Structs Part 2

Goals For Today

- Review some information about structs
- Look at some sample struct code

What is a struct?

- Structs are custom made data types that hold multiple values
- Similar to tuples, structs had hold data of multiple types without issue
  - Unlike tuples, you must declare the type of data that you are storing
- An Object of type struct is called an instance of that struct
- Structs are functionally similar to classes
  - Instances do not share variables
  - Can declare local functions using traits or as methods with impl
  - Declared prior to the methods they are used in

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

How do I create a struct?


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

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

Struct Methods

- Say we want to add functions to our structs
- How do we go about making a struct owned function?
  - Key word impl
  - Indicates the implementation of a function for a struct, can have multiple impl
  - Function accepts a borrowed instance to access data in the instance
  - All Methods in impl require you to pass &self irrespective of use

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

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

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

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

```rust
fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };
    let rect2 = Rectangle {
        width: 10,
        height: 40,
    };
    let rect3 = Rectangle {
        width: 60,
        height: 45,
    };

    println!("Can rect1 hold rect2? {}", rect1.can_hold(&rect2));
    println!("Can rect1 hold rect3? {}", rect1.can_hold(&rect3));
}
```

Associated functions

- The only exception to needing to pass &self as a parameter are associated functions
- These are functions not called by the dot notation but with :: instead
- Typically used for constructors are demonstrated below
- Self is an alias for Rectangle in this case, rather than an instance

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

Tuple Structs

- Functionally the same as a tuple and can access data in the same way (dot notation)
- Mutable
- Allows you to distinguish tuples of interest

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

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

Reminders:

Chapter 5 in the rust docs: Using Structs to Structure Related Data - The Rust Programming Language (rust-lang.org)

HW 7 has been released

MP2 us being released soon. It will be focusing on Structs and Object Oriented Programming Concepts

Review Lecture 8 for a much more deeper dive into Struct theory and how Structs work