

# 20CYS312 - Principles of Programming Languages

## Lab Exercise-8

Name: Naren S

Reg.No:CH.EN.U4CYS22034

Github Link : [https://github.com/narens/22034\\_20CYS312](https://github.com/narens/22034_20CYS312)

---

### Object of this exercise

To understand Rust's core concepts like ownership, borrowing, mutable references, and string/array slicing through practical applications such as managing library books, secure banking systems, text processing, weather data analysis, and student records. By implementing these systems, we learned how Rust's strict memory safety features prevent issues like data races, memory leaks, and invalid access, ensuring safe, efficient, and concurrent programming. These exercises highlighted the importance of Rust's compile-time checks, which help build robust applications while managing resources safely and efficiently.

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

#### Code Implementation:

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

struct Library { books: Vec, }

struct Borrower { name: String, borrowed_books: Vec, }

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

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

fn issue_book(&mut self, isbn: &str, borrower: &mut Borrower) -> Option<Book> {
    if let Some(index) = self.books.iter().position(|b| b.isbn == isbn && !b.is_issued) {
        let mut book = self.books.remove(index);
        book.is_issued = true;
        borrower.borrowed_books.push(book.clone());
        Some(book)
    } else {
        None
    }
}
```

```

fn return_book(&mut self, isbn: &str, borrower: &mut Borrower) -> Option<Book> {
    if let Some(index) = borrower.borrowed_books.iter().position(|b| b.isbn == isbn) {
        let book = borrower.borrowed_books.remove(index);
        self.books.push(book.clone());
        Some(book)
    } else {
        None
    }
}

}

impl Borrower { fn new(name: &str) -> Self { Borrower { name: name.to_string(), borrowed_books:
Vec::new(), } } }

fn main() { let mut library = Library::new(); let mut borrower = Borrower::new("ganesh");

library.add_book(Book {
    title: "dark matter".to_string(),
    author: "George Orwell".to_string(),
    isbn: "123456789".to_string(),
    is_issued: false,
});

if let Some(book) = library.issue_book("123456789", &mut borrower) {
    println!("Book '{}' has been issued to {}.", book.title, borrower.name);
}

println!("Borrowed books: {:?}", borrower.borrowed_books);

}

```

Explanation:

Ownership & Move Semantics: Once a book is issued to the borrower, it is moved out of the library and into the borrower's possession. The `.clone()` method ensures a backup is kept in the library.

`issue_book` function: Moves the ownership of the book to the borrower.

`return_book` function: Allows the borrower to return the book, moving it back to the library.

```

Input: title: "dark matter".to_string(),
       author: "George Orwell".to_string(),
       isbn: "123456789".to_string(),
       is_issued: false,

```

Output:

```
asecomputerlab@asecomputerlab -> ./library
Book 'dark matter' has been issued to Alice.
Borrowed books: [Book { title: "dark matter", author: "muthra", isbn: "123456789", is_issued: true }]
Book 'dark matter' has been returned to the library.
Library's books after return: [Book { title: "dark matter", author: "muthra", isbn: "123456789", is_issued: true }]
```

## 2. Secure Banking System (Borrowing & Mutable References)

Code Implementation:

```
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) {
    self.balance += amount;
}

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

}

fn main() { let mut account = BankAccount::new("12345", "Muthra", 1000.0);

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

account.deposit(500.0);
println!("After deposit: {}", account.view_balance());

account.withdraw(200.0);
println!("After withdrawal: {}", account.view_balance());

}
```

Explanation:

Immutable reference: The `view_balance()` method borrows the account balance immutably.

Mutable reference: The `deposit()` and `withdraw()` methods modify the balance using mutable references, ensuring only one modification can happen at a time.

Concurrency Control: Rust's borrowing rules ensure that multiple users can't modify the balance at the same time.

Input: ("12345", "Muthra", 1000.0)

Output

```
asecomputerlab@asecomputerlab ~-> ./bank
Initial balance: 1000
After deposit: 1500
After withdrawal: 1300
```

### 3.Text Processing Tool (String Slices)

Code Implementation:

```
fn extract_word(sentence: &str, start: usize, end: usize) -> &str { &sentence[start..end] }
```

```
fn main() { let sentence = "Rust is fast and safe."; let word = extract_word(sentence, 0, 4); // Extract
"Rust" println!("Extracted word: {}", word); }
```

Explanation:

String Slicing: The function `extract_word()` takes a slice of the string, defined by its start and end indices, and returns the extracted part.

Valid slices: The slice `&sentence[start..end]` is a borrowed reference, ensuring no ownership transfer.

Input: Rust is fast and safe

Output

