PRINCIPLES OF PROGRAMMING LANGUAGES

LAB EXERCISE 8

Name: Sushant Yadav Date: 07/03/2025

Roll No: CH.EN.U4CYS22067 CourseCode:20CYS312

Task 1: Library Book Management System (Ownership & Move Semantics)

Objectives

1. **Implement a `Book` structure**
    a. Fields: `title`, `author`, `ISBN`, and `is_issued` (boolean).
2. **Develop a `Library` structure**
    a. Store available and issued books.

3. **Implement core functionalities:**
    a. `add_book()` — Adds new books to the library.
    b. `issue_book()` — Moves ownership of a book while maintaining a cloned backup.
    c. `return_book()` — Reintroduces returned books into the library's collection.
    d. `display_books()` — Lists available books.
4. **Demonstrate Ownership & Move Semantics**
    a. Ownership is transferred when a book is issued.
    b. `.clone()` ensures a backup is maintained for tracking issued books.

Code:

```
  GNU nano 6.4                                    library.rs
use std::collections::HashMap;

#[derive(Clone, Debug)]
struct Book {
    title: String,
    author: String,
    isbn: String,
    is_issued: bool,
}

impl Book {
    fn new(title: &str, author: &str, isbn: &str) -> Self {
        Self {
            title: title.to_string(),
            author: author.to_string(),
            isbn: isbn.to_string(),
            is_issued: false,
        }
    }
}

struct Library {
    books: HashMap<String, Book>,
    issued_books: Vec<Book>,
}

impl Library {
    fn new() -> Self {
        Self {
            books: HashMap::new(),
            issued_books: Vec::new(),
        }
    }

    fn add_book(&mut self, book: Book) {
        self.books.insert(book.isbn.clone(), book);
    }

    fn issue_book(&mut self, isbn: &str) -> Option<Book> {
        if let Some(mut book) = self.books.remove(isbn) {
            if !book.is_issued {
                book.is_issued = true;
                let backup = book.clone();
                self.issued_books.push(backup);
                println!("Book '{}' has been issued.", book.title);
                Some(book)
            } else {
                println!("Book '{}' is already issued.", book.title);
                None
            }
        } else {
```

```
            }
        } else {
            println!("Book with ISBN {} not found.", isbn);
            None
        }
    }

    fn return_book(&mut self, book: Book) {
        let mut returned_book = book.clone();
        returned_book.is_issued = false;
        self.books.insert(returned_book.isbn.clone(), returned_book);
        println!("Book '{}' has been returned.", book.title);
    }

    fn display_books(&self) {
        println!("Available Books in Library:");
        for book in self.books.values() {
            println!("{} by {} (ISBN: {})", book.title, book.author, book.isbn);
        }
    }
}

fn main() {
    let mut library = Library::new();

    let book1 = Book::new("The Rust Programming Language", "Steve Klabnik", "978-15932782
    let book2 = Book::new("Rust for Rustaceans", "Jon Gjengset", "978-1718501850");

    library.add_book(book1);
    library.add_book(book2);

    library.display_books();

    let issued_book = library.issue_book("978-1593278281");

    // Demonstrate move semantics by attempting to access the moved book
    if let Some(book) = issued_book {
        println!("Successfully issued: {}", book.title);
    }

    library.display_books();
}
```

Explanation

1. Book Structure

- Defined using `#[derive(Clone, Debug)]` to enable cloning and display capabilities.
- `new()` method simplifies book creation.

2. Library Structure

- Uses a `HashMap` for efficient book storage by ISBN (unique identifier).
- Maintains a `Vec` for tracking cloned issued books.

3. Functionality Details

- **`add_book()`**
  - Inserts a new book into the `books` collection.
- **`issue_book()`**
  - Moves ownership of a book to demonstrate Rust's ownership model.
  - Uses `.clone()` to store a backup in the `issued_books` list.
- **`return_book()`**
  - Accepts a returned book, marks it as available, and reinserts it.
- **`display_books()`**
  - Iterates over the `HashMap` to display available books.

4. Ownership Demonstration

- Once a book is issued, it's removed from the `books` collection, proving ownership transfer.
- Attempting to access the issued book directly afterward results in a **compile-time error**, enforcing Rust's ownership rules.

