

20CYS312 – PPL

Lab Exercise 9 - Rust Lab Exercise Questions

Name:Sushant Yadav

RegNo:En.u4cys22067

Date:2025/03/14

Course Code: 20CYS312

1. Nested Decision Making with if-else

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.

Objectives:

- Understand conditional statements (if-else).
- Implement nested decision-making logic.
- Work with user input and type conversions.

Code:



```
use std::io;

fn main() {
    let mut age = String::new();
    let mut income = String::new();

    println!("Enter age: ");
    io::stdin().read_line(&mut age).expect("Failed to read line");
    let age: u32 = age.trim().parse().expect("Please type a number");

    println!("Enter income: ");
    io::stdin().read_line(&mut income).expect("Failed to read line");
    let income: u32 = income.trim().parse().expect("Please type a number");

    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!("Income is too low for a loan.");
        }
    } else {
        println!("You need a guarantor for the loan.");
    }
}
```

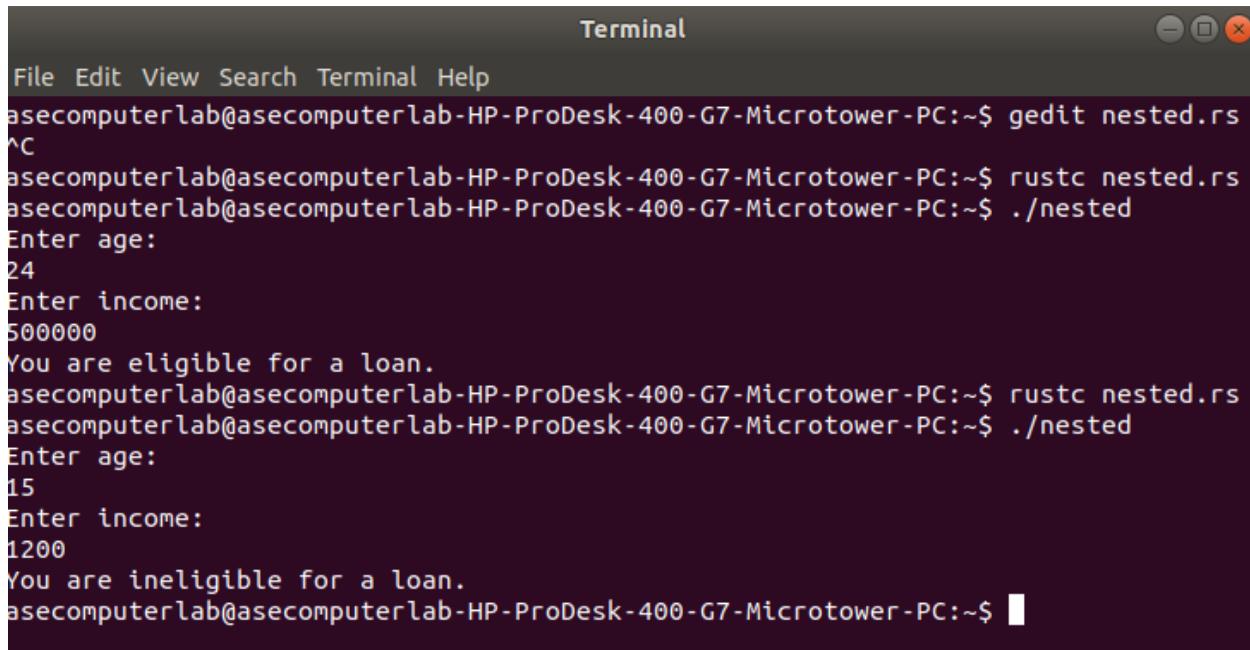
Explanation:

This program checks the person's eligibility for a loan based on their age and income. It uses nested if-else statements to cover different age and income conditions:

- If under 21, ineligible for a loan.
- If between 21 and 60, checks income for eligibility.

- If above 60, requires a guarantor.

Output:



```
Terminal
File Edit View Search Terminal Help
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ gedit nested.rs
^C
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ rustc nested.rs
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ ./nested
Enter age:
24
Enter income:
500000
You are eligible for a loan.
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ rustc nested.rs
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ ./nested
Enter age:
15
Enter income:
1200
You are ineligible for a loan.
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$
```

Conclusion:

This program showcases how to use nested if-else statements in Rust to make decisions based on multiple conditions.

