Structs

Mustakimur R. Khandaker

Structs

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.

- it is like an object's data attributes (OOP).
- Very similar to tuples.
  - Unlike tuples, we can name each piece of data.
  - Unlike tuples, we can define functions of struct.

To define a struct, we enter the keyword struct and name the entire struct.

- Then, inside curly brackets, define the names and types of the pieces of data.
  - which we call fields.

We create an instance of a struct by specifying concrete values for each of the fields.

To get a specific value from a struct, we can use dot notation.

- To able to change a field of the struct, the entire instance must be mutable.

Code

```
struct User {
    username: String,
    email: String,
    sign_in_count: u64,
    active: bool,
}
```

```
fn main() {
    code1();

    let email = String::from("anemail@example.com");
    let username = String::from("anotheruser");
    code2(email, username);
}
```

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

```
fn code1() {
    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");
}
```

```
fn code2(email: String, username: String) -> User {
    User {
        email,
        username,
        active: true,
        sign_in_count: 1,
    }
}
```

Creating Instances From Other Instances

```rust
fn code3(user1: User) {
    let user2 = User {
        email: String::from("another@example.com"),
        username: String::from("anotherusername567"),
        active: user1.active,
        sign_in_count: user1.sign_in_count,
    };
    println!("{:#?}", user2);
}
```

```rust
fn code3(user1: User) {
    let user2 = User {
        email: String::from("another@example.com"),
        username: String::from("anotherusername567"),
        ..user1
    };
    println!("{:#?}", user2);
}
```

Tuple alike Struct

Define structs that look similar to tuples, called ***tuple structs***.

- They are useful when we want to
  - give the whole tuple a name
  - make the tuple be a different type from other tuples
  - naming each field as in a regular struct would be verbose or redundant.

```rust
struct Color(i32, i32, i32);
struct Point(i32, i32, i32);

fn code4() {
    let black = Color(0, 0, 0);
    let origin = Point(0, 0, 0);
}
```

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.

- we can use a . followed by the index to access an individual value.

Ownership of Struct Data

In all example, we have use String type in the field.

- Because, we want instances of the 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.

Lifetimes ensure that the data referenced by a struct is valid for as long as the struct is.

```rust
struct UserSTR {
    username: &str,
    email: &str,
    sign_in_count: u64,
    active: bool,
}
```

```rust
fn code5() {
    let user1 = User {
        email: "someone@example.com",
        username: "someusername123",
        active: true,
        sign_in_count: 1,
    };
}
```

Debug Structs

We can print an instance of any Struct while we're debugging our program and see the values for all its fields.

- Rust does include functionality to print out debugging information, but we have to explicitly opt in to make that functionality available.
- For Struct debug, add the annotation `#[derive(Debug)]` just before the struct definition.
- In println!(), use either {:?} or {:#?} to print formatted struct instance.

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

```
fn code6() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };

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

Method Vs Function

Methods are similar to functions:

- they’re declared with the fn keyword and their name, they can have parameters and a return value.
- they contain code that is run when they’re called.

However, methods are different from functions:

- they’re defined within the context of a struct (or an enum or a trait object).
- their first parameter is always self, which represents the instance of the struct the method is being called on.

We create a block of code using impl to define methods for a struct.

- Struct name following impl keyword.
- Anywhere in the source.
- Any number of time in the source.

Use of Methods

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

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

```rust
fn code7() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };

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

We use &self instead of rectangle: &Rectangle.

- because Rust knows the type of self is Rectangle.
    - due to the method's being inside the impl Rectangle context.
- Methods can take ownership of self, borrow self immutably (in this example), or borrow self mutably.

Self: Best Practices

&self

- We don’t want to transfer ownership of the instance.
- We don’t require to modify the instance of itself.

&mut self

- We don’t want to transfer ownership of the instance.
- We do require to modify the instance of itself.

self

- Stands for transfer of the ownership of the instance to the method.
    - Unlikely to ever need it.
    - Second transfer of ownership will be required to keep it alive.

Automatic Referencing and Dereferencing

In C and C++, two different operators are used for calling methods. We use:

- . if we're calling a method on the object directly
- -> if we're calling the method on a pointer to the object and need to dereference the pointer first.

When we call a method with object.something(), Rust automatically adds in &, &mut, or * so object matches the signature of the method.

p1.distance(&p2); equivalence to (&p1).distance(&p2);

Instance as Parameter

We can send more than one parameter to the method besides the self.

- It includes another instance of the struct.

```
impl Rectangle {
    fn can_hold(&self, other: &Rectangle) -> bool {
        self.area() > other.area()
    }
}
```

```
fn code8() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };
    let rect2 = Rectangle {
        width: 10,
        height: 40,
    };

    println!("Can rect1 hold rect2? {}", rect1.can_hold(&rect2));
}
```

Associate Functions

Associated functions are function not method although they are defined within an impl block.

- Difference: They don’t take self as parameter.
- They act similar to constructor of C++ class.
    - So, it will return the instance of the struct.

Remember, we have used String::from(“TEXT”) to create instance of String type. From(“TEXT”) is an associate function of the Struct String.

- To call this associated function, we use the :: syntax with the struct name.

```rust
impl Rectangle {
    fn square(size: u32) -> Rectangle {
        Rectangle {
            width: size,
            height: size,
        }
    }
}
```

```rust
fn code9() {
    let sq = Rectangle::square(3);
    let area = sq.area();
    println!("Rectangle area: {}", area);
}
```

< Structs />