# CoE 164

Computing Platforms

02b: Rust Enums and Structures

## DATA STRUCTURES

Aside from the primitive types, Rust also has *compound data types*, which can be constructed using the primitive data types.

- `struct`
- `enum`

# STRUCTURES

A **structure (struct)** is a compound data type that groups multiple data placed into *fields*. Each field has a name and data type separated by commas.

We declare a struct using the `struct` keyword. Names are in CamelCase by convention.

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

Example

## STRUCTS: INITIALIZATION

We can instantiate a struct by writing the struct definition, but the data types are replaced by the values that the struct should have.

```
let user_a = UserAcct {
    active: true,
    username: "abcxyz".to_string(),
    sign_in_count: 0,
};
```

Example

## STRUCTS: MEMBERSHIP

To get the value of a field of a struct, the dot notation is used. It is possible to also mutate a field value via assignment.

```
let user_a = UserAcct {
    active: true,
    username: "abcxyz".to_string(),
    sign_in_count: 0,
};

println!("sign-ins: {}", user_a.sign_in_count);
```

Example

## STRUCTS: MUTABILITY

It is not possible to pick certain fields of a struct to be mutable. The whole struct should be marked as mutable if any of the fields should be editable.

```
let mut user_a = UserAcct {
    active: true,
    username: "abcxyz".to_string(),
    sign_in_count: 0,
};

user_a.sign_in_count += 1;
```

Example

## STRUCTS: FUNCTIONS

Functions can accept structs as arguments and return values.

```
fn is_active(user: &UserAcct) -> bool {
    user.active
}

fn build(username: String) -> UserAcct {
    UserAcct {
        active: false,
        username: username,
        sign_in_count: 0,
    }
}
```

Example

## STRUCTS: FUNCTIONS

Variables with the same name can be assigned to fields of a struct with the same name using the shorthand syntax during initialization.

```
fn build(username: String) -> UserAcct {
    UserAcct {
        active: false,
        username,
        sign_in_count: 0,
    }
}
```

Example

## STRUCTS: COPYING

Data from one structure to another can be copied using the dot notation. However, the **struct update syntax** can be used if only selected fields will differ from the origin struct.

Note that the last field containing the origin struct should not have a trailing comma.

Example

```
let user_a = UserAcct {
    active: true,
    username:
"userA".to_string(),
    sign_in_count: 50,
};

let user_b = UserAcct {
    username:
"userB".to_string(),
    ..user_a
};
```

# STRUCTS: COPYING

Copying and moving data from one struct to another, or even assigning data to a struct, still follows the ownership rules. Hence, there may be instances where the fields of the origin struct may not be usable due to move.

Example

```
let my_name =
"hello".to_string();


let user_c = UserAcct {

    active: true,

    username: my_name,

    sign_in_count: 50,

};


// Compile error below!
println!("{my_name}");
```

## STRUCTS: TUPLES

A **tuple struct** can be created by writing the tuple definition after the struct name. These are structs with unnamed fields with numbered indices as their field names - or conversely - tuples that have names.

```
struct RGBColor(i32, i32, i32);
struct HSVColor(i32, i32, i32);


let green = RGBColor(0, 255, 0);
println!("RGB({}, {}, {})", green.0, green.1, green.2);
```

Example

## STRUCTS: UNITS

A **unit-like struct** can be created by writing nothing after the struct name. It is the same as a tuple struct with the empty or unit tuple.

This kind of struct is useful if we want a "class" that can have methods but no kind of data stored in any of its instances.

```
struct SampleStruct;

let ss = SampleStruct;
```

Example

## STRUCTS: METHODS

A **method** is a function that is *bound* to a specific instance of some "object". Methods can be added to structs by writing the methods inside an `impl` block. A struct can have multiple `impl` blocks spread across the program.

Methods are called using the dot notation.

```
impl UserAcct {
    fn get_name(&self) -> &String {
        &self.username
    }
}

println("{}", user_a.get_name());
```

Example

## STRUCTS: METHODS

The first parameter of a method should always be the instance of the struct given by `self: &Self`. Most programmers will write `&self` instead as shorthand.

`self` is almost always given as a reference, but it is possible for the method to take ownership of it.

Example

```
impl UserAcct {
    fn get_name(&self) -> &String {
        &self.username
    }

    fn is_act(this: &Self) -> bool {
        this.active
    }

    fn add_signup(&mut self) {
        self.sign_in_count += 1
    }
}
```

## STRUCTS: METHODS

If a method should have parameters, those parameters are added after the first parameter containing `Self`.

When calling a method, the first parameter in the call will be assigned to the second parameter in the method declaration.

```
impl UserAcct {
    fn prefix_name(&self, title: String) -> String {
        title + &String::from(" ") + &self.username.to_string()
    }
}

println!("{}", user_a.prefix_name(String::from("Mr.")));
```

Example

# STRUCTS: ASSOCIATED FUNCTIONS

Functions bound to a struct whose first parameter is not `Self` is an **associated function**. Such functions usually are **constructors**, which are functions that create an instance of that struct.

Associated functions are called by writing the function name prefixed with the name of the struct separated by a double colon.

Example

```rust
impl UserAcct {
    fn new() -> Self {
        Self {
            active: false,
            username: String::from(""),
            sign_in_count: 0,
        }
    }

    fn add(a: i64, b: i64) -> i64 {
        a + b
    }
}

let blank_acct = UserAcct::new();
let c = UserAcct::add(3, 5);
```

## ENUMERATIONS

An **enumeration (enum)** is a compound data type that can take on one of its possibly many **variants**. Enums are usually used to force data to be only from one of the selected choices.

```
enum UserType {
    SuperAdmin,
    Admin,
    User,
    Unknown,
}
```

Example

## ENUMS: VARIANTS

Enums can also hold some data in any of its variants, which can be retrieved when the enum is of that variant.

With this example, we can think of each variant as a struct grouped under a single name.

```
enum UserType {
    SuperAdmin,
    Admin(bool, u16),      // tuple struct
    User { chown: u16 }, // struct (with named fields)
    Unknown,               // unit struct
}
```

Example

## ENUMS: VARIANTS

Enums can be assigned to variables by specifying one of its variants and the data that they will hold. Two colons are used to separate the enum and variant name.

```
let admin_all = UserType::Admin( true, 0o777);
let read_only = UserType::User { chown:  0o444 };
let mut unknown_user = UserType::Unknown;
```

Example

## ENUMS: METHODS

Enums can also be bound to methods or associated functions through `impl` blocks. These methods or functions apply to all variants of the enum.

```
impl UserType {
    fn is_acct(&self) -> bool {
        true
    }
}

let admin_type = UserType::Admin(true, 0o777);
blank_acct.is_acct();
```

Example

## RESOURCES

- The Rust Book

# CoE 164

Computing Platforms

02b: Rust Enums and Structures