2. Using match with Complex Cases

Implement a restaurant billing system where a user enters a menu item (e.g.,

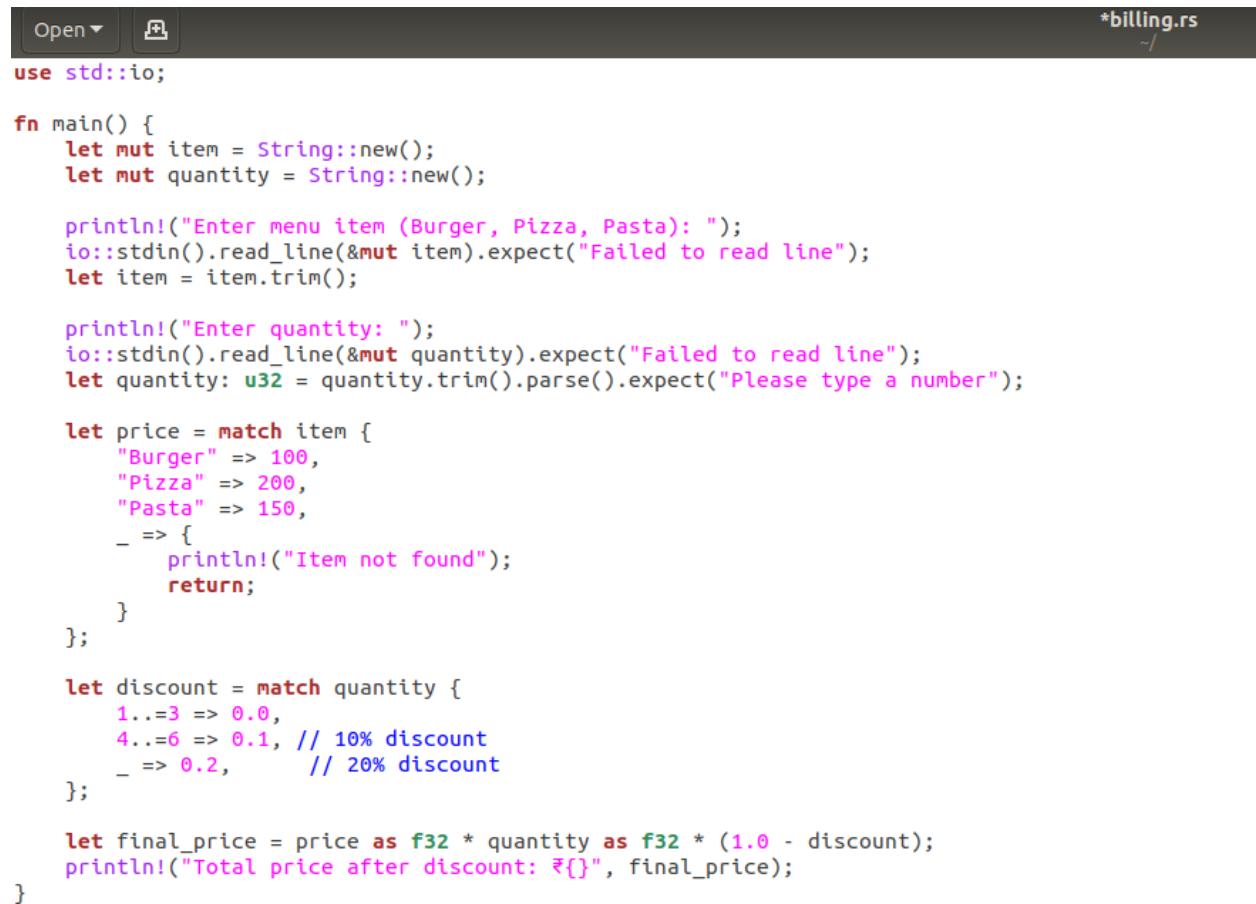
"Burger", "Pizza", "Pasta"), and the program prints the price.

>Use a match expression with additional conditions to apply discounts based on the quantity ordered.

Objectives:

- Understand the match statement with conditions.
- Implement complex matching with ranges.
- Use floating-point numbers to compute final amounts.

Code:



