Rust - Day 6

Structs

> Similar to custom types in typescript

- **A `struct` or `structure` is a custom data type that lets you package together and name multiple related values that make up a meaningful group.**
- **It is similar to an object's data attributes in object-oriented languages**
- We'll compare and contrast tuples with structs to build on what you already know and demonstrate when structs are a better way to group data.

Defining and Instantiating Structs

- Structs are similar to tuples as both hold multiple related values.
- The pieces of struct can be of different types, but **you'll name each piece of data in struct, so it's clear what the values mean.**
- Adding names makes structs more flexible than tuples as you don't have to rely on the order of data to specify or access values.

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

- **Each piece of data (name: type) is known as a `field`**
- **To use a struct, after defining, we create an instance of it by specifying concrete values for each of the fields.**

```
fn main() {
    let user1 = User {
        active: true,
        username: String::from("ayroid"),
        email: String::from("ayroid@random.com"),
        sign_in_count: 1,
    };
}
```

- The fields **need not be in any specify order.**

> Struct is a general template, instance fill in the template to create values of the type.

- **To access the values from a struct, use dot notation.**
- **Mutable instance field values can also be modified using dot notation**
- The complete instance must be mutable to modify values, rust does not allow setting fields as mutable.

```
fn main() {
    let mut user1 = User {
        active: true,
        username: String::from("ayroid"),
        email: String::from("ayroid@random.com"),
        sign_in_count: 1,
    };
    user1.email = String::from("ayroid@notrandom.com");
}
```

- Just like any expression, we can construct a new instance of the struct as the last expression in the function body to implicitly return the new instance.

```
fn build_user(emai: String, username: String) → User {
    User {
        active: true,
        username: username,
        email: email,
        sign_in_count: 1,
    }
}
```

- If the parameter names and the struct field names are exactly the same, then we can user the `field init shorthand` syntax to rewrite it.

```
fn build_user(emai: String, username: String) → User {
    User {
        active: true,
        username,
        email,
        sign_in_count: 1,
    }
}
```

Creating Instances from Other instances with Struct Update Syntax

- It's often useful to create a new instance of a struct that includes most of the values from another instance, but changes some. You can do this using *struct update syntax*.
- This below code is without the *struct update syntax*

```
fn main() {
    // --snip--

    let user2 = User {
        active: user1.active,
        username: user1.username,
        email: String::from("another@example.com"),
        sign_in_count: user1.sign_in_count,
    };
}
```

- Using the *struct update syntax*, we can achieve the same with lesser code.

> Similar to spread operator in JS and TS

```
fn main() {
    // --snip--

    let user2 = User {
        email: String::from("another@example.com"),
        ..user1
    }
}
```

- **❗ The struct update syntax uses '=' like an assignment i.e. it moves the data. In this example, we can no longer user `user1` as a whole after creating `user2` because the `String` in the `username` field of `user1` was moved to `user2`.**
- **If we had just moved `active` and `sign_in_count` then there would be no problem and `user1` would still be valid after creating `user2` as both `active` and `sign_in_count` are types that implement the `Copy` trait**

Using tuple structs without named fields to create different types

- Rust also supports structs that look similar to tuples, called *tuple structs*.
- Tuple structs don't have names associated with their fields, rather they just have the types of the fields.
- **Tuple structs are useful when you want to give the whole tuple a name and make the tuple a different type from other tuples, and when naming each field as in a regular struct would be verbose or redundant.**

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

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

- Note that the `black` and `origin` values are different types because they're instances of different tuple structs. Each struct you define is its own type, even though the fields within the struct might have the same types.
- **For example, 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.**
- **Otherwise, tuple struct instances are similar to tuples in that you can destructure them into their individual pieces, and you can use a `.` followed by the index to access an individual value.**

Unit-Like structs without any fields ❗

- These are called *unit-like structs* because they behave similarly to `()`, the unit type.
- Unit-like structs can be useful when you need to implement a trait on some type but don't have any data that you want to store in the type itself. **This would be explained in Chapter 10**

```
struct AlwaysEqual;

fn main() {
    let subject = AlwaysEqual;
}
```

Ownership of Struct Data

- In the `User` struct definition, we used the owned `String` type rather than the `&str` string slice type. This is a deliberate choice because we want each instance of this 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 also possible for structs to store references to data owned by something else, but to do so requires the use of *lifetimes*, a Rust feature that we'll discuss in Chapter 10. Lifetimes ensure that the data referenced by a struct is valid for as long as the struct is. Let's say you try to store a reference in a struct without specifying lifetimes, like the following; this won't work:

