Contents

1 Using structs to structure related data

- so, it says that some of the best things about structs are the methods and type checking that go along with them
-

2 5.1 defining and instantiating structs

```
fn main() {
```

2.1 Structs: More flexible than tuples

- every value is named
- these named values make up the struct's *fields*

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

```
let user1 = User {
    email: String::from("abc@123.com.ru.rs.sc.am.y.ou"),
    username: String::from("someone"),
    active: true,
    sign_in_count: 1,
};
```

- Note that the lines between struct fields are separated by commas, because it isn't a statement(";") or expression("<nothing>")

2.2 mutability

- either all of the instance of a struct is mutable, or none of it is.
- you can't declare individual fields to be mutable

```
let mut user1 = User {
    email: String::from("abc@123.com.ru.rs.sc.am.y.ou"),
    username: String::from("someone"),
    active: true,
    sign_in_count: 1,
};
// now it can be mutated
user1.email = String::from("abababa@example.com");
```

#+end_src

2.3 using a function

- In the build_user function below, having the User{} part as the last **expression** (aka no semicolon) of the function means (implicitly) that it is the value to be returned

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

2.4 Ok, now the field init shorthand!!!!!!!!!!

- As long as the named parameters in a function that instantiates a struct are the same as the struct's fields, you can use the `field init shorthand` syntax
- saves time, less repetitive
- Also, syntactically, you can give the last line a comma

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

- so above, some of the fields are done by field init and the others are instantiated within the function

2.5 `Struct Update` syntax: creating instances from other instances

2.5.1 With and Without `Struct update` syntax

```
// without Struct Update syntax - manually declaring values
// let user2 = User {
//     active: user1.active,
//     username: user1.username,
//     email: String::from("another@example.com"),
//     sign_in_count: user1.sign_in_count,
// };

// With Struct update syntax - showing how fields not explicitly set should take their
    // BUT! below no es bueno because the ownership of user1's "username" value was not
let user2 = User {
    email: String::from("jack@jack.com"),
    ..user1
};
```

1. SERIOUSLY important information re: above
    - OK: I think it is because the email and username fields are of `String` type, they therefore aren't string literals
    - aka since they're not string literals, they are just pointers to the same
    - I tried a few things to make it work but no bueno so far. Not sure why you'd ever want the use case where you could still have it in scope in a different User object, though
    - importantly, the user2 example above **moves** the ownership of user1's `username` field into user2
    - it is therefore out of scope for user

2.6 Using Tuple Structs w/o named fields to create different types

- Ok, so the below is useful/important because now there are different types for two different pieces of data that would otherwise share the exact same structure
- this way, functions that are supposed to take structs of type `Color` won't work if given a type `Point`

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

let black = Color(0,0,0);
let origin = Point(0,0,0);
```

2.7 unit-like structs without any fields

- These behave similarly to `()` (c.f. ch3.2)
- "Useful in situations in which you need to implement a trait on some type but don;t have any data taht you want to store inthe type itself"

```
struct AlwaysEqual;
let subject = AlwaysEqual;
```

- implications of this will be covered more in ch. 10

2.8 Ownership of struct data

- this box is talking more about stuff we'll go into more depth in later
- This is important, but I don't wnat to type it out right now so here's the link
- go to the bottom of the page

2.9 close main

```
}
```

3 5.2 example program using structs

```
fn main() {
```

3.1 beginning of the thing, without structs

"we'll start with a single variable, and then refactor until we're using structs instead"

```
let width1 = 30;
let height1 = 50;

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

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

- There are some interesting design points in this section
- they say that "the area function is supposed to calculate the area of one rectangle, but we've wrote a function with two parameters"
- since the parameters are related, eg they always only describe one rectangle, it would be better practice for legibility to group them together

3.2 grouping using tuples

```
let rect1 = (30, 50);
println!(
    "the area of the rectangle is {} square pixels",
    area(width1, height1)
);

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

- so, this is more structured but is also in a way less clear
- tuples don't name their fields
- so it's simple in this example, but would get very complicated in others

3.3 refactoring with structs

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

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

```
println!(
    "The rectangle's area is: {} square pixels",
    area(&rect1)
);

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

- ok cool, so here we make the rectangle struct and then create an instance of it.
- when we call the area function on it, we **borrow** the instance rect1
- the area function defined knows to expect a **borrowed** Rectangle struct
  - this parameter in the area function is further defined to be an **immutable borrow**
- this is definitely more clear, as we can then call rectangle.width and rectangle.height

3.4 Adding useful functionality with derived traits

- so, we can't currently print a rect1 struct
- we need to define a Display formatting
- we would ALSO need to define a Debug formatting for our new struct Rectangle
- adding the debugging is easy -> you effectively have to opt in

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
    }
fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
        };
    // prints the whole structure
    println!("rect1 is {:?}",rect1);
    // prints it in a line-by-line way
    println!("rect1 is {:#?}",rect1);
    }
```

- debug (called by `{:?}`) effectively allows the entire instance to be printed as it is defined in the code
- shows all fields of the instance
- with larger structs we can use `{:#?}` and fields will all be printed on separate lines

3.5 dbg! macro

- `dbg!` macro:
  - takes ownership of an expression
  - prints the file and line number where the `dbg!` macro was
  - prints the value of the expression
  - returns ownership of the value
  - then prints this value to the `stderr` stream
- this printing to `stderr` is different from how println! prints to `stdout`

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

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

- in the example above, we can use `dbg!` to

3.6 close main

```
}
```