Structs

## Defining a struct

- Structs in Rust are like the ones in C and C++, they help define complex types

- Structs are immutable by default

- Let's define a simple struct:

  ```
  struct User{
  username: String,
  email: String,
  sign_in_count: i32,
  active: bool,
  }
  ```

  - Structs are defined outside of the fn main() usually but can be defined anywhere
  - The values within a struct can be accessed with the `.` operator

- Lets initalize one struct

  ```
  let user1 = User{
      email : String::from("world@example.com"),
      username : String::from("some_name"),
      sign_in_count : 0,
      active : false,
  };
  ```

  - This is now immutable and has a starting value for each field

  - **Note** that the struct object can be mutable using `let mut user1 =etc..`

    - You have to make the whole struct mutable, you cannot do that from a specific field

  - Field-Init-Shorthand Syntax:

    ```
    let email :String = String::from("world@example.com");
    let username :String = String::from("some_name");
    let user1 = User{
        email,
        username,
        sign_in_count : 0,
        active : false,
    };
    ```

        - Since the name of `email` and `username` are already defined within the same scope, we can use shorthand syntax.
        - Just note that by doing this shorthand syntax, you are performing a move and not a borrow or copy

## Utitilizing a struct's fields

~~~
```rust
let name = user1.username;
println!("Hello {} from {}", name, user1.email);
```
~~~

- In this example two important things to note
    - One, `name` now owns `user1.username`, and that field can no longer be used
    - Two, although one field is taken, other fields are still accessible
- Struct constructors:
    - Example:

    ```
    let user2 = build_user(String::from("earth@example.com"),
    String::from("newUser"));

    fn build_user(email: String, username: String) -> User {
    let user =  User{
        email,
        username,
        sign_in_count: 0,
        active: false,
    };

    user
    }
    ```

- Reusing instances of Structs
    - You can define a new struct with an existing struct's fields
    - You can do this with `..<existing_struct_object>`
    - Example

    ```
    let user3 = User{
        username : String::from("user3"),
    ```

```
    ..user2
};
```

    - Note, this also performs a move for any fields that do not copy by default.
    - Fields that are not copied are no longer owned by the previous instance.

  - Tuple Structs
    - These are structs that define a new tuple type
    - They are useful if you want a tuple to have a specific collection of types and still a different name from other tuples
    - Example

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

      let new_color = Color(0, 1, 2);
      let new_point = Point(1, 2, 3);
      ```

      - Note that Color and Point are the exact same tuple structure,but have different names
      - This allows us to make a point instance that is a specific point type
        - we can access the fields of this type using the normal tuple structure
          - new_point.0 -> 1, new_point.1 -> 2, new_point.2 -> 3

## Printing out a struct

- Structs need a display trait
- Similar to tuple and arrays, these don't have a display traits
- What about the `{:?}` operator?
- This is the debug trait operator that prints out complex data types in a meaningful way for debugging and developers
  - This operator is defined by default for tuples and arrays but not Structs
  - We need to add a debug trait **ontop** of the struct `#[derive(Debug)]`
  - Example:

    ```
    #[derive(Debug)]
    struct User{
    username: String,
    email: String,
    sign_in_count: i32,
    active: bool,
    }
    ```

```
#[derive(Debug)]
struct Color(i32, i32, i32);
#[derive(Debug)]
struct Point(i32, i32, i32);

let new_color = Color(0, 0, 0);
let new_point = Point(0, 0, 0);
println!("new_point -> {:?}, user3 -> {:?}", new_point, user3);
```

## Methods for Structs

- Structs can have member methods
- Member methods are functions that only apply to instances of the struct
- You need a block or scope to label the methods.
    - This is done using the implementation keyword `impl`
- Example:

    ```
    struct Rectangle{
      width: i32,
      height: i32,
    }

    impl Rectangle {
        fn area(&self) -> i32{
            self.width * self.height
        }

        fn make_square(&mut self) {
        self.width = self.height;
      }
    }

    impl Rectangle {
        fn make_new_square(size: i32) -> Rectangle{
          let rect : Rectangle = Rectangle{
            width : size,
            height: size,
          };
          return rect;
        }
    }
    ```

    - In this example we can see that there is an implementation block where a member method is defined.

    - This function takes in **a reference to the object** (i.e. it borrows the object) and returns an i32 result.
    - We could also make it take in a &mut reference to the object, which would allow us to mutate the value in the object

- To call the member method simply use the `.` operator
    - `rect.area()`
    - This applies to methods that mutate the value as well.

- Associated Functions
    - These are functions that are associated to a Struct but not a method of one
    - In other words the function doesn't pass in the instance of itself as a parameter
    - Example:

```rust
impl Rectangle {
    fn make_new_square(size: i32) -> Rectangle{
      let rect : Rectangle = Rectangle{
        width : size,
        height: size,
      };
      return rect;
    }
}

let rect : Rectangle = Rectangle::make_new_square(10);
```

        - Here the make_new_square is an Associated Function since there is no `&self` as a parameter.
        - We can also see that to call this function is similar to a static method of a class
        - This is associated to Rectangle so it needs to use Rectangle namespace
        - This is also similar to how String::from works.