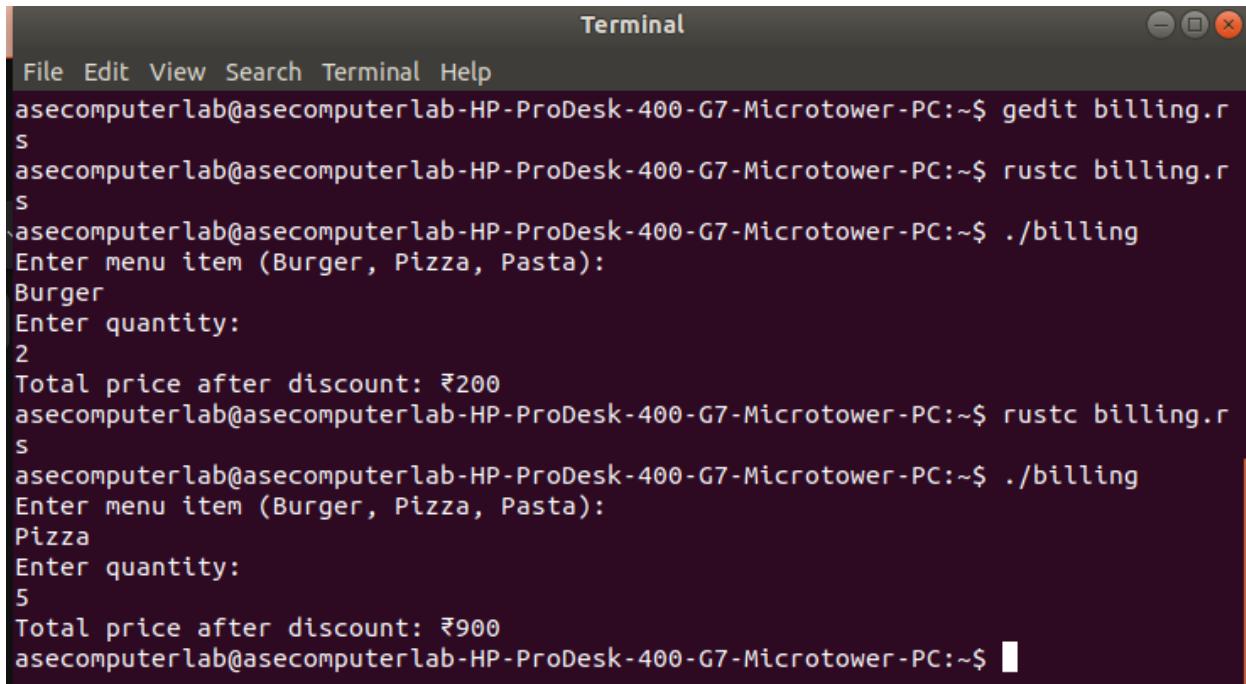
The screenshot shows a code editor window with a dark theme. The title bar says "*billing.rs". The code itself is written in Rust and performs the following tasks:

- Imports the `std::io` module.
- Defines a `main` function.
- Creates mutable `String` variables for `item` and `quantity`.
- Prompts the user to enter a menu item ("Burger", "Pizza", or "Pasta") and reads it from standard input.
- Prompts the user to enter a quantity and reads it from standard input.
- Calculates the price based on the menu item using a match expression:
 - "Burger" = 100
 - "Pizza" = 200
 - "Pasta" = 150
 - Any other value results in an "Item not found" message and returns.
- Calculates the discount based on the quantity using a match expression:
 - 1..=3 = 0.0 (no discount)
 - 4..=6 = 0.1 (10% discount)
 - 7..=10 = 0.2 (20% discount)
 - Any other value results in an "Item not found" message and returns.
- Computes the final price as the product of the price and quantity, applying the discount.
- Prints the total price after discount.

Explanation:

This program calculates the price of a meal based on the menu item and applies a discount based on the quantity ordered. It uses match to choose the price based on the item and a second match to apply the discount.

Output:



```
Terminal
File Edit View Search Terminal Help
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ gedit billing.rs
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ rustc billing.rs
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ ./billing
Enter menu item (Burger, Pizza, Pasta):
Burger
Enter quantity:
2
Total price after discount: ₹200
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ rustc billing.rs
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ ./billing
Enter menu item (Burger, Pizza, Pasta):
Pizza
Enter quantity:
5
Total price after discount: ₹900
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$
```

Conclusion:

This program demonstrates the power of match in handling complex conditions and scenarios in Rust.

3. Using Loops for Data Processing

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.

Objectives:

- Learn how to generate sequences with loops.
- Understand vector operations and indexing.
- Use the for loop effectively.