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

fn main() {
    let user1 = User {
        active: true,
        username: "someusername123",
        email: "someone@example.com",
```

```
        sign_in_count: 1,
    };
}
```

```
$ cargo run
   Compiling structs v0.1.0 (file:///projects/structs)
error[E0106]: missing lifetime specifier
 --> src/main.rs:3:15
  |
3 |     username: &str,
  |               ^ expected named lifetime parameter
  |
help: consider introducing a named lifetime parameter
  |
1 ~ struct User<'a> {
2 |     active: bool,
3 ~     username: &'a str,
  |

error[E0106]: missing lifetime specifier
 --> src/main.rs:4:12
  |
4 |     email: &str,
  |            ^ expected named lifetime parameter
  |
help: consider introducing a named lifetime parameter
  |
1 ~ struct User<'a> {
2 |     active: bool,
3 |     username: &str,
4 ~     email: &'a str,
  |

For more information about this error, try `rustc --explain E0106`.
error: could not compile `structs` (bin "structs") due to 2 previous errors
```

- In Chapter 10, we'll discuss how to fix these errors so you can store references in structs, but for now, we'll fix errors like these using owned types like `String` instead of references like `&str`.

Writing a program using Structs

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

fn main() {
    // Default way

    let width1: u32 = 30;
    let height1: u32 = 50;

    println!(
        "The area of the rectangle is {} square pixels.",
        area(width1, height1)
    );

    // Using tuple
    // Tuple is a way to group together a number of values with a variety of types into one compound type.
    // This organizes data in a way that is easy to access and manipulate. but it does not name each piece of data, making it less readable and harder to work with.

    let rect1: (u32, u32) = (30, 50);
```

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

    // Using struct
    // Structs are a way to create more complex data types. They allow you to name and package together multiple related values that make up a meaningful group.
    // Structs are similar to tuples. Like tuples, the pieces of a struct can be different types. Unlike with tuples, you'll name each piece of data so it's clear what the values mean.

    let rect2 = Rectangle {
        width: 30,
        height: 50,
    };

    // using & we are passing a reference to the struct instead of taking ownership of it.

    println!(
        "The area_struct of the rectangle is {} square pixels.",
        area_struct(&rect2)
    );
}

fn area(width: u32, height: u32) -> u32 {
    width * height
}

fn area_tuple(dimensions: (u32, u32)) -> u32 {
    dimensions.0 * dimensions.1
}

fn area_struct(rectangle: &Rectangle) -> u32 {
    rectangle.width * rectangle.height
}
```

Adding useful functionality with derived traits

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

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

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

- When we compile this code, we get an error with this core message:

```
error[E0277]: `Rectangle` doesn't implement `std::fmt::Display`
```

- The `println!` macro can do many kinds of formatting, and by default, the curly brackets tell `println!` to use formatting known as `Display`: output intended for direct end user consumption. The primitive types we've seen so far implement `Display` by default because there's only one way you'd want to show a `1` or any other primitive type to a user. But with structs, the way `println!` should format the output is less clear because there are more display possibilities: Do you want commas or not? Do you want to print the curly brackets? Should all the fields be shown? Due to this ambiguity, Rust doesn't try to guess what we want, and structs don't have a provided implementation of `Display` to use with `println!` and the `{}` placeholder.

```
  = help: the trait `std::fmt::Display` is not implemented for `Rectangle`
  = note: in format strings you may be able to use `{:?}` (or {:#?} for pretty-print) instead
```

- The `println!` macro call will now look like `println!("rect1 is {rect1:?}");`. Putting the specifier `:?` inside the curly brackets tells `println!` we want to use an output format called `Debug`. The `Debug` trait enables us to print our struct in a way that is useful for developers so we can see its value while we're debugging our code.

```
error[E0277]: `Rectangle` doesn't implement `Debug`
```

```
  = help: the trait `Debug` is not implemented for `Rectangle`
  = note: add `#[derive(Debug)]` to `Rectangle` or manually `impl Debug for Rectangle`
```

- ❗ **Rust *does* include functionality to print out debugging information, but we have to explicitly opt in to make that functionality available for our struct. To do that, we add the outer attribute `#[derive(Debug)]` just before the struct definition**

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

fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };
    println!("rect1 is {rect1:?}");
}
```

- This would work without any erros

```
$ cargo run
   Compiling rectangles v0.1.0 (file:///projects/rectangles)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.48s
     Running `target/debug/rectangles`
rect1 is Rectangle { width: 30, height: 50 }
```

- **When we have larger structs, it's useful to have output that's a bit easier to read; in those cases, we can use `{:#?}` instead of `{:?}` in the `println!` string. In this example, using the `{:#?}` style will output the following:**

```
$ cargo run
   Compiling rectangles v0.1.0 (file:///projects/rectangles)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.48s
     Running `target/debug/rectangles`
rect1 is Rectangle {
    width: 30,
    height: 50,
}
```

- Another way to print out a value using the `Debug` format is to use the [`dbg!` macro](), which takes ownership of an expression (as opposed to `println!`, which takes a reference), prints the file and line number of where that `dbg!` macro call occurs in your code along with the resultant value of that expression, and returns ownership of the value.