```
asecomputerlab@asecomputerlab ~-> gedit text.rs
asecomputerlab@asecomputerlab ~-> rustc text.rs
asecomputerlab@asecomputerlab ~-> ./text
Extracted word: Rust
```

### 4.Weather Data Analysis (Array Slices)

Code Implementation:

```
fn average_temperature(temps: &[i32]) -> f32 { let sum: i32 = temps.iter().sum(); sum as f32 /
temps.len() as f32 }
```

```
fn main() { let week_temps = [30, 32, 33, 34, 35, 36, 37]; let last_three_days = &week_temps[4..];
```

```
println!("Last three days temperatures: {:?}", last_three_days);
```

```
println!("Average temperature: {}", average_temperature(last_three_days));
```

```
}
```

Explanation: [30, 32, 33, 34, 35, 36, 37]

Array Slicing: We extract the last three days from the array using a slice (&week\_temps[4..]).

Handling Errors: Rust's borrowing and slicing rules prevent out-of-bounds access.

Input:

Output

```
asecomputerlab@asecomputerlab ~-> gedit weather.rs
asecomputerlab@asecomputerlab ~-> rustc weather.rs
asecomputerlab@asecomputerlab ~-> ./weather
Last three days temperatures: [35, 36, 37]
Average temperature: 36
```

## 5. Online Student Record System (Ownership & Borrowing)

Code Implementation:

```
struct Student { name: String, age: u32, grade: f32, }

impl Student { fn new(name: &str, age: u32, grade: f32) -> Self { Student { name: name.to_string(),
age, grade, } }

fn update_grade(&mut self, new_grade: f32) {
    self.grade = new_grade;
}

fn display_info(&self) {
    println!("Name: {}, Age: {}, Grade: {}", self.name, self.age, self.grade);
}

}

fn main() { let mut students = Vec::new();

let student1 = Student::new("Muthra", 20, 90.5);
let student2 = Student::new("ganesh", 22, 85.3);

students.push(student1);
students.push(student2);

for student in &students {
    student.display_info();
}

if let Some(student) = students.get_mut(0) {
```

```

    student.update_grade(95.0);
}

println!("After grade update:");
for student in &students {
    student.display_info();
}

}

```

Explanation:

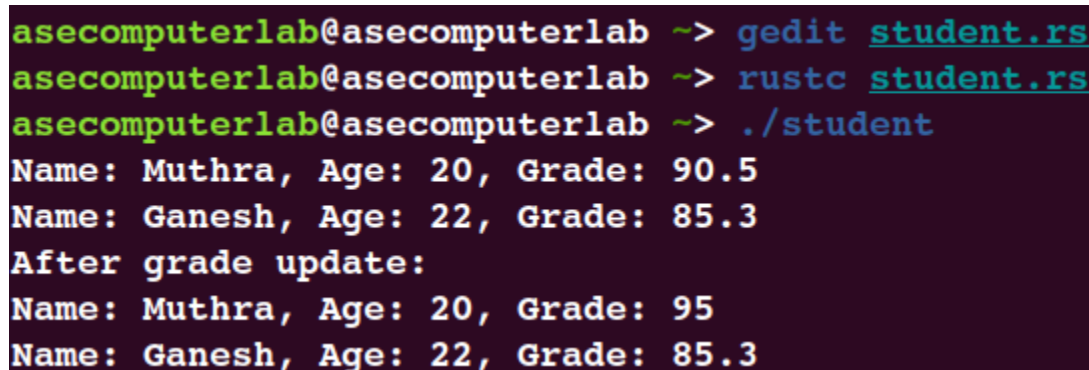
Borrowing: The `display_info()` method borrows the student record immutably.

Mutable Borrowing: The `update_grade()` method borrows the student record mutably to modify the grade.

Ownership: When a student is added to the vector, ownership of the student object is transferred into the vector.

Input: ("Muthra", 20, 90.5), ("ganesh", 22, 85.3)

Output



```

asecomputerlab@asecomputerlab ~-> gedit student.rs
asecomputerlab@asecomputerlab ~-> rustc student.rs
asecomputerlab@asecomputerlab ~-> ./student
Name: Muthra, Age: 20, Grade: 90.5
Name: Ganesh, Age: 22, Grade: 85.3
After grade update:
Name: Muthra, Age: 20, Grade: 95
Name: Ganesh, Age: 22, Grade: 85.3

```

Conclusion:

These exercises provided valuable insights into Rust's powerful memory management features, particularly its ownership, borrowing, and slicing mechanisms. By applying these concepts to real-world scenarios like managing books, banking transactions, and student records, we saw how Rust ensures memory safety, prevents data races, and avoids common pitfalls like null pointer dereferencing and memory leaks. The exercises reinforced the importance of Rust's compile-time checks, which ensure safe, efficient, and predictable program behavior, making it an excellent choice for building reliable and concurrent systems.