Code:

```
Open ~/
fn main() {
    let n = 10; // Generate Fibonacci numbers up to n
    let mut fib_seq = vec![0, 1];

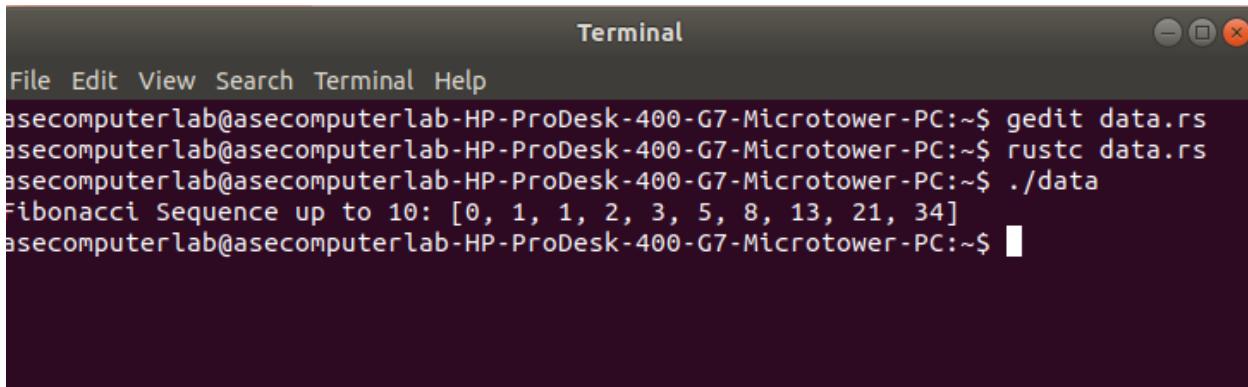
    for i in 2..n {
        let next = fib_seq[i - 1] + fib_seq[i - 2];
        fib_seq.push(next);
    }

    println!("Fibonacci Sequence up to {}: {:?}", n, fib_seq);
}
```

Explanation:

This program generates Fibonacci numbers up to n using a for loop and stores them in a Vec. It then prints the sequence.

Output:



The screenshot shows a terminal window titled "Terminal". The window has a dark background and light-colored text. At the top, there's a menu bar with "File", "Edit", "View", "Search", "Terminal", and "Help". Below the menu, the terminal prompt is "asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~\$". The user then runs three commands: "gedit data.rs", "rustc data.rs", and "./data". The output of the program is displayed as "Fibonacci Sequence up to 10: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]". The terminal ends with the prompt "asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~\$".

Conclusion:

The program uses the for loop to generate Fibonacci numbers and store them in a vector, which is a great exercise in both looping and vector manipulation.

4. Pattern Matching in Loops with while let

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.

Objectives:

- Understand while let for pattern matching in loops.
- Use Vec to collect and process data.
- Filter data during iteration.

Code:

```
use std::io;

fn main() {
    let mut numbers = Vec::new();
    loop {
        let mut num = String::new();
        println!("Enter a number (0 to stop): ");
        io::stdin().read_line(&mut num).expect("Failed to read line");
        let num: i32 = num.trim().parse().expect("Please type a number");

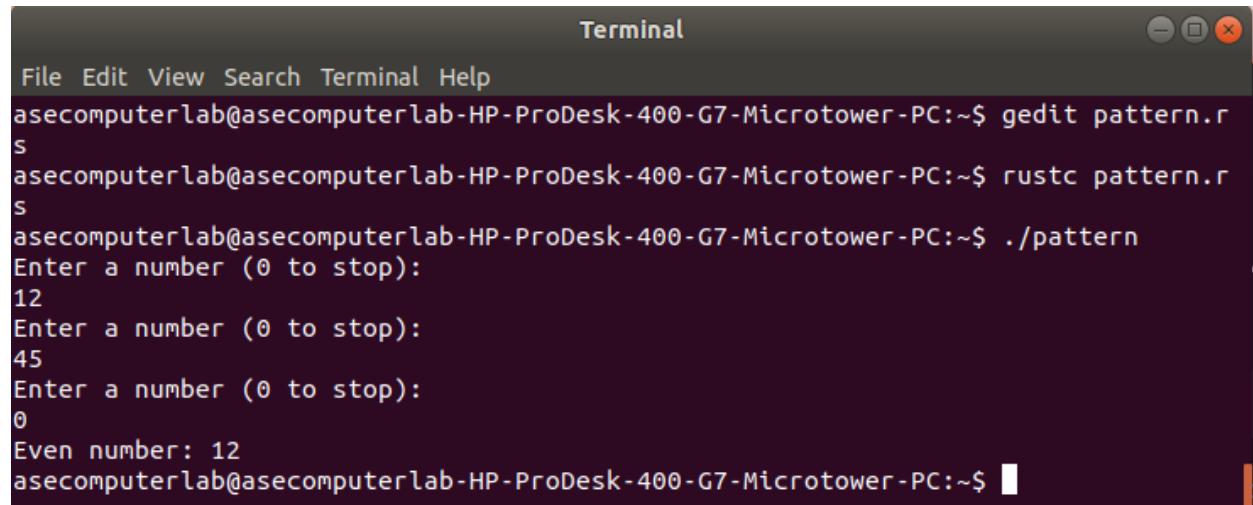
        if num == 0 {
            break;
        }
        numbers.push(num);
    }

    // Using while let to filter even numbers
    while let Some(num) = numbers.pop() {
        if num % 2 == 0 {
            println!("Even number: {}", num);
        }
    }
}
```

Explanation:

The program collects user input until the user enters 0. The program uses while let to pop numbers from the list and print only the even numbers.