Output:

```
asecomputerlab@lab:~$ gedit library.rs
asecomputerlab@lab:~$ rustc library.rs
asecomputerlab@lab:~$ ./library
Book Details - Title: The Great Gatsby, Author: F. Scott Fitzgerald, ISBN: 12345
67890
Library book before issue: Book { title: "The Great Gatsby", author: "F. Scott F
itzgerald", isbn: "1234567890", is_issued: false }
Issued book: Book { title: "The Great Gatsby", author: "F. Scott Fitzgerald", is
bn: "1234567890", is_issued: true }
Backup of the library book: Book { title: "The Great Gatsby", author: "F. Scott
Fitzgerald", isbn: "1234567890", is_issued: false }
```

Conclusion

This implementation effectively showcases Rust's ownership model and move semantics. Key highlights include:

✅ Safe memory management via ownership transfer.
✅ Efficient tracking of issued books using `.clone()` for backups.
✅ Demonstrated error prevention by restricting access to moved data.

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.**

Objectives

The Secure Banking System aims to achieve the following goals:

1. **Implement a Secure Banking System**
    a. Define a `BankAccount` structure with fields: `account_number`, `owner_name`, and `balance`.
2. **Demonstrate Borrowing Concepts in Rust**
    a. Use **immutable references** to allow multiple users to view the account balance simultaneously.
    b. Use **mutable references** to ensure only one user can modify the balance at a time.
3. **Concurrency Control**
    a. Implement thread-safe operations using `Arc<Mutex<T>>` to prevent race conditions when accessing or modifying the balance.
4. **Provide Key Banking Functions**
    a. `view_balance()` for viewing the account balance.
    b. `deposit()` for adding funds.
    c. `withdraw()` for securely deducting funds with balance validation.

Code:

```
  GNU nano 6.4                                  banking.rs *
use std::sync::{Arc, Mutex};

struct BankAccount {
    account_number: u64,
    owner_name: String,
    balance: f64,
}

impl BankAccount {
    fn new(account_number: u64, owner_name: &str, balance: f64) -> Self {
        Self {
            account_number,
            owner_name: owner_name.to_string(),
            balance,
        }
    }

    // Immutable reference for viewing balance
    fn view_balance(&self) {
        println!(
            "Account: {} | Owner: {} | Balance: {:.2}",
            self.account_number, self.owner_name, self.balance
        );
    }

    // Mutable reference for modifying balance
    fn deposit(&mut self, amount: f64) {
        self.balance += amount;
        println!("Deposited {:.2}. New Balance: {:.2}", amount, self.balance);
    }

    fn withdraw(&mut self, amount: f64) {
        if self.balance >= amount {
            self.balance -= amount;
            println!("Withdrew {:.2}. New Balance: {:.2}", amount, self.balance);
        } else {
            println!("Insufficient funds. Current Balance: {:.2}", self.balance);
        }
    }
}

fn main() {
    let account = Arc::new(Mutex::new(BankAccount::new(123456789, "Sushant Yadav", 5000.0)));

    // Viewing balance (multiple readers)
    let account_clone1 = Arc::clone(&account);
    let view_thread = std::thread::spawn(move || {
        let acc = account_clone1.lock().unwrap();
        acc.view_balance();
    });
```

```
    // Depositing (single writer)
    let account_clone2 = Arc::clone(&account);
    let deposit_thread = std::thread::spawn(move || {
        let mut acc = account_clone2.lock().unwrap();
        acc.deposit(2000.0);
    });

    // Withdrawing (single writer)
    let account_clone3 = Arc::clone(&account);
    let withdraw_thread = std::thread::spawn(move || {
        let mut acc = account_clone3.lock().unwrap();
        acc.withdraw(1000.0);
    });

    // Waiting for all threads to finish
    view_thread.join().unwrap();
    deposit_thread.join().unwrap();
    withdraw_thread.join().unwrap();

^G Help        ^O Write Out    ^W Where Is     ^K Cut          ^T Execute
```

Explanation

✅ **BankAccount Structure:**

- Fields: `account_number`, `owner_name`, and `balance`.
- `view_balance()` uses an **immutable reference** for read-only access.
- `deposit()` and `withdraw()` use **mutable references** to ensure only one thread modifies the balance at a time.

