // this is a field
        email: String,
        sign_in_count: u64,
        active: bool,
    }

    let user1 = User { // this is an instance of the "User" struct
        email: String::from("someone@example.com"),
        username: String::from("someusername123"),
        active: true,
        sign_in_count: 1,
    };
}
main();
```

To get a specific value from a struct, we can use dot notation. If the instance is mutable, we can change a value by using the dot notation and assigning into a particular field.

> **Note:** the entire instance must be mutable; Rust doesn't allow us to mark only certain fields as mutable.

[ ]:
```
fn main() {
    struct User {
        username: String,  // this is a field
        email: String,
```

```
        sign_in_count: u64,
        active: bool,
    }

    let mut user1 = User { // <-- user1 is mutable
        email: String::from("someone@example.com"),
        username: String::from("someusername123"),
        active: true,
        sign_in_count: 1,
    };
    // mutate email in user1
    user1.email = String::from("anotheremail@example.com");
}
main();
```

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.

[ ]:
```
fn main() {
    struct User {
        username: String,  // this is a field
        email: String,
        sign_in_count: u64,
        active: bool,
    }

    fn build_user(email: String, username: String) -> User {
        User { // <-- last expression of  the build_user function
            email: email,
            username: username,
            active: true,
            sign_in_count: 1,
        }
    }
}
main();
```

When the parameter names and the struct field names are exactly the same in this example, 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.

[ ]:
```
fn main() {
    struct User {
        username: String,  // this is a field
        email: String,
        sign_in_count: u64,
        active: bool,
    }
```

```
    fn build_user(email: String, username: String) -> User {
        User {
            email,        // <-- shorthand instead of email: email
            username,     // <-- same
            active: true,
            sign_in_count: 1,
        }
    }
}
main();
```

1.1 Creating instances from other instances

To create an instance from another instance without struct update syntax:

[ ]:
```
fn main() {
        struct User {
                username: String,  // this is a field
                email: String,
                sign_in_count: u64,
                active: bool,
        };

        let user1 = User {
                email: String::from("someone@example.com"),
                username: String::from("someusername123"),
                active: true,
                sign_in_count: 1,
        };
        let user2 = User { // without the struct update syntax
                email: String::from("another@example.com"),
                username: String::from("anotherusername567"),
                active: user1.active,
                sign_in_count: user1.sign_in_count,
        };
        println!("user2: {}", user2.username);
        let user3 = User { // update syntax
                email: String::from("yet_another@example.com"),
                username: String::from("username456"),
                ..user1  // <-- rest of the members will be taken from user1
        };
        println!("user3: {}", user3.username);
}
main();
```

1.2 Using Tuple Structs without Named Fields to Create Different Types

You can also define structs that look similar to tuples, called tuple structs. Tuple structs have the added meaning the struct name provides but don't have names associated with their fields; rather, they just have 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, and naming each field as in a regular struct would be verbose or redundant.

[ ]:
```
fn main() {
    struct Color(i32, i32, i32);
    struct Point(i32, i32, i32);

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

    println!("black = ({}, {}, {})", black.0, black.1, black.2);
}
main();
```

Note that the `black` and `origin` values are different types, because they're instances of different tuple structs. Each struct you define is its own type, even though the fields within the struct have the same types. Tuple struct instances behave like tuples: you can destructure them into their individual pieces, you can use a `.` followed by the index to access an individual value, and so on.

1.3 Defining *methods*

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).

When a method need to refer to the instance of the struct, their first parameter is always `self`, which represents the instance of the struct the method is being called on.

[ ]:
```
fn main() {
    struct Rectangle {
        width: u32,
        height: u32,
    }

    impl Rectangle {
        fn area(&self) -> u32 {
            self.width * self.height
        }
    }

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

```
    println!(
        "The area of the rectangle is {} square pixels.",
        rect1.area()
    );
}
main();
```

`&self` is used as a parameter because we don't want to take ownership, and we just want to read the data in the struct, not write to it. 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. Ownership is discussed later in another chapter.

It is possible to define methods without a `self` parameter, in which case they act as static methods for the type.

In fact, this is exactly what is used in Rust as a convention for constructors. Did you notice that until this point we only used explicit construction of structures? The convention in Rust is to define a method named `new` that takes adequate arguments and return the initialized structure :

> Note that despite being named `new`, the function has no special meaning in Rust. There is no `new` keyword. Using `new` as the name of a function to build an instance is a convention.

