Struct and Generics

Struct

Structs, short for structures, are custom data types that help you group related data together. They are helpful in organizing your code and making it more readable and maintainable. Think of struct as a blueprint for creating objects with specific properties and behaviors.

Defining struct

Defining a struct starts with the key word `struct`, followed by the name of the struct and a pair of curly braces `{}`. Then inside the curly braces, you define the fields of the struct with a name and a type, separated by a colon.

Example: Let's say you are building a program that deals with information about books. We know a book has a title, an author, and publication year. We create a struct book with the related fields:

```
struct Book {
    title: String,
    author: String,
    publication_year: u32,
}
```

Creating an instance of a book:

```
let book = Book {
    title: String::from("Rust Book"),
    author: String::from("Steve"),
    publication_year: 2018
};
```

Accessing the struct fields:

```
struct Book {
    title: String,
```

```
    author: String,
    publication_year: u32,
}

fn main() {
    let book = Book {
        title: String::from("Rust Book"),
        author: String::from("Steve"),
        publication_year: 2018
    };
    println!("Title: {}", book.title);
}
```

Tuple Structs

A tuple struct is similar to a regular struct, but its fields do not have names. Instead, they are accessed by their position. Tuple structs can be useful when you want to create a simple data structure with a few fields but don't need the complexity of named fields.

Example:

```
struct BankAccount(u32, f64);

fn main() {
    // Creating an instance tuple
    let account = BankAccount(12, 1500.75);

    // Access the tuple
    println!("Account ID: {}, Balance: ${}", account.0, account.1);
}
```

Unit Structs

Unit structs are a special kind of struct in Rust that don't have any fields. They are useful when you want to create a distinct type without carrying any data with it. It's often used for marker traits, or when you just need a type but don't need to store any data.

Example:

```
struct ReadOnly;

fn process_data(_: ReadOnly) {
    println!("Processing data in read-only mode...");
}
```

```
fn main() {
    let mode = ReadOnly;
    process_data(mode);
}
```

Implementing Methods for Structs

Methods allow us to define functions that are specifically tied to a struct, making it easier to work with the struct's data and providing a more organized way to interact with the struct. They are similar to regular functions, but they're defined within an `impl` block, which associates the methods with the struct.

Example:

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

impl Rectangle {


    // Method to calculate area
    fn area(&self) -> u32 {
        self.width * self.height
    }
    fn perimeter(&self)-> u32{
        (self.width + self.height)*2
    }
}

fn main() {
    let rect = Rectangle{ width: 30,  height: 50};
    println!("Area: {}", rect.area());
    println!("perimeter: {}", rect.perimeter());
}
```

Generics

Let's talk about **generics** but don't panic! They're just a way to make your code flexible without extra effort. Think of it like having one shirt that magically changes color instead of buying a dozen nearly identical ones. Why write the same function for integers, floats, and strings when **one** can do it all?

Generics are Rust's way of saying, **"Chill, I got this."** They let us write code that adapts to different types, keeping things DRY (Don't Repeat Yourself) and elegant.

Generic Function Example

```rust
fn largest<T>(list: &[T]) -> &T {
    let mut largest = &list[0];

    for item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let result = largest(&number_list);
    println!("The largest number is {result}");

    let char_list = vec!['y', 'm', 'a', 'q'];

    let result = largest(&char_list);
    println!("The largest char is {result}");
}
```

Generic Struct Example

```rust
// Generic struct that can hold any type
struct Container<T> {
    value: T,
}

impl<T> Container<T> {
    // Constructor
    fn new(value: T) -> Container<T> {
        Container { value }
    }

    // Getter method
    fn get_value(&self) -> &T {
        &self.value
    }
}

fn main() {
    // Container with integer
    let number = Container::new(42);
    println!("Number: {}", number.get_value());
```

```rust
    // Container with string
    let text = Container::new(String::from("Hello, Rust!"));
    println!("Text: {}", text.get_value());
}
```

Generic Struct with Multiple Types

```rust
struct Pair<T, U> {
    first: T,
    second: U,
}

impl<T, U> Pair<T, U> {
    fn new(first: T, second: U) -> Pair<T, U> {
        Pair { first, second }
    }
}

fn main() {
    let pair = Pair::new(1, "hello");
    println!("First: {}, Second: {}", pair.first, pair.second);
}
```

Benefits of Generics

- Code reusability
- Type safety at compile time
- Zero runtime overhead
- Clean and maintainable code

Common Use Cases

- Collections (Vec<T>, HashMap<K, V>)
- Option<T> and Result<T, E>
- Generic functions and methods

Best Practices

Use meaningful type parameter names (T for type, K for key, V for value)

Find me on

github -> Joanne-cmd

X->@joannetich