Structs

Learn to Code with Rust / Section Review

Structs

- A **struct** is a container for related data. We use structs to model complex real-world types.
- Rust has 3 types of structs:
  - Named field structs (assigns a name to each field)
  - Tuple-like structs (assigns a position/order to each field)
  - Unit-like structs (has no fields)
- Declare a struct with the **struct** keyword, then the name in **PascalCase**.

Named Field Structs

- A struct declaration defines the *blueprint* for what the struct will look like. It establishes a new type in Rust.
- Write the field names and their associated data types. Separate the fields with commas.
- An **instance** is a concrete struct value.
- To create an instance, write the struct name and a pair of curly braces. Provide the fields and their associated values.
- The struct owns the fields. The fields own their values.

Struct Initialization and Updates

- If a field matches an existing name (variable or parameter), we can simplify the **field: value** syntax to **field**.
- Use .. to copy one struct's fields' values to another.
- Declare fields whose values you'd like to define *before* the .. syntax. Rust will *not* overwrite these fields.
- Ownership rules apply to .. syntax. If a type does not implement the Copy trait, ownership will move from the first struct's field to the assigned struct's field.

The Debug Trait

- A **trait** is a contract that requires that a type supports one or more methods.
- We can implement the **Debug** trait for a struct with the **#[derive(Debug)]** attribute
- The **Debug** string representation of a struct includes its name and all of the fields + values.
- Structs by default do not implement the **Copy** trait so ownership principles apply.

Methods

- Methods are functions attached to a value.
- A method can accept parameters (inputs) and produce a return value (output).
- Define struct methods in one or more **impl** blocks using regular **fn** syntax.

The **self** Parameter

- Methods receive **self** as the first parameter.
- **self** can represent either the owned instance or a reference to it.
- We can receive both instances and references mutably or immutably.
- Methods must receive either the mutable struct or a mutable reference to the struct to modify its field values.

self Declarations

- **Immutable Ownership**
  - self: MyStruct
  - self: Self
  - self
- **Mutable Ownership**
  - mut self: MyStruct
  - mut self: Self
  - mut self
- **Immutable Reference**
  - self: &MyStruct
  - self: &Self
  - &self
- **Mutable Reference**
  - self: &mut MyStruct
  - self: &mut Self
  - &mut self

Method Invocations

- Invoke a method by providing the value, a dot, the method name, and a pair of parentheses.

- Rust will provide the right value for **self** automatically.

- Methods can define additional parameters after **self**. Pass those arguments in during invocation.

Methods on References

- Methods can be invoked on references.
- Rust will dereference (follow the address) to the original struct.
- The advantage of this design is that a method's code remains the same if the parameter is an instance *or* a reference to an instance.

Associated Functions

- An **associated function** is a function that lives within the struct's namespace. It cannot be accessed outside of the struct.
- Use **MyStruct::associated_function()** to invoke the associated function.
- The most common use case is a **constructor function**, which creates and returns an instance.
- The constructor function is usually called **new**.

Tuple Structs and Unit Structs

- A **tuple struct** orders its fields by position.
- Access field values with a dot and the field's index position. The index starts at 0.
- A tuple struct is not interchangeable with a tuple or another tuple struct with the same fields.
- A **unit-like struct** is a struct without data.