> Note: Calling the `dbg!` macro prints to the standard error console stream (`stderr`), as opposed to `println!` , which prints to the standard output console stream (`stdout`).

- Here's an example where we're interested in the value that gets assigned to the `width` field, as well as the value of the whole struct in `rect1`:

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

fn main() {
    let scale = 2;
    let rect1 = Rectangle {
        width: dbg!(30 * scale),
        height: 50,
    };

    dbg!(&rect1);
}
```

```
$ cargo run
   Compiling rectangles v0.1.0 (file:///projects/rectangles)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.61s
     Running `target/debug/rectangles`
[src/main.rs:10:16] 30 * scale = 60
[src/main.rs:14:5] &rect1 = Rectangle {
    width: 60,
    height: 50,
}
```

- **The `dbg!` macro can be really helpful when you're trying to figure out what your code is doing!**

Method Syntax

- Similar to functions in naming and declaration. Unlike functions, methods are 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.

Defining methods

- Let's change the `area` function that has a `Rectangle` instance as a parameter and instead make an `area` method defined on the `Rectangle` struct

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

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

fn main() {
```

```
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };

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

- To define the function within the context of `Rectangle`, we start an `impl` (implementation) block for `Rectangle`.
- Everything within this `impl` block will be associated with the `Rectangle` type.
- Then we move the `area` function within the `impl` curly brackets and change the first (and in this case, only) parameter to be `self` in the signature and everywhere within the body.
- In `main`, where we called the `area` function and passed `rect1` as an argument, we can instead use *method syntax* to call the `area` method on our `Rectangle` instance. The method syntax goes after an instance: we add a dot followed by the method name, parentheses, and any arguments.
- **In the signature for `area`, we use `&self` instead of `rectangle: &Rectangle`. The `&self` is actually short for `self: &Self`. `Self` is like `this` keyword in other languages.
- Methods must have a parameter named `self` of type `Self` for their first parameter, so Rust lets you abbreviate this with only the name `self` in the first parameter spot.
- **Note that we still need to use the `&` in front of the `self` shorthand to indicate that this method borrows the `Self` instance, just as we did in `rectangle: &Rectangle`. Methods can take ownership of `self`, borrow `self` immutably, as we've done here, or borrow `self` mutably, just as they can any other parameter.**
- We chose `&self` because we don't want to take ownership, and we just want to read the data in the struct, not write to it. If we wanted to change the instance that we've called the method on as part of what the method does, we'd use `&mut self` as the first parameter.
- The main reason for using methods instead of functions, in addition to providing method syntax and not having to repeat the type of `self` in every method's signature, is for organization.
- **We've put all the things we can do with an instance of a type in one `impl` block rather than making future users of our code search for capabilities of `Rectangle` in various places in the library we provide.**

```
impl Rectangle {
    fn width(&self) -> bool {
        self.width > 0
    }
}

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

    if rect1.width() {
        println!("The rectangle has a nonzero width; it is {}", rect1.width);
    }
}
```

- Here, we're choosing to make the `width` method return `true` if the value in the instance's `width` field is greater than `0` and `false` if the value is `0`: we can use a field within a method of the same name for any purpose.
- **In `main`, when we follow `rect1.width` with parentheses, Rust knows we mean the method `width`. When we don't use parentheses, Rust knows we mean the field `width`.**

- **Often, but not always, when we give a method the same name as a field we want it to only return the value in the field and do nothing else. Methods like this are called *getters*, and Rust does not implement them automatically for struct fields as some other languages do.**
- **Getters are useful because you can make the field private but the method public**, and thus **enable read-only access** to that field as part of the type's public API.

Automatic Referencing and Dereferencing

- In C and C++, two different operators are used for calling methods: you use `.` if you're calling a method on the object directly and `->` if you're calling the method on a pointer to the object and need to dereference the pointer first. In other words, if `object` is a pointer, `object->something()` is similar to `(*object).something()`.
- Rust doesn't have an equivalent to the `->` operator; instead, Rust has a feature called *automatic referencing and dereferencing*. Calling methods is one of the few places in Rust that has this behavior.
- Here's how it works: when you call a method with `object.something()`, Rust automatically adds in `&`, `&mut`, or `*` so `object` matches the signature of the method. In other words, the following are the same:

