20CYS312 - Principles of Programing Languages - Lab Exercise 8

Roll number: CH.EN.U4CYS22025

Name: Pranish K

1. Library Book Management System (Ownership & Move Semantics)

Problem Statement: You are developing a **library book management system** where books are added, issued, and returned. Implement the following functionalities in Rust:

- **Define a `Book` structure** with fields: `title`, `author`, `ISBN`, and `is_issued` (boolean).
- **Implement an `issue_book` function** that moves ownership of a book from the library to a borrower.
- **Demonstrate ownership transfer** by preventing access to the book once it is issued.
- **Use `.clone()` to allow the library to maintain a backup of issued books.**

Objective:

The objective of this program is to implement a **Library Book Management System** in Rust that demonstrates ownership transfer, move semantics, and cloning. Specifically, it should:

1. Define a `Book` structure with fields for `title`, `author`, `ISBN`, and `is_issued` (a boolean flag to track whether the book is issued or not).

2. Implement a function `issue_book` that moves the ownership of a book from the library (the original owner) to a borrower (when the book is issued).
3. Prevent access to the original book once it has been issued by transferring ownership.
4. Use `.clone()` to create a backup of the book before it is issued, ensuring that the library keeps a record of the book even after it has been issued.

Code

```
struct Book {
    title: String,
    author: String,
    isbn: String,
    is_issued: bool,
}

impl Book {
    fn new(title: &str, author: &str, isbn: &str) -> Self {
        Book {
            title: title.to_string(),
            author: author.to_string(),
            isbn: isbn.to_string(),
            is_issued: false,
        }
    }
    fn issue_book(self) -> Book {
        println!("Issuing the book: {}", self.title);
        Book {
            title: self.title,
            author: self.author,
            isbn: self.isbn,
            is_issued: true,
        }
    }
```

```
    fn is_issued(&self) -> bool {
        self.is_issued
    }

    fn details(&self) {
        println!("Title: {}, Author: {}, ISBN: {}, Issued: {}",
                 self.title, self.author, self.isbn, self.is_issued);
    }
}

fn main() {
    let book1 = Book::new("The Rust Book", "John Doe", "123-456-789");
    let backup_book = book1.clone();

    println!("Library backup (before issue):");
    backup_book.details();

    let issued_book = book1.issue_book();

    println!("\nIssued book details:");
    issued_book.details();
}
```

Explanation of the Code:

1. **Struct Definition (`Book`):**
    - The `Book` struct is defined with fields:
        - `title`: The title of the book.
        - `author`: The author of the book.
        - `isbn`: The ISBN number of the book.
        - `is_issued`: A boolean indicating whether the book is issued.

2. **Ownership and Move Semantics:**
    - The `issue_book` function is implemented to take ownership of the book (`self`), which means once a book is issued, its ownership is transferred from the library (or original owner) to the borrower.
    - After the ownership is transferred, the original `book1` cannot be accessed anymore. This is enforced by Rust's ownership system.
    - The `clone()` function is used to create a backup copy of the book before it is issued. This ensures the library maintains a record of the book, even if the book itself is moved.
3. **Methods (`new`, `issue_book`, `details`):**
    - The `new` method is a constructor to create a new `Book` instance.
    - The `issue_book` method moves ownership and updates the `is_issued` flag to `true`.
    - The `details` method prints the details of the book, including whether it has been issued.
4. **Main Function:**
    - A book is created using `Book::new()`, and then a backup copy is made using `.clone()`.
    - The book is issued by calling the `issue_book()` method, and the details of the backup and issued books are printed.
5. **Rust's Ownership Model:**
    - When the book is issued, ownership is moved from the original `book1` to `issued_book`. Attempting to access `book1` after it is moved will result in a compile-time error, demonstrating Rust's strict ownership rules.
    - The use of `.clone()` ensures that a backup of the original book is maintained, allowing the library to keep records of all books, even those that have been issued.

Output screenshot

```
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~/vexo/6th-sem-labs/PPL/Lab-08$ ./q1
Library backup (before issue):
Title: The Rust Book, Author: John Doe, ISBN: 123-456-789, Issued: false
Issuing the book: The Rust Book

Issued book details:
Title: The Rust Book, Author: John Doe, ISBN: 123-456-789, Issued: true
```

2. Secure Banking System (Borrowing & Mutable References)

Problem Statement: Design a **secure banking system** where multiple users can check their balance, but only one user can modify it at a time.

