Lecture 9

Structs II

What we will cover today

Structs in Rust

- Methods
- Associated types
- Debug trait

Optional Reading:

The Rust Book Chapter 5.2, 5.3 – Method Syntax

Review: What are Structs?

- Short for “structure”
- Composite type – Holds multiple related values
- Similar concept to objects in OOP languages like Java
- Structs vs tuples
    - In a struct you can name each piece of data

Defining structs

```rust
struct Car {
    brand: String,
    make: String,
    mpg: i32,
}
```

fields

Keyword: `struct`

In each field:

- name followed by type

Instantiating structs

```
struct Car {
    brand: String,
    make: String,
    mpg: i32,
}
```

```
let my_car = Car {
    brand: String::from("Toyota"),
    make: String::from("Camry"),
    mpg: 30,
};
```

To instantiate a struct, we specify a concrete value for each field

Rectangle

```rust
struct Rectangle{
    width: u32,
    height: u32,
}
```

```rust
let rect = Rectangle{
    width: 50,
    height: 100
};
```

How do we calculate the area?

```rust
let area = rect.width * rect.height;
```

Rectangle - Problems

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

```
let rect = Rectangle{
   width: 50,
   height: 100
};
```

```
let area = rect.width * rect.height;
```

What if we have a more complex function? E.g. finding angle of diagonal

- Code repetition
- Different code for different instances

Rectangle - Problems

```rust
struct Rectangle{
    width: u32,
    height: u32,
}
```

```rust
let rect = Rectangle{
    width: 50,
    height: 100
};
```

```rust
let area = rect.width * rect.height;
```

What if we have a more complex function? E.g. finding angle of diagonal
Ans: Write it into a function!

Rectangle - A simple function

```rust
fn rect_area(rect: &Rectangle) -> u32 {
    rect.width * rect.height
}
...
let area = rect_area(&rect);
```

What if we have a more complex function? E.g. finding angle of diagonal
Ans: Write it into a function!

Methods!

Methods are functions that are defined within the context of a struct

- Helps with organization!

```
let rect = Rectangle{ ... };

let area = rect_area(&rect); // Normal function syntax

let area2 = rect.rect_area(); // Method syntax
```

How to write methods

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

Keyword: `impl <type>`

How to write methods

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

Method takes in a “self” parameter → This is the specific instance!

&self ⇐⇒ rect: &Rectangle

&self is a shorthand!

Methods and ownership

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

For first parameter, method can take in:

- Reference
- Mutable reference
- Take ownership / Move

Methods - Mutable reference

```rust
impl Rectangle{
...
   fn change_width(&mut self, new_width: u32){
       self.width = new_width;
   }
...
}
```

This example: Takes in a mutable reference, modifies the instance!

Methods - Takes ownership

```
impl Rectangle{
...
    fn swap(self) -> Self{
        Rectangle { width: 42, height: 42 }
    }
...
}
```

This example: Takes ownership, returns a completely new instance

Methods - Moved into function

```
impl Rectangle{
...
   fn disappear(self){
       // Do nothing
   }
...
}
```

This example: Instance moved into function, no longer usable

Associated functions

What are associated functions?

- Any function defined in an impl
- Can write functions that don’t take in the self (thus are not methods)
- Tied to the type (e.g. Rectangle) rather than specific instances
- Rather like static methods in Java (“Belong to class rather than object of a class”)

Associated functions

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

Associated functions commonly used for constructors

Call `Rectangle::new(5, 5)`

Associated functions

```
impl Rectangle{
...
    fn description() -> String{
        String::from("I like parallelograms more...")
    }
...
}
```

Just an example...

Not methods vs methods

```
let desc = Rectangle::description(); // Assoc fn

let area = rect.area(); // Method
```

You call the functions differently

Not methods: <type>::<fn>
Method: <instance>.<fn>

Debug trait

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

Allows you to pretty-print out struct instances

Debug trait

```
println!("{:?}", rect);
...
Rectangle { width: 69, height: 100 }
```

```
println!("{:#?}", rect);
...
Rectangle {
    width: 69,
    height: 100,
}
```

Recap

Structs – Composite type, has named fields

Associated functions – Any functions defined in an `impl` block

Methods – Called on an instance, can take in {`&self, &mut self, self`}

Not methods – Called on the type, uses `Rectangle::new` syntax

Debug trait – `#[derive(Debug)]` – Lets you pretty print stuff

Announcements

HW7 released today on PrairieLearn
Due 1 week from now — Next Wed 03/05 23:59

MP1 was released on PrairieLearn last week
Due — Next Wed 03/05 23:59

MP2 coming soon!