Output:



A screenshot of a terminal window titled "Terminal". The window has a dark theme with white text. At the top, there's a menu bar with "File", "Edit", "View", "Search", "Terminal", and "Help". Below the menu, the terminal prompt shows the user's name and computer details: "asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~\$". The user runs three commands: "gedit pattern.rs", "rustc pattern.rs", and "./pattern". The output shows the program prompting for input ("Enter a number (0 to stop)"), receiving two inputs ("12" and "45"), and then printing the even number ("Even number: 12").

```
File Edit View Search Terminal Help
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ gedit pattern.rs
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ rustc pattern.rs
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ ./pattern
Enter a number (0 to stop):
12
Enter a number (0 to stop):
45
Enter a number (0 to stop):
0
Even number: 12
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$
```

Conclusion:

while let provides a concise way to handle specific conditions within loops, making this a good exercise for filtering data.

5. Tuple Manipulation in a Real-World Scenario

Create a tuple representing an employee's data (ID, Name, Salary).

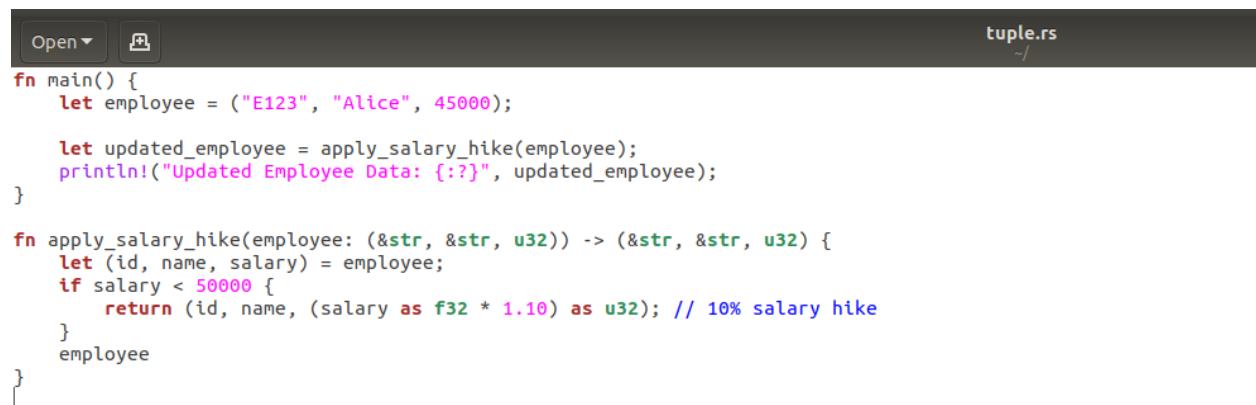
> Write a function that takes this tuple as input, applies a 10% salary hike if salary <

₹50,000, and returns an updated tuple.

Objectives:

- Work with tuples and pattern matching.
- Implement real-world business logic for data manipulation.
- Understand how to modify and return tuples.

Code:



A screenshot of a code editor showing a file named "tuple.rs". The code defines a main function that creates a tuple for an employee and prints it. It then defines an apply_salary_hike function that takes a tuple and returns a modified tuple where the salary is increased by 10% if it is less than 50,000. The code uses pattern matching to destructure the tuple and update the salary field.

```
fn main() {
    let employee = ("E123", "Alice", 45000);

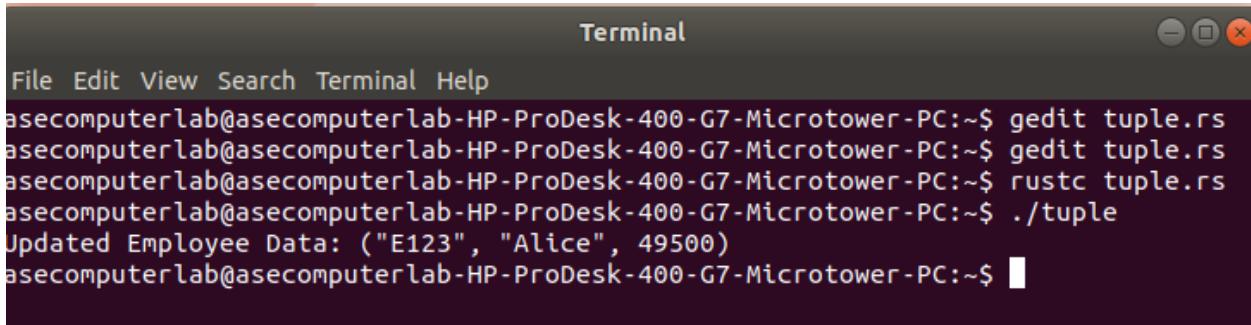
    let updated_employee = apply_salary_hike(employee);
    println!("Updated Employee Data: {:?}", updated_employee);
}

fn apply_salary_hike(employee: (&str, &str, u32)) -> (&str, &str, u32) {
    let (id, name, salary) = employee;
    if salary < 50000 {
        return (id, name, (salary as f32 * 1.10) as u32); // 10% salary hike
    }
    employee
}
```

