Using Structs to Structure Related Data

Defining and instantiating structs

Struct allows you to compose different type of data together into one big object, just like structs in C.

You will have to name each piece of data that you are using so that you can access them when you instantiate a struct later.

Here is how to define a sample struct (Note you would write this outside of functions):

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

Then to instantiate a struct:

```
fn main() {
    let user1 = User {
        active: true,
        username: String::from("Ricky"),
        email: String::from("irebo@gmail.com"),
    };
}
```

To access the fields that you have instantiated you would use the dot notation, `user1.active, user1.username, user1.email`.

To make the struct mutable you would also attach the `mut` modifier to the variable. The entire instance of struct must be mutable, Rust doesn't allow partial field mutability.

Field init shorthand

Say you have a function that builds your struct and return it as its return value depending on the parameter you passed:

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

Writing it like this will get repetitive, especially if there are going to be lot of parameter, instead you can use a shorthand, just ignore the key if the parameter that you passed into the function is the same as the key name:

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

This is much more concise without the repetition.

Creating instances from other instances with struct update syntax

Sometimes you might want to create a new instances from the old instances, changing some of the old values but keep the rest the same.

You can do it the hard coded way like such:

```
fn main() {
// created user1 here

    let user2 = User {
    active: user1.active,
        username: user1.username,
        email: String::from("new email@gmail.com"),
```

```
};
}
```

This works but you have to type out all of the fields that are repeated, a much shorter way to do this is to use struct update syntax:

```
fn main() {
// create user1 here

    let user2 = User {
    email: String::from("new email here"),
        ..user1
}
}
```

With this syntax, you only have to worry about writing the new value for the new instance, and leave all of the old values to struct update syntax to handle.

`..user1` must come last to specify that any remaining fields should be getting their values from the corresponding fields in `user1`.

> struct update syntax uses `=` like an assignment, so it will be moving data. After doing struct update you can no longer use `user1` as a whole after creating `user2` since data like `username` is moved to `user2` and not copied!

Tuple struct

You can also create a tuple struct, which is like struct but doesn't have names associated with their fields, they only have type of the fields.

This is useful if you just want to give a simple tuple a name. And separate different type of tuple from each other.

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

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

You can access the tuple using the same tuple syntax, `tuple.<index number>`

Method syntax

Methods are like function but they are defined in the context of a struct, enum, or a trait object.

The first parameter of a method is always self, which represents the instance of the struct that the method is called on.

Defining methods

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

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

fn main() {
    let r1 = Rectangle {
        width: 30,
        height: 50,
    };

    println!("The area is {}", r1.area());
}
```

To define function in the context of in this case a struct, you write the `impl` block for `Rectangle`. Everything inside this `impl` block will be associated with the `Rectangle` type. Then you can write the method itself, making sure that the first parameter is a reference to `self`.

Then you can call the method on the object.

The first parameter must be `&self` and is actually a shorthand for `self: &Self`, `Self` type itself is an alias for the type that the `impl` block is for. This is so that we don't have to write `rectangle: &Rectangle` instead.

Method can take ownership of `self`, borrow self immutable or mutably. In the example, it is just borrowing self immutably since it is just reading data.

>

> Just for your information, no matter which `self` you do, taking ownership or do borrowing, Rust will automatically add the appropriate `&, &mut, or *` for you automatically. This is so that you can just focus on calling the method on the object without worrying about anything else

```
p1.ditance(&p2);
// vs
(&p1).distance(&p2);
```

Associated functions

Functions implemented with `impl` block are called assocaited functions because they are connected to the type of struct they are defined. You are able to define associated functions that don't have `self` as first parameter (hence they are no longer methods) they are reference to as static or class methods. They are associated with the type rather than an instance.

Associated functions that don't have `self` are often used for constructors that return new instance of the struct, just like `String::from`.

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

> `Self` in this case is an alias for the type that appears after the `impl` keyword, `Rectangle` in this case.

Then to call associated functions that doesn't take `self` as first parameter you use the `::` syntax with the struct name like such:

```
let sq = Rectangle::square(3);
```

Multiple impl block

You can separate different methods into `impl` blocks, although there is really no reason to unless for readability.

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

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

This is perfectly valid syntax.

Revision #2
Created 2023-01-28 22:04:09 UTC by Tamarine
Updated 2023-01-29 02:49:08 UTC by Tamarine