20CYS312 -PRINCIPLE OF PROGRAMMING LANGUAGES

Date: 14-03-2025

Name: Naren S

Roll no.: CH.EN.U4CYS22034

Github: https://github.com/narenss/22034_20CYS312

LAB-9

Objective:

In this lab exercise, I will explore various fundamental concepts in Rust such as decision making with if-else, using match expressions, loops, pattern matching, working with tuples, vectors, structs, and enums.

1. Nested Decision Making with if-else

Problem: Write a Rust program that takes a person's age and income as input and determines their eligibility for a loan. The program should check: If the person is below 21, they are ineligible. If between 21 and 60, they are eligible based on income (> ₹50,000). If above 60, they need a guarantor.

Code:

main.rs

```rust
use std::io;

fn main() {
    let mut age_input = String::new();
    let mut income_input = String::new();

    // Taking age input
    println!("Enter your age:");
    io::stdin().read_line(&mut age_input).expect("Failed to read line");
    let age: u32 = age_input.trim().parse().expect("Please enter a valid number");

    // Taking income input
    println!("Enter your income:");
    io::stdin().read_line(&mut income_input).expect("Failed to read line");
    let income: u32 = income_input.trim().parse().expect("Please enter a valid number");

    // Loan eligibility based on age and income
    if age < 21 {
        println!("You are ineligible for a loan.");
    } else if age >= 21 && age <= 60 {
        if income > 50000 {
            println!("You are eligible for a loan.");
        } else {
            println!("You are ineligible for a loan due to insufficient income.");
        }
    } else {
        println!("You are eligible for a loan, but you need a guarantor.");
    }
}
```

Output:

Output	Output
Enter your age: 20 Enter your income: 55000 You are ineligible for a loan. === Code Execution Successful ===	Enter your age: 25 Enter your income: 2342342342 You are eligible for a loan. === Code Execution Successful ===

Explanation:

1. The program starts by taking the user's age and income as inputs.
2. It then checks the eligibility criteria for the loan using nested if-else statements:
    a. If the person is below 21, they are ineligible.
    b. If the person is between 21 and 60 years old, eligibility is based on whether their income is greater than ₹50,000.
    c. If the person is above 60, they need a guarantor.

2. Using match with Complex Cases

Problem: Implement a restaurant billing system where a user enters a menu item (e.g., "Burger", "Pizza", "Pasta"), and the program prints the price. Use a match expression with additional conditions to apply discounts based on the quantity ordered.

Code:

main.rs

```rust
use std::io;

fn main() {
    let mut menu_item = String::new();
    let mut quantity_input = String::new();

    // Taking menu item and quantity input
    println!("Enter the menu item (Burger, Pizza, Pasta):");
    io::stdin().read_line(&mut menu_item).expect("Failed to read line");
    println!("Enter the quantity:");
    io::stdin().read_line(&mut quantity_input).expect("Failed to read line");

    let quantity: u32 = quantity_input.trim().parse().expect("Please enter a valid number");

    let price = match menu_item.trim() {
        "Burger" => 100,
        "Pizza" => 200,
        "Pasta" => 150,
        _ => {
            println!("Invalid menu item.");
            return;
        }
    };

    let total_price = price * quantity;

    let discount = if quantity >= 3 {
        0.1 // 10% discount for 3 or more items
    } else {
        0.0
    };

    let final_price = total_price as f32 * (1.0 - discount);

    println!("The total price is: ₹{}", final_price);
}
```

Output:

Output	Output	Output
Enter the menu item (Burger, Pizza, Pasta): Burger Enter the quantity: 5 The total price is: ₹450 === Code Execution Successful ===	Enter the menu item (Burger, Pizza, Pasta): Pasta Enter the quantity: 100 The total price is: ₹13500 === Code Execution Successful ===	Enter the menu item (Burger, Pizza, Pasta): Pizza Enter the quantity: 20 The total price is: ₹3600 === Code Execution Successful ===

Explaination:

- The program prompts the user to enter a menu item and quantity.
- Using a match expression, it finds the price of the selected menu item.
- If 3 or more items are ordered, a 10% discount is applied.
- The final price is calculated after applying the discount.