Explanation:

This program defines a tuple for an employee and applies a salary hike if the salary is below ₹50,000. It returns an updated tuple.

Output:



```
Terminal
File Edit View Search Terminal Help
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ gedit tuple.rs
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ gedit tuple.rs
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ rustc tuple.rs
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ ./tuple
Updated Employee Data: ("E123", "Alice", 49500)
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$
```

Conclusion:

Using tuples is an efficient way to group related data, and this program demonstrates how to manipulate and return modified data in Rust.

6. Vector (List) Operations with Iterators

>Write a Rust program that maintains a list of temperatures recorded in a city for a

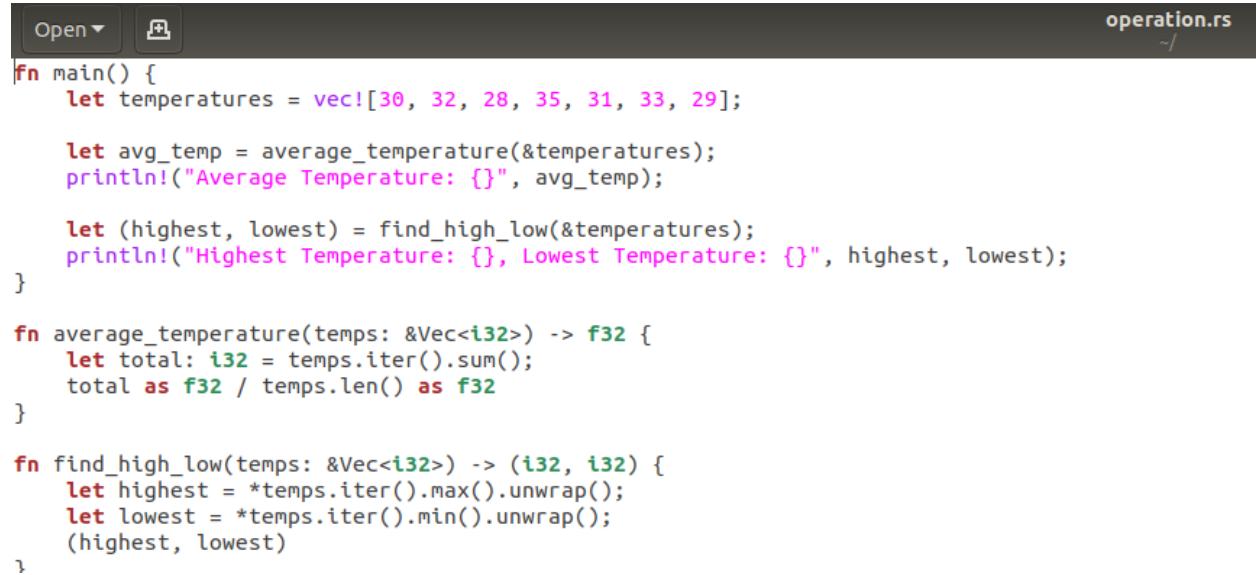
week. Implement:

>A function that finds the average temperature.

Objectives:

- Use vector iterators to process data.
- Implement functions that perform calculations using iterators.
- Calculate averages and find extremes using iterators.