```
p1.distance(&p2);
(&p1).distance(&p2);
```

- The first one looks much cleaner. This automatic referencing behavior works because methods have a clear receiver—the type of `self`. Given the receiver and name of a method, Rust can figure out definitively whether the method is reading (`&self`), mutating (`&mut self`), or consuming (`self`). The fact that Rust makes borrowing implicit for method receivers is a big part of making ownership ergonomic in practice.

Methods with More Parameters

- Let's practice using methods by implementing a second method on the `Rectangle` struct. This time we want an instance of `Rectangle` to take another instance of `Rectangle` and return `true` if the second `Rectangle` can fit completely within `self` (the first `Rectangle`); otherwise, it should return `false`. That is, once we've defined the `can_hold` method, we want to be able to write the program

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

        let rect2 = Rectangle {
                width: 10,
                height: 40,
        };

        let rect4 = Rectangle {
                width: 60,
                height: 45,
        };

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

```
Can rect1 hold rect2? true
Can rect1 hold rect3? false
```

- We want to define a method named `can_hold` which will be defined within the `impl Rectangle` block. It takes an immutable borrow of another `Rectangle` as a parameter.

- The return value of `can_hold` will be a Boolean, and the implementation will check whether the width and height of `self` are greater than the width and height of the other `Rectangle`, respectively.

```
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
    }
}
```

- Methods can take multiple parameters that we add to the signature after the `self` parameter, and those parameters work just like parameters in functions.

Associated Functions

- All functions defined within an `impl` block are called *associated functions* because they're associated with the type named after `impl`.
- We can define associated functions that don't have `self` as their first parameter (and thus are not methods) because they don't need an instance of the type to work with.
- **We've already used one function like this: the `String::from` function that's defined on the `String` type.**
- Associated functions that aren't methods are often used for constructors that will return a new instance of the struct. These are often called `new`, but `new` isn't a special name and isn't built into the language.
- For example, we could choose to provide an associated function named `square` that would have one dimension parameter and use that as both width and height, thus making it easier to create a square `Rectangle` rather than having to specify the same value twice:

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

- The `Self` keywords in the return type and in the body of the function are aliases for the type that appears after the `impl` keyword, which in this case is `Rectangle`.
- **To call this associated function, we use the `::` syntax with the struct name; `let sq = Rectangle::square(3);` is an example.** This function is namespaced by the struct: the `::` syntax is used for both associated functions and namespaces created by modules.

Mutliple impl Blocks

- Each struct can have multiple `impl` blocks but this is not recommended. Yet, there's a usecase where multiple `impl` blocks are useful in **generic types and traits.**

Summary

- Struct lets you create custom types that are meaningful for your domain.
- Using them, you can keep associated pieces of data connected to each other and name each piece to make you code clear.
- You can define associated functions with your type, and methods which are a kind of associated function that lets you specify the behavior that instances of your structs have.
- But structs aren't the only way you can create custom types: let's turn to Rust's enum feature to add another tool to your toolbox.

```
Dataview (inline field 'help: the trait `std::fmt::Display` is not implemented
for `Rectangle`
   = note: in format strings you may be able to use `{:?}` (or {:#?} for pretty-
print) instead'): Error:
-- PARSING FAILED --------------------------------------------------

> 1 | help: the trait `std::fmt::Display` is not implemented for `Rectangle`
    |     ^
  2 |    = note: in format strings you may be able to use `{:?}` (or {:#?} for
pretty-print) instead

Expected one of the following:

'(', '*' or '/' or '%', '+' or '-', '.', '>=' or '<=' or '!=' or '=' or '>' or
'<', '[', 'and' or 'or', /[0-9\p{Letter}_-]/u, EOF, text
```

```
Dataview (inline field 'help: the trait `Debug` is not implemented for
`Rectangle`
   = note: add `#[derive(Debug)]` to `Rectangle` or manually `impl Debug for
Rectangle`'): Error:
-- PARSING FAILED --------------------------------------------------

> 1 | help: the trait `Debug` is not implemented for `Rectangle`
    |     ^
  2 |    = note: add `#[derive(Debug)]` to `Rectangle` or manually `impl Debug
for Rectangle`

Expected one of the following:

'(', '*' or '/' or '%', '+' or '-', '.', '>=' or '<=' or '!=' or '=' or '>' or
'<', '[', 'and' or 'or', /[0-9\p{Letter}_-]/u, EOF, text
```