3. Using Loops for Data Processing (Fibonacci Numbers)

Problem: Write a Rust program to generate Fibonacci numbers up to a given n using a for loop. Store the sequence in a list (vector) and print the values.

Code and Output Examples:

main.rs

```rust
fn main() {
    let n = 10;  // Fibonacci sequence up to 10th number
    let mut fib_sequence = vec![0, 1];

    for i in 2..n {
        let next = fib_sequence[i - 1] + fib_sequence[i - 2];
        fib_sequence.push(next);
    }

    println!("Fibonacci sequence: {:?}", fib_sequence);
}
```

Output

```
Fibonacci sequence: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]

=== Code Execution Successful ===
```

Explanation:

- The program generates Fibonacci numbers up to the n-th term using a for loop.
- It stores the sequence in a vector and prints it.

4. Pattern Matching in Loops with while let

Problem: Implement a program where a user enters multiple numbers, and the program keeps adding them to a list until they enter 0. Use while let to process the list and print only the even numbers.

Code and Output Examples:

main.rs

```rust
use std::io;

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

    loop {
        let mut input = String::new();
        println!("Enter a number (0 to stop):");
        io::stdin().read_line(&mut input).expect("Failed to read line");

        let num: i32 = input.trim().parse().expect("Please enter a valid number");

        if num == 0 {
            break;
        } else {
            numbers.push(num);
        }
    }

    println!("Even numbers entered:");

    // Using while let to filter even numbers
    let mut index = 0;
    while let Some(num) = numbers.get(index) {
        if num % 2 == 0 {
            println!("{}", num);
        }
        index += 1;
    }
}
```

Output

```
Enter a number (0 to stop):
4
Enter a number (0 to stop):
3
Enter a number (0 to stop):
2
Enter a number (0 to stop):
1
Enter a number (0 to stop):
-9
Enter a number (0 to stop):
0
Even numbers entered:
4
2

=== Code Execution Successful ===
```

Explaination:

- The program takes user input and adds numbers to a vector until the user enters 0.
- It then uses a while let loop to process the numbers and print only the even ones.

5. Tuple Manipulation in a Real-World Scenario

Problem: Create a tuple representing an employee's data (ID, Name, Salary). Write a function that takes this tuple as input, applies a 10% salary hike if salary < ₹50,000, and returns an updated tuple.

Code and Output Examples:

main.rs

```rust
fn update_salary(employee: (u32, &str, f64)) -> (u32, String, f64)
    {
    let (id, name, salary) = employee;
    let new_salary = if salary < 50000.0 { salary * 1.10 } else {
        salary };
    (id, name.to_string(), new_salary)
}

fn main() {
    let employee = (22034, "NAREN S", 45000.0);
    let updated_employee = update_salary(employee);
    println!("Updated Employee Data: {:?}", updated_employee);
}
```

Output

```
Updated Employee Data: (22034, "NAREN S", 49500.00000000001)

=== Code Execution Successful ===
```

Explaination:

- The function update_salary takes an employee tuple (ID, Name, Salary).
- It checks if the salary is below ₹50,000 and applies a 10% hike if true.
- The updated tuple is returned with name converted to String for mutability.
- The main function creates an employee, updates the salary, and prints the result.

6. Vector (List) Operations with Iterators

Problem: Write a Rust program that maintains a list of temperatures recorded in a city for a week. Implement: A function that finds the average temperature. Another function that finds the highest and lowest temperature using iterators.

Code and Output Examples:

main.rs

```rust
fn average_temperature(temps: &Vec<f64>) -> f64 {
    let sum: f64 = temps.iter().sum();
    sum / temps.len() as f64
}

fn min_max_temperature(temps: &Vec<f64>) -> (f64, f64) {
    let min_temp = temps.iter().cloned().fold(f64::INFINITY, f64::min);
    let max_temp = temps.iter().cloned().fold(f64::NEG_INFINITY, f64::max);
    (min_temp, max_temp)
}

fn main() {
    let temperatures = vec![30.5, 32.0, 31.8, 15.5, 30.0, 33.2, 34.1,36.0];

    let avg_temp = average_temperature(&temperatures);
    let (min_temp, max_temp) = min_max_temperature(&temperatures);

    println!("Average Temperature: {:.2}", avg_temp);
    println!("Lowest Temperature: {:.2}", min_temp);
    println!("Highest Temperature: {:.2}", max_temp);
}
```

Output

```
Average Temperature: 30.39
Lowest Temperature: 15.50
Highest Temperature: 36.00

=== Code Execution Successful ===
```

Explaination:

- average_temperature: Computes the sum of elements and divides by the count.
- min_max_temperature: Uses fold() to iterate and find min/max values.
- main: Initializes a temperature list, calls both functions, and prints results.