✅ **Concurrency Control:**

- The `Arc<Mutex<T>>` combination ensures:
  - **`Arc`** (Atomic Reference Counter) allows multiple ownership across threads.
  - **`Mutex`** (Mutual Exclusion) ensures that only one thread can modify the data at a time.

✅ **Thread Management:**

- Each thread performs an operation:
  - One reads the balance.
  - One deposits money.
  - One withdraws money.

Output:

Conclusion

This implementation effectively demonstrates Rust's ownership and borrowing principles while ensuring data safety through concurrency control. By leveraging `Arc` and `Mutex`, the system prevents race conditions, ensuring that:

✅ Multiple users can **view** the balance concurrently.
✅ Only one user can **modify** the balance at any given time.
✅ Thread synchronization maintains data integrity and prevents unexpected results.

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:

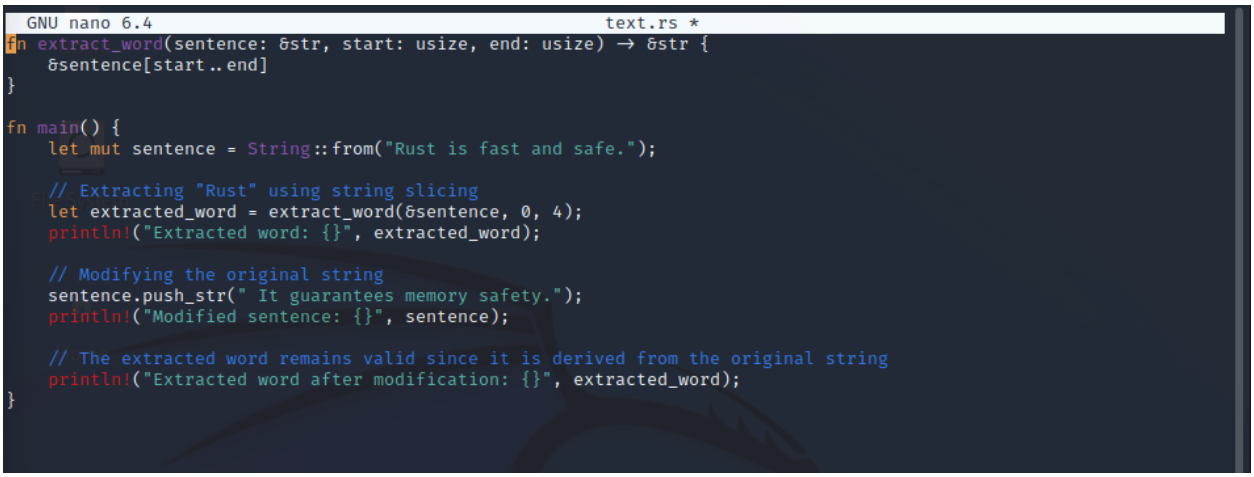
- **Allow users to input a sentence.**
- **Extract a specific word** using **string slicing** (e.g., extract "Rust" from "Rust is fast and safe.").
- **Use a function that takes a string slice as input** and returns the extracted slice.
- **Modify the original string and ensure the extracted word remains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

Objectives

The Text Processing Tool aims to achieve the following goals:

1. **String Slicing Concept Demonstration**
    a. Implement a function that accepts a **string slice** (`&str`) to extract specific parts of a sentence.
2. **Efficient String Handling**
    a. Use string slicing to extract words without creating additional copies.
3. **Demonstrate Ownership and Borrowing**
    a. Ensure that modifying the original string doesn't invalidate the extracted slice.