- **Define a** `BankAccount` **struct** with fields: `account_number`, `owner_name`, and `balance`.
- **Implement** `view_balance()` to allow multiple users to **borrow** (immutable reference) the balance.
- **Implement** `deposit()` **and** `withdraw()` **functions** that modify the balance using **mutable borrowing**.
- **Ensure only one function modifies the balance at a time.**

Objective:

The objective of this program is to implement a **secure banking system** where users can:

1. View the balance concurrently by borrowing an immutable reference.
2. Deposit and withdraw funds by borrowing a mutable reference (modifying the balance).
3. Ensure that only one operation can modify the balance at any time using Rust's borrowing rules to prevent concurrent mutable access.

Code

```
struct BankAccount {
    account_number: String,
    owner_name: String,
    balance: f64,
}
```

```
impl BankAccount {
    fn new(account_number: &str, owner_name: &str, balance: f64) -> Self {
        BankAccount {
            account_number: account_number.to_string(),
            owner_name: owner_name.to_string(),
            balance,
        }
    }

    fn view_balance(&self) -> f64 {
        self.balance
    }

    fn deposit(&mut self, amount: f64) {
        if amount > 0.0 {
            self.balance += amount;
            println!("Deposited ${}. New balance: ${}", amount, self.balance);
        } else {
            println!("Deposit amount must be greater than zero.");
        }
    }

    fn withdraw(&mut self, amount: f64) {
        if amount > 0.0 && self.balance >= amount {
            self.balance -= amount;
            println!("Withdrew ${}. New balance: ${}", amount, self.balance);
        } else {
            println!("Insufficient funds or invalid withdrawal amount.");
        }
    }
}

fn main() {
    let mut account = BankAccount::new("123456", "Alice", 500.0);
```

```
    println!("Initial balance: ${}", account.view_balance());

    account.deposit(200.0);
    println!("Balance after deposit: ${}", account.view_balance());

    account.withdraw(150.0);
    println!("Balance after withdrawal: ${}", account.view_balance());
}
```

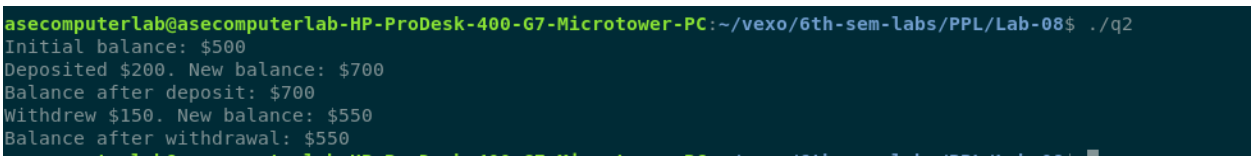
Explanation

1. **BankAccount Struct:**
   - The `BankAccount` struct is defined with fields:
     - `account_number`: A unique identifier for the account.
     - `owner_name`: The name of the account holder.
     - `balance`: The current balance of the account.
2. **Methods:**
   - **`new`**: A constructor to create a new `BankAccount` with an account number, owner name, and initial balance.
   - **`view_balance`**: A method that borrows an immutable reference to the `BankAccount` to allow users to view the current balance.
   - **`deposit`**: A method that borrows a mutable reference to the `BankAccount` to deposit a certain amount into the account. The deposit is only allowed if the amount is greater than zero.
   - **`withdraw`**: A method that borrows a mutable reference to the `BankAccount` to withdraw a certain amount. It checks if there are enough funds for the withdrawal and ensures the amount is positive.
3. **Mutable and Immutable Borrowing:**
   - **Immutable borrowing** (`&self`) is used in `view_balance()` to allow multiple users to check the balance concurrently without modifying it.

- **Mutable borrowing** (`&mut self`) is used in the `deposit()` and `withdraw()` methods to ensure that only one user can modify the balance at a time. Rust's borrowing rules enforce that no other mutable or immutable references can coexist when modifying the balance, ensuring thread safety.
- If another mutable reference were attempted (e.g., calling `deposit()` and `withdraw()` simultaneously), the Rust compiler would prevent this due to its strict borrowing rules.

4. **Main Function:**
   - The main function demonstrates creating a `BankAccount`, viewing its balance, making a deposit, and withdrawing funds. Each operation is done sequentially to ensure that no two mutable references to the account's balance are active at the same time.
5. **Ensuring Safe Access:**
   - The program enforces Rust's ownership and borrowing rules, preventing issues like race conditions or concurrent mutable access to the balance, which is a key feature for safe concurrency in a banking system.