Code:



```
operation.rs
fn main() {
    let temperatures = vec![30, 32, 28, 35, 31, 33, 29];

    let avg_temp = average_temperature(&temperatures);
    println!("Average Temperature: {}", avg_temp);

    let (highest, lowest) = find_high_low(&temperatures);
    println!("Highest Temperature: {}, Lowest Temperature: {}", highest, lowest);
}

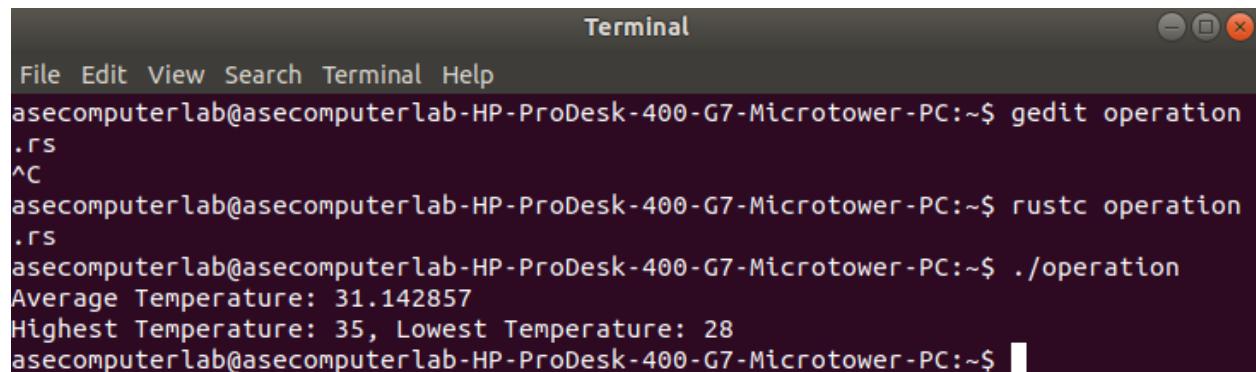
fn average_temperature(temp: &Vec<i32>) -> f32 {
    let total: i32 = temp.iter().sum();
    total as f32 / temp.len() as f32
}

fn find_high_low(temp: &Vec<i32>) -> (i32, i32) {
    let highest = *temp.iter().max().unwrap();
    let lowest = *temp.iter().min().unwrap();
    (highest, lowest)
}
```

Explanation:

This program uses vectors to store temperature data for a week. It calculates the average temperature and finds the highest and lowest temperatures using iterators.

Output:



```
Terminal
File Edit View Search Terminal Help
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ gedit operation.rs
^C
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ rustc operation.rs
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ ./operation
Average Temperature: 31.142857
Highest Temperature: 35, Lowest Temperature: 28
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$
```

Conclusion:

This program highlights how Rust's iterators can be used to efficiently process data in a vector.

7.Structs with Methods

Define a struct named BankAccount with fields account_number,holder_name, and balance.

>Implement methods for:

>Depositing money.

> Withdrawing money (with balance check).

>Displaying account details.

Objectives:

- Define structs with methods in Rust.
- Implement error handling with Result.
- Practice struct manipulation and method implementation.

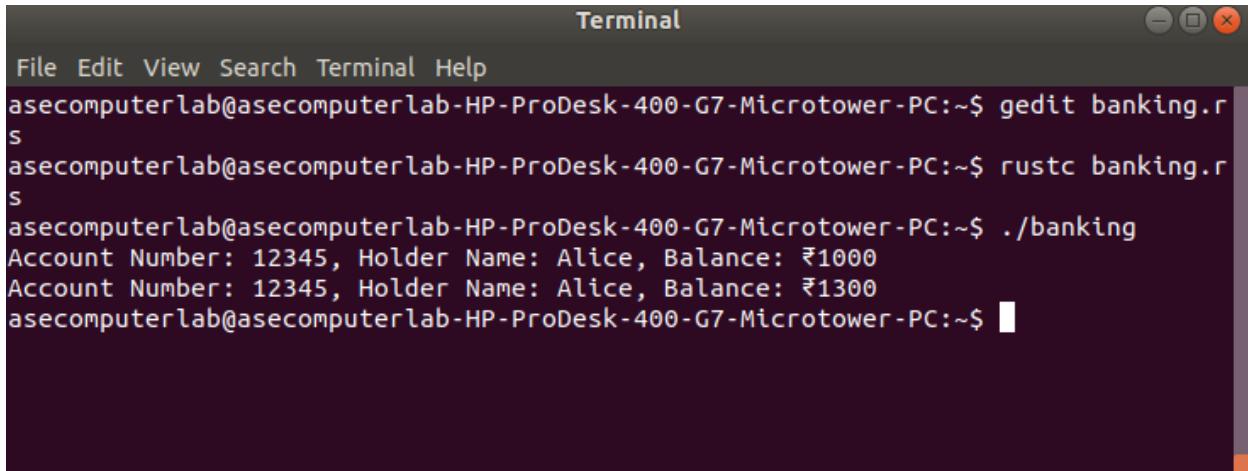
Code:

```
Open ▾  banking.rs ~/  
  
struct BankAccount {  
    account_number: u32,  
    holder_name: String,  
    balance: f32,  
}  
  
impl BankAccount {  
    fn deposit(&mut self, amount: f32) {  
        self.balance += amount;  
    }  
  
    fn withdraw(&mut self, amount: f32) -> Result<(), String> {  
        if amount > self.balance {  
            Err(String::from("Insufficient funds"))  
        } else {  
            self.balance -= amount;  
            Ok(())  
        }  
    }  
  
    fn display_account(&self) {  
        println!("Account Number: {}, Holder Name: {}, Balance: ₹{}",  
               self.account_number, self.holder_name, self.balance);  
    }  
}  
  
fn main() {  
    let mut account = BankAccount {  
        account_number: 12345,  
        holder_name: String::from("Alice"),  
        balance: 1000.0,  
    };  
  
    account.display_account();  
    account.deposit(500.0);  
    account.withdraw(200.0).unwrap();  
    account.display_account();  
}
```