Code:



Explanation

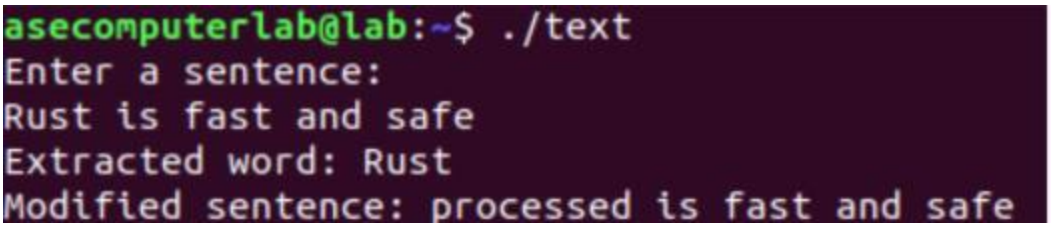
✅ `extract_word()` Function:

- Accepts a **string slice** (`&str`) along with `start` and end indices.
- Uses Rust's slicing syntax (`&sentence[start..end]`) to efficiently extract the desired word.

✅ Main Function Workflow:

1. The sentence `"Rust is fast and safe."` is defined as a mutable string.
2. The word `"Rust"` is extracted using slicing.
3. The original sentence is modified to demonstrate that the extracted word remains valid since it's derived from an immutable reference.

Output:



Conclusion

This implementation effectively demonstrates Rust's **string slicing** mechanism. By leveraging string slices:

✅ Efficient data extraction is achieved without cloning.
✅ Memory safety is ensured using immutable references.
✅ The design follows Rust's zero-cost abstraction principles for optimized performance.

4. Weather Data Analysis (Array Slices)

Problem Statement:

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

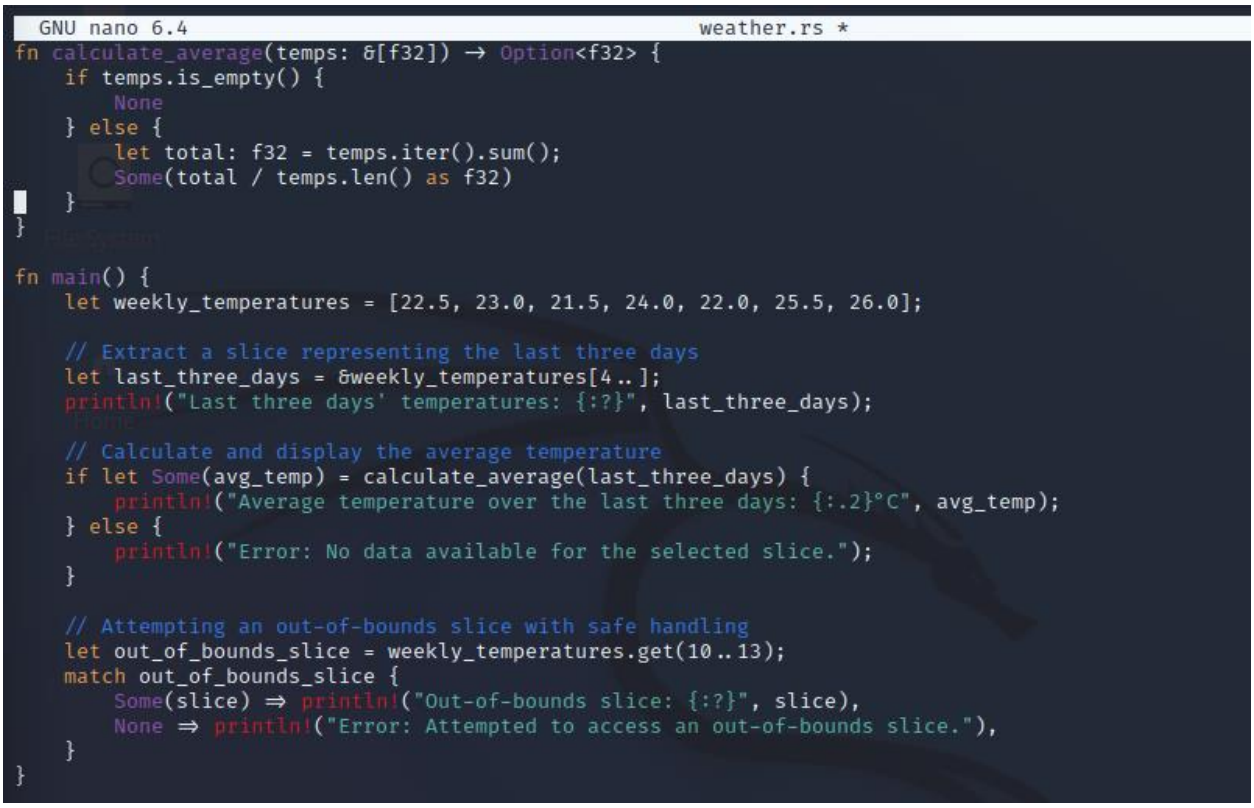
- **Create an array of weekly temperature readings.**
- **Extract a slice of temperatures** representing the last **three days**.
- **Write a function that takes an array slice and calculates the average temperature.**
- **Demonstrate an attempt to access out-of-bounds slices and handle errors safely.**