Screenshot



3. Text Processing Tool (String Slices)

Problem Statement: You are building a **text-processing tool** that extracts useful information from user input. Implement the following functionalities:

- **Allow users to input a sentence.**
- **Extract a specific word** using **string slicing** (e.g., extract `"Rust"` from `"Rust is fast and safe."`).
- **Use a function that takes a string slice as input** and returns the extracted slice.

- **Modify the original string and ensure the extracted word remains valid.**

Code

```
use std::io;

fn main() {
    // Prompt user for input
    println!("Please enter a sentence:");

    // Read user input
    let mut input = String::new();
    io::stdin().read_line(&mut input).expect("Failed to read line");

    // Trim whitespace and extract the word "Rust"
    let word_to_extract = "Rust";
    if let Some(extracted_word) = extract_word(&input.trim(), word_to_extract) {
        println!("Extracted word: {}", extracted_word);

        // Modify the original string
        let modified_string = modify_string(&input.trim(), extracted_word);
        println!("Modified string: {}", modified_string);
    } else {
        println!("The word '{}' was not found in the input.", word_to_extract);
    }
}

// Function to extract a specific word from the input string
fn extract_word<'a>(input: &'a str, word: &'a str) -> Option<&'a str> {
    if input.contains(word) {
        let start = input.find(word).unwrap();
        let end = start + word.len();
        return Some(&input[start..end]);
    }
    None
}
```

```rust
// Function to modify the original string by removing the extracted word
fn modify_string(input: &str, word: &str) -> String {
    input.replace(word, "").trim().to_string()
}
```

Explanation

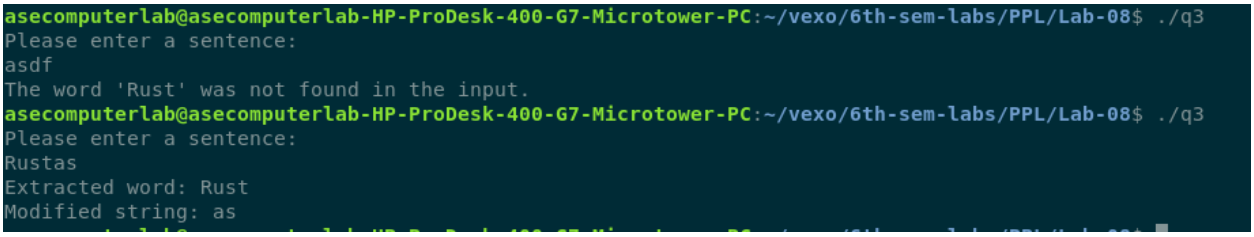
1. `extract_word` **Function:**

- **Purpose:** Extracts a specific word from the input string.
- **How:** Checks if the word is in the string and returns a slice of the word if found, or `None` if not.

2. `modify_string` **Function:**

- **Purpose:** Removes a word from the input string.
- **How:** Replaces the word with an empty string and trims any extra whitespace, returning the modified string.

Screenshot



4. Weather Data Analysis (Array Slices)

Problem Statement: Develop a **weather analysis tool** that processes **temperature readings** from a weather station.

- **Create an array of weekly temperature readings.**
- **Extract a slice of temperatures** representing the last **three days**.
- **Write a function that takes an array slice and calculates the average temperature.**

- **Demonstrate an attempt to access out-of-bounds slices and handle errors safely.**

Code

```
fn main() {
    let temperatures: [f32; 7] = [22.5, 23.0, 24.1, 25.0, 26.3, 27.4, 28.2];

    // Extract a slice for the last three days
    let last_three_days = &temperatures[4..7];

    // Calculate and print the average temperature of the last three days
    match calculate_average(last_three_days) {
        Some(average) ⇒ println!("Average temperature for the last 3 days: {:.2}°C",
        None ⇒ println!("Error: Unable to calculate the average temperature."),
    }

    // Demonstrate an attempt to access out-of-bounds slice
    // Uncommenting the next line will cause a runtime panic due to out-of-bounds
    // let out_of_bounds = &temperatures[10..15];  // This will cause an error
    // println!("{:?}", out_of_bounds);  // This won't be executed because of the err
}

// Function to calculate the average of an array slice
fn calculate_average(temps: &[f32]) -> Option<f32> {
    if temps.is_empty() {
        return None;
    }

    let sum: f32 = temps.iter().sum();
    Some(sum / temps.len() as f32)
}
```

