5. Using Structs in Rust

Structs offer a great way to structure data in your code. In this chapter, we will discuss what they are, how they can be used and why their usage is crucial to writing good code.

Structs in C / Python / Rust

You might already know the word `struct` from other languages like C. In fact, the structs in C are quite similar to those in Rust. In object-oriented languages like Python or Java, a corresponding data structure to the structs in Rust are Classes. Rust structs can also be used in an object-oriented fashion, as we will see later.

Defining Structs

Defining structs is really quite simple. For example, we can write a struct “Person” as seen in 1. Inside of the struct, one can put variables, separated by a comma. One has to annotate the type of the variables, since the compiler has to know the size of the struct.

```rust
struct Person {
  age: i32,
  firstname: String,
  lastname: String,
}
```

Illustration 1: A struct in Rust

Instantiating Structs

There are several methods to instantiate a previously defined struct. The first (and most straightforward) method can be seen in 2: the wanted values are directly given during the instantiation.

```rust
fn main() {
  let p1 = Person {
    age: 25,
    firstname: String::from("Max"),
    lastname: String::from("Mustermann"),
  };
}
```

Illustration 2: Direct instantiating

Another, in many cases more clean solution can be seen in 3. Here, a function creates the new Instance and returns it to the caller. This is already quite close to a constructor known from languages like Java. In this case, the variable `age` can be passed to the struct directly, since it has the same name in the function parameter. This cleans up the code a lot.

```rust
fn create_person(age: i32) -> Person {
  Person {
    age,
    firstname: String::from("Max"),
    lastname: String::from("Mustermann"),
  }
}
```

Illustration 3: Instantiating function

Tuple Structs

Tuple structs are something between a tuple and a struct. An example can be seen in . They can be used in the same fashion like normal structs; however, their variables can be accessed like you would with a tuple (e.g. `origin.1` for the x Coordinate of `origin`).

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

fn main() {
  let red = Color(255, 0, 0);
  let origin = Coordinates(0, 0);
}
```

Illustration 4: Tuple Structs

The impl block

For now, the Rust structs acted the same as a struct in C would. Using impl blocks, a more object-oriented approach can be taken. Impl blocks can be used for different things. In this chapter, we will take a look at how they can add constructors and methods to an object; however, they can also be used to implement so-called `traits`, which will be discussed in a later lecture.

In 5, you can see an example for an impl block. There are a few things to note here:

```
struct Person {
  age: i32,
  firstname: String,
  lastname: String,
}

impl Person {
  fn new(age: i32, firstname: String, lastname: String) -> Person {
    Person { age, firstname, lastname }
  }
  fn print(&self) {
    println!("Ich heiße {} {} und bin {} Jahre alt.",
      self.firstname, self.lastname, self.age);
  }
}

fn main() {
  let p1 = Person::new(21, String::from("Max"), String::from("Mustermann"));
  p1.print();
}
```

Illustration 5: A impl block for the struct Person

Firstly, methods for the objects of type `Person` are defined just like normal functions. However, as you can see in the `print()` method, methods have the parameter `self`, while functions (such as `new()` or `main()`) don't have it. This parameter tells the compiler which object the method was called on. Variables can be accessed inside methods using `self.var_name`.

The constructor `new()` is just a function inside the impl block, returning a new object. This is quite similar to the example we already saw in 3.

Sources

- All information is taken from *The Rust Programming Language*, chapter 5, found at https://doc.rust-lang.org/book. I adjusted the code examples.
- Images are screenshots taken by myself.