7. Structs with Methods

Problem: Define a struct named BankAccount with fields account_number, holder_name, and balance. o Implement methods for: Depositing money. Withdrawing money (with balance check). Displaying account details.

Code and Output Examples:

main.rs

```rust
struct BankAccount {
    account_number: u32,
    holder_name: String,
    balance: f64,
}

impl BankAccount {
    fn deposit(&mut self, amount: f64) {
        self.balance += amount;
        println!("₹{:.2} deposited. New Balance: ₹{:.2}", amount, self.balance);
    }

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

    fn display_details(&self) {
        println!(
            "Account Number: {}\nHolder Name: {}\nBalance: ₹{:.2}",
            self.account_number, self.holder_name, self.balance
        );
    }
}

fn main() {
    let mut account = BankAccount {
        account_number: 22034,
        holder_name: "Naren".to_string(),
        balance: 10000.0,
    };

    account.display_details();
    account.deposit(2000.0);
    account.withdraw(5000.0);
    account.withdraw(20000.0);
}
```

Output

```
Account Number: 22034
Holder Name: Naren
Balance: ₹10000.00
₹2000.00 deposited. New Balance: ₹12000.00
₹5000.00 withdrawn. New Balance: ₹7000.00
Insufficient balance!

=== Code Execution Successful ===
```

Explaination:

- Defines a BankAccount struct with fields: account_number, holder_name, and balance.
- Implements methods:
- **deposit**: Adds money to the balance.
- **withdraw**: Deducts money if sufficient balance exists.
- **display_details**: Prints account details.
- In main, an account is created, and the methods are called to simulate deposits and withdrawals.

8. Structs and Enums Together – Vehicle Registration System

Problem: Define an enum named FuelType with variants Petrol, Diesel, and Electric. Define a struct named Vehicle with fields brand, model, and fuel_type. Implement a function that takes a list of vehicles and filters only electric vehicles for display.

Code and Output Examples:

main.rs

```rust
enum FuelType {
    Petrol,
    Diesel,
    Electric,
}

struct Vehicle {
    brand: String,
    model: String,
    fuel_type: FuelType,
}

fn filter_electric_vehicles(vehicles: &Vec<Vehicle>) -> Vec<&Vehicle> {
    vehicles
        .iter()
        .filter(|v| matches!(v.fuel_type, FuelType::Electric))
        .collect()
}

fn main() {
    let vehicles = vec![
        Vehicle {
            brand: "Tesla".to_string(),
            model: "Model S".to_string(),
            fuel_type: FuelType::Electric,
        },
        Vehicle {
            brand: "Ford".to_string(),
            model: "Mustang".to_string(),
            fuel_type: FuelType::Petrol,
        },
        Vehicle {
            brand: "Hyundai".to_string(),
            model: "Kona EV".to_string(),
            fuel_type: FuelType::Diesel,
        },
    ];

    let electric_vehicles = filter_electric_vehicles(&vehicles);

    println!("Electric Vehicles:");
    for ev in electric_vehicles {
        println!("Brand: {}, Model: {}", ev.brand, ev.model);
    }
}
```

Output

```
Electric Vehicles:
Brand: Tesla, Model: Model S

=== Code Execution Successful ===
```

```
Electric Vehicles:
Brand: Tesla, Model: Model S
Brand: Ford, Model: Mustang

=== Code Execution Successful ===
```

Explaination:

- Defines an enum FuelType with variants: Petrol, Diesel, and Electric.
- Defines a Vehicle struct with brand, model, and fuel_type.
- Implements filter_electric_vehicles to extract only electric vehicles using iter() and filter().
- In main, a list of vehicles is created, filtered, and printed if they are electric.