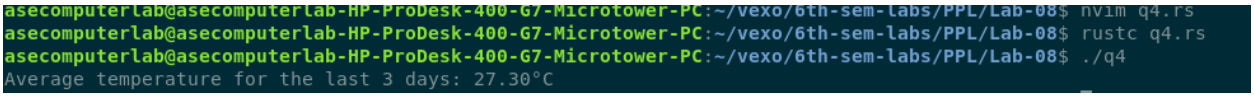
Explanation:

1. **Array and Slice:**

- `temperatures` is an array holding weekly temperature readings.
- We create a slice `last_three_days` from the `temperatures` array, which represents the last three days: `&temperatures[4..7]`.

2. **Average Calculation:**
   - The function `calculate_average` takes an array slice and calculates the average temperature using `iter()` and `sum()`.
   - It returns `None` if the slice is empty, and the calculated average wrapped in `Some(f32)` if not.
3. **Error Handling:**
   - The line `let out_of_bounds = &temperatures[10..15];` tries to access an out-of-bounds slice, which would result in a runtime error.
   - Rust will panic at runtime if this line is executed, and thus it is a demonstration of how not to access slices outside of the valid range.

Screenshot



5. Online Student Record System (Ownership & Borrowing)

Problem Statement: Develop a **student record system** where students can be added, updated, and displayed.

- **Use a `Student` struct** with fields: `name`, `age`, and `grade`.
- **Store multiple student records in a `Vec<Student>`.**
- **Implement a function that borrows student records** (immutable reference) to display them.
- **Implement another function that modifies a student's grade using mutable borrowing.**
- **Ensure Rust's borrowing rules prevent simultaneous modifications.**

code

```
// Define the Student struct with fields: name, age, and grade
#[derive(Debug)]
struct Student {
    name: String,
    age: u32,
    grade: String,  // Change grade type to String
}

impl Student {
    // Constructor to create a new Student
    fn new(name: &str, age: u32, grade: &str) -> Self {
        Student {
            name: name.to_string(),
            age,
            grade: grade.to_string(),  // Ensure grade is a String
        }
    }

    // Function to display student information (borrowed reference)
    fn display_student(student: &Student) {
        println!("Name: {}, Age: {}, Grade: {}", student.name, student.age, student.g
    }

    // Function to modify student's grade (mutable reference)
    fn update_grade(&mut self, new_grade: &str) {
        self.grade = new_grade.to_string();  // Convert the new_grade to String
    }
}

fn main() {
    // Create a Vec to store multiple Student records
    let mut students: Vec<Student> = Vec::new();

    // Add some students to the Vec
```

```
    students.push(Student::new("Alice", 20, "B"));
    students.push(Student::new("Bob", 22, "A"));
    students.push(Student::new("Charlie", 21, "C"));

    // Display all students (immutable borrowing)
    println!("Student records:");
    for student in &students {
        Student::display_student(student);
    }

    // Modify Bob's grade (mutable borrowing)
    if let Some(bob) = students.iter_mut().find(|s| s.name == "Bob") {
        bob.update_grade("A+");  // Now using a string literal
    }

    // Display updated records
    println!("\nUpdated student records:");
    for student in &students {
        Student::display_student(student);
    }
}
```

Key Features and Explanation:

1. **Student struct**:
   - `name` is a `String` to store the student's name.
   - `age` is a `u32` to store the student's age.
   - `grade` is a `char` to store the student's grade (e.g., 'A', 'B', 'C').
2. **Borrowing Functions**:
   - `display_student`: This function takes an immutable reference (`&Student`) and prints the student's details. It borrows the student record without taking ownership, so it cannot modify it.

- `update_grade`: This function takes a mutable reference (`&mut self`), allowing it to modify the student's grade.

3. **Vec<Student>**:
   - We store multiple students in a `Vec<Student>`, which allows for dynamic collection and manipulation of student records.
   - The `Vec` is mutable, allowing us to modify the student records (such as updating grades).
4. **Borrowing Rules**:
   - Rust enforces that you can have either an immutable reference or a mutable reference to a value, but not both simultaneously. This ensures safety when modifying or accessing student records.
   - In the `main` function, we demonstrate this by first borrowing students immutably to display them and later mutably to update Bob's grade.

Screenshot