Department of Cyber Security

Amrita School of Computing

Amrita Vishwa Vidyapeetham, Chennai Campus

Principals of Programming Languages

Subject Code: 20CYS312	Date:2025/2/21
Name: Sushant Yadav	Roll Number:CH.EN.U4CYS22067

LAB:7 Programming with RUST

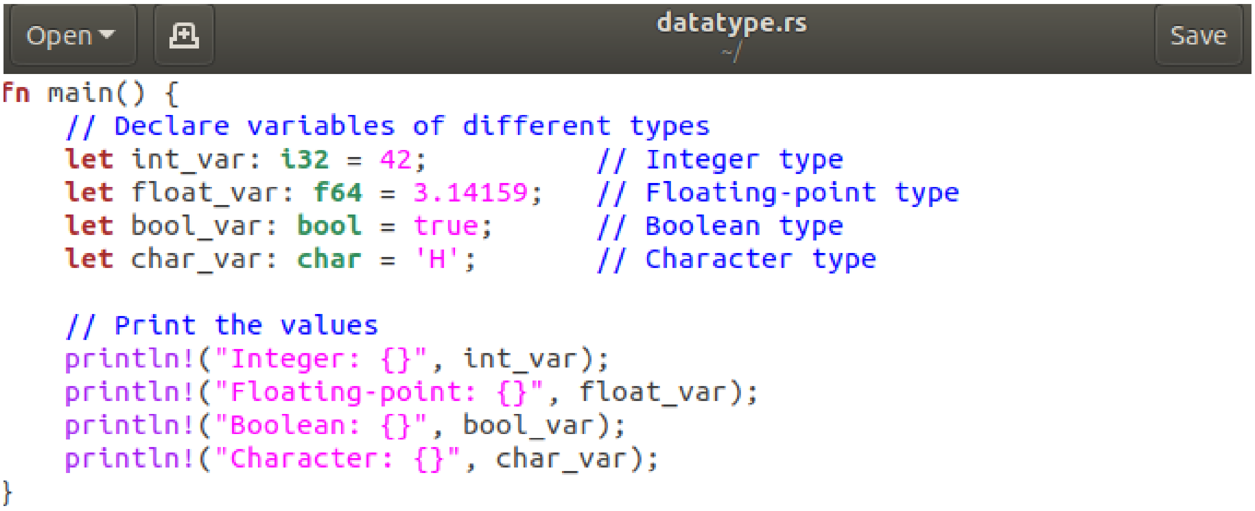
Task 1: Data Types and Variables

1. Declare variables of the following types: integer, floating-point, boolean, and character. Print the value of each variable.

Objective: Understand different **data types** and how to declare variables in Rust.

- Learn integer (i32), floating-point (f64), boolean (bool), and character (char) types.
- Practice using println! for output.

Code:



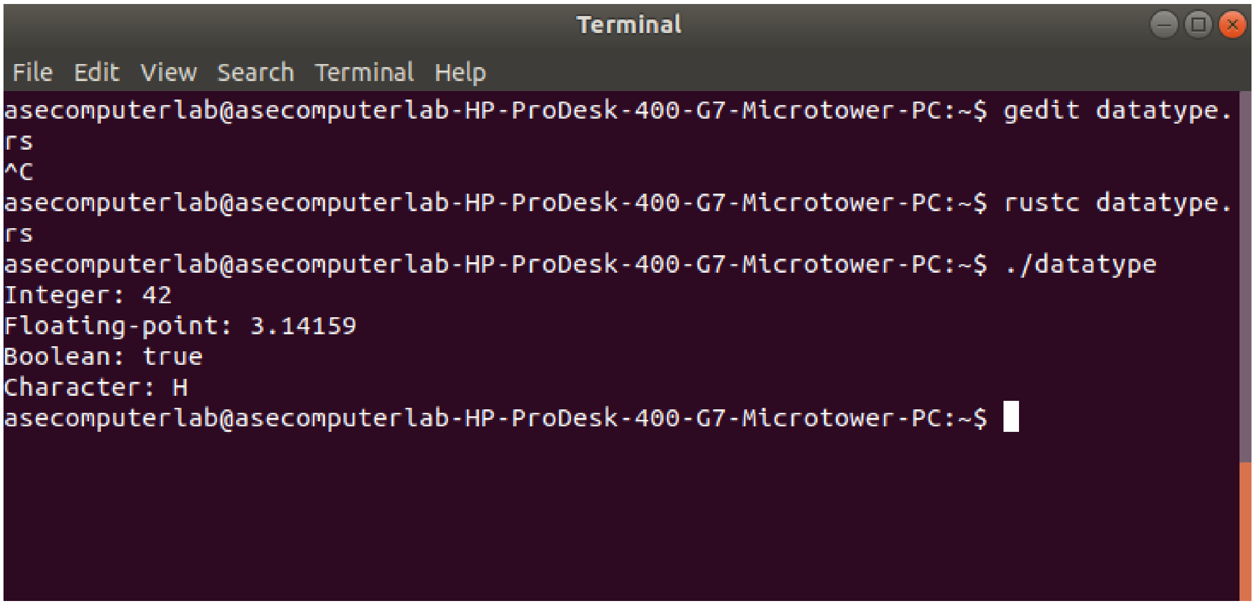
```rust
fn main() {
    // Declare variables of different types
    let int_var: i32 = 42;          // Integer type
    let float_var: f64 = 3.14159;   // Floating-point type
    let bool_var: bool = true;      // Boolean type
    let char_var: char = 'H';       // Character type

    // Print the values
    println!("Integer: {}", int_var);
    println!("Floating-point: {}", float_var);
    println!("Boolean: {}", bool_var);
    println!("Character: {}", char_var);
}
```

Explanation:

- **Integer (i32)**: Rust has several integer types, with i32 being the default for 32-bit signed integers. let int_var: i32 = 42; defines an integer variable.

- **Floating-point (f64)**: Rust has two floating-point types, f32 and f64, where f64 is the default. let float_var: f64 = 3.14159; defines a floating-point variable.
- **Boolean (bool)**: A boolean type can hold true or false. let bool_var: bool = true; declares a boolean variable.
- **Character (char)**: A character type in Rust can store a single Unicode character. let char_var: char = 'H'; defines a character variable.

output:



```
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ gedit datatype.
rs
^C
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ rustc datatype.
rs
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ ./datatype
Integer: 42
Floating-point: 3.14159
Boolean: true
Character: H
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ 
```

Conclusion:

- **Rust** is a statically typed language, meaning each variable must have a defined type. You can explicitly annotate the type (e.g., i32, f64, bool, char), or Rust can infer the type based on the assigned value.
- The println! macro is used to output values to the console in Rust.

Task 2: Simple Arithmetic Operations