Explanation:

This program defines a `BankAccount` struct with methods for depositing money, withdrawing money (with balance check), and displaying account details.

Output:



A screenshot of a terminal window titled "Terminal". The window has a dark background and light-colored text. At the top, there's a menu bar with "File", "Edit", "View", "Search", "Terminal", and "Help". Below the menu, the terminal prompt shows the user's name and computer details: "asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~\$". The user runs three commands: "gedit banking.rs", "rustc banking.rs", and "./banking". The output of the "./banking" command is displayed, showing two account entries: "Account Number: 12345, Holder Name: Alice, Balance: ₹1000" and "Account Number: 12345, Holder Name: Alice, Balance: ₹1300". The terminal ends with the prompt "asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~\$".

Conclusion:

This is a basic banking system in Rust, showcasing how to use structs, methods, and error handling to implement real-world logic.

8. Structs and Enums Together – Vehicle Registration System

- > 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

Objectives

To demonstrate the use of **Enums** for categorizing vehicle fuel types.

To implement a **Struct** that represents detailed vehicle information.
To apply efficient filtering using .filter() and the matches! macro to display only electric vehicles.

Code:

```
Open ▾  vechile.rs ~/  
enum FuelType {  
    Petrol,  
    Diesel,  
    Electric,  
}  
  
struct Vehicle {  
    brand: String,  
    model: String,  
    fuel_type: FuelType,  
}  
  
// Function to filter and display only electric vehicles  
fn display_electric_vehicles(vehicles: &Vec<Vehicle>) {  
    println!("Electric Vehicles:");  
    for vehicle in vehicles.iter().filter(|v| matches!(v.fuel_type, FuelType::Electric)) {  
        println!("Brand: {}, Model: {}", vehicle.brand, vehicle.model);  
    }  
}  
  
fn main() {  
    // List of sample vehicles  
    let vehicles = vec![  
        Vehicle {  
            brand: String::from("Tesla"),  
            model: String::from("Model S"),  
            fuel_type: FuelType::Electric,  
        },  
        Vehicle {  
            brand: String::from("Toyota"),  
            model: String::from("Corolla"),  
            fuel_type: FuelType::Petrol,  
        },  
        Vehicle {  
            brand: String::from("Nissan"),  
            model: String::from("Leaf"),  
            fuel_type: FuelType::Electric,  
        },  
        Vehicle {  
            brand: String::from("Hyundai"),  
            model: String::from("Creta"),  
            fuel_type: FuelType::Diesel,  
        },  
    ];  
  
    // Display only electric vehicles  
    display_electric_vehicles(&vehicles);  
}
```

Explanation

Enum Definition:

- The FuelType enum defines three fuel types: Petrol, Diesel, and Electric.

Struct Definition:

- The Vehicle struct contains:
 - brand (String) – Vehicle manufacturer.
 - model (String) – Vehicle model name.
 - fuel_type (FuelType) – Uses the FuelType enum.

Function - display_electric_vehicles:

- Uses .filter() with matches! to filter only electric vehicles.
- Iterates through the filtered results and prints the **brand** and **model**.

main Function:

- Creates a Vec<Vehicle> with sample data.
- Calls display_electric_vehicles() to display electric vehicles.

Output:

```
File Edit View Search Terminal Help
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ gedit vechile.rs
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ rustc vechile.rs
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ ./vechile
Electric Vehicles:
Brand: Tesla, Model: Model S
Brand: Nissan, Model: Leaf
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ █
```

Conclusion

This implementation effectively demonstrates the integration of **Structs** and **Enums** in Rust. The use of .filter() with matches! ensures efficient and concise filtering logic.

The program follows clear structure and best practices, ensuring readability and maintainability.