[ ]:
```
fn main() {
    struct Rectangle {
        width: u32,
        height: u32,
    }

    impl Rectangle {
        /// this is our constructor, the name new is nothing special
        fn new(width: u32, height: u32) -> Rectangle {
            Rectangle { width, height }
        }

        fn area(&self) -> u32 {
            self.width * self.height
        }
    }

    let rect1 = Rectangle::new(30, 50);

    println!(
        "The area of the rectangle is {} square pixels.",
        rect1.area()
    );
}
main();
```

2 Exercises

Exercice 1: Re-write the following code using struct.

[ ]:
```
let width: u32 = 30; // the width of the rectangle
let height: u32 = 50; // the height of the rectangle
println!("The width of the rectangle is {} and the height is {}", width,␣
 ↪height);
```

[ ]:
```
// TO DO (1) Create a structure called Rectangle

// TO DO (2) Initialize the structure


// TO DO (3) Display the width and height
```

Solution:

[ ]:
```
fn main() {
    struct Rectangle {
        width: u32,
        height: u32,
    }

    impl Rectangle {
        fn new(width: u32, height: u32) -> Rectangle {
            Rectangle { width, height }
        }

        fn display(&self) {
            println!("The width of the rectangle is {} and the height is {}",␣
 ↪self.width, self.height);
        }
    }

    let rect1 = Rectangle::new(30, 50);
    rect1.display();
}
main();
```

What if we want to display the instance of a structure? (could be useful for debugging)

[ ]:
```
struct Rectangle {
    width: u32,
    height: u32,
}

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

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

The `println!` macro can do many kinds of formatting, and by default, the curly brackets tell 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 brackets? 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`.

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.

[ ]:
```
fn main() {
    #[derive(Debug)]
    struct Rectangle {
        width: u32,
        height: u32,
    }


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

    println!("rect1 is {:?}", rect1); // tell println! to use an output format␣
 ↪called Debug
}
```

Putting the specifier **:?** inside the curly brackets tells `println!` we want to use an output format called `Debug`. The Debug **trait** enables us to print our struct in a way that is useful for developers so we can see its value while we're debugging our code.

> ***Note:*** *Traits are similar to a feature often called interfaces in other languages, although with some differences.*

When we have larger structs, it's useful to have output that's a bit easier to read; in those cases, we can use **{:#?}**.

Exercise 2: To see the result when using {:#?}, try it on the previous example.

Exercise 3: Define a structure *GeoCoordinate* containing a geographical *latitude* and *longitude* and implement the *to_degrees()* and *to_radians()* methods that transforms coordinates into degrees and radians respectively. Then define a GeoBoundingBox structure containing the min and max coordinates of a bounding box. Test your implementation.

Sample test data: latitude: 44.84, longitude: -0.58

> **Tips:** Open the documentation pages for the f64 primitive type, and look for the `to_degrees` and `to_radians` methods. https://doc.rust-

lang.org/std/primitive.f64.html#method.to_degrees

[ ]:
```
// TO DO (1) Define the GeoCoordinate structure

// TO DO (2) Implement for the GeoCoordinate structure the to_degrees and␣
 ↪to_radians methods

fn main() {
    // TO DO (3) Test your implementation

}
main();
```

Solution:

[ ]:
```
#[derive(Debug)]
struct GeoCoordinate {
    /// The latitude
    latitude: f64,
    /// The longitude
    longitude: f64
}

impl GeoCoordinate {
    /// Transforms to values in degrees
    fn to_degrees(&self) -> GeoCoordinate {
        GeoCoordinate {
            latitude: self.latitude.to_degrees(),
            longitude: self.longitude.to_degrees()
        }
    }

    /// Transforms to velues in radians
    fn to_radians(&self) -> GeoCoordinate {
        GeoCoordinate {
            latitude: self.latitude.to_radians(),
            longitude: self.longitude.to_radians()
        }
    }
}

fn main() {
    let x = GeoCoordinate {latitude: 44.84, longitude: -0.58}; // coordonnées␣
 ↪de la ville de Bordeaux
    println!("{:#?}", x.to_degrees());
}
main();
```