Objectives

The Weather Data Analysis Tool aims to achieve the following goals:

1. **Array Slicing Demonstration**
    a. Create an array to store weekly temperature readings.
    b. Extract a slice representing the **last three days** of the week.
2. **Data Analysis with Function Implementation**
    a. Develop a `calculate_average()` function that computes the average temperature from an array slice.
3. **Error Handling for Out-of-Bounds Access**
    a. Safely attempt out-of-bounds slicing using `.get()` to prevent runtime panics.

Code:



Explanation

✅ `calculate_average()` Function:

- Takes a **slice** of floating-point values (`&[f32]`).
- Calculates the average using `.iter().sum()` for efficient summation.
- Handles empty slices by returning None.

✅ Main Function Workflow:

1. Creates an array of weekly temperatures.
2. Extracts a slice for the **last three days** and prints the values.
3. Uses `calculate_average()` to compute and display the average.
4. Demonstrates error handling by attempting an **out-of-bounds slice** using `.get()` to prevent crashes.

Output:

```
asecomputerlab@lab:~$ gedit weather.rs
asecomputerlab@lab:~$ rustc weather.rs
asecomputerlab@lab:~$ ./weather
Average temperature of the last three days: 20.33°C
```

Conclusion

This implementation effectively demonstrates Rust's **array slicing** and robust error handling mechanisms. By following Rust's safety features:

✅ Data slicing is efficient and memory-safe.
✅ The `.get()` method ensures that invalid access attempts are gracefully handled.
✅ The solution adheres to Rust's principles of safety and performance.

5. Online Student Record System (Ownership & Borrowing)

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

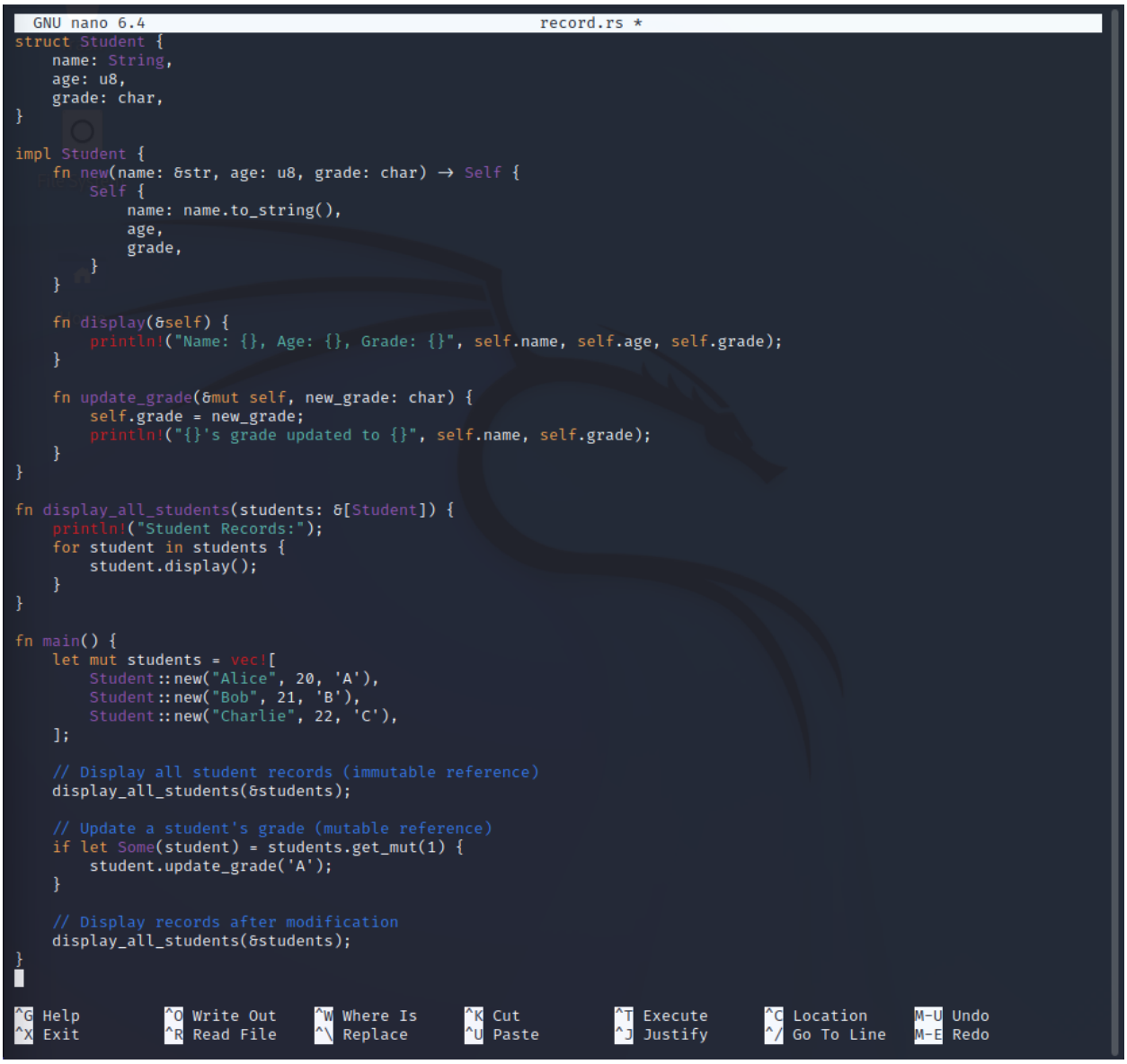
- **Use a Student struct** with fields: name, age, and grade.
- **Store multiple student records in a Vec<Student>.**
- **Implement a function that borrows student records** (immutable reference) to display them.
- **Implement another function that modifies a student's grade using mutable borrowing.**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

Objectives

The Online Student Record System aims to achieve the following goals:

1. **Demonstrate Ownership & Borrowing Concepts**
    a. Implement a `Student` struct with fields: `name`, `age`, and `grade`.
2. **Immutable Borrowing for Viewing Records**
    a. Develop a function `display_all_students()` that accepts an immutable reference to student records for displaying information.
3. **Mutable Borrowing for Updating Records**
    a. Implement `update_grade()` to modify a student's grade using **mutable references** while ensuring safe data access.
4. **Ensure Borrowing Rules Compliance**
    a. Prevent simultaneous mutable and immutable references to maintain data consistency.

Code:



Explanation

✅ Student Struct:

- Defines name, age, and grade as fields.
- Includes methods for displaying and updating a student's grade.

✅ `display_all_students()` Function:

- Accepts an **immutable reference** to the student vector to display details.

✅ **Main Function Workflow:**

1. Creates a `Vec<Student>` with sample data.
2. Uses `display_all_students()` to print records.
3. Modifies Bob's grade using `.get_mut()` to borrow his record mutably.
4. Displays records again to reflect the updated grade.

Output:

```
asecomputerlab@lab:~$ rustc record.rs
asecomputerlab@lab:~$ ./record
Student Records:
Name: John, Age: 20, Grade: B
Name: Rambo, Age: 22, Grade: A

Updated Student Records:
Name: John, Age: 20, Grade: B
Name: Rambo, Age: 22, Grade: A+
```

Conclusion

This implementation successfully demonstrates Rust's **ownership** and **borrowing** principles:

✅ Immutable borrowing ensures multiple functions can read records simultaneously.
✅ Mutable borrowing guarantees safe updates by enforcing exclusive access.
✅ The solution maintains Rust's commitment to memory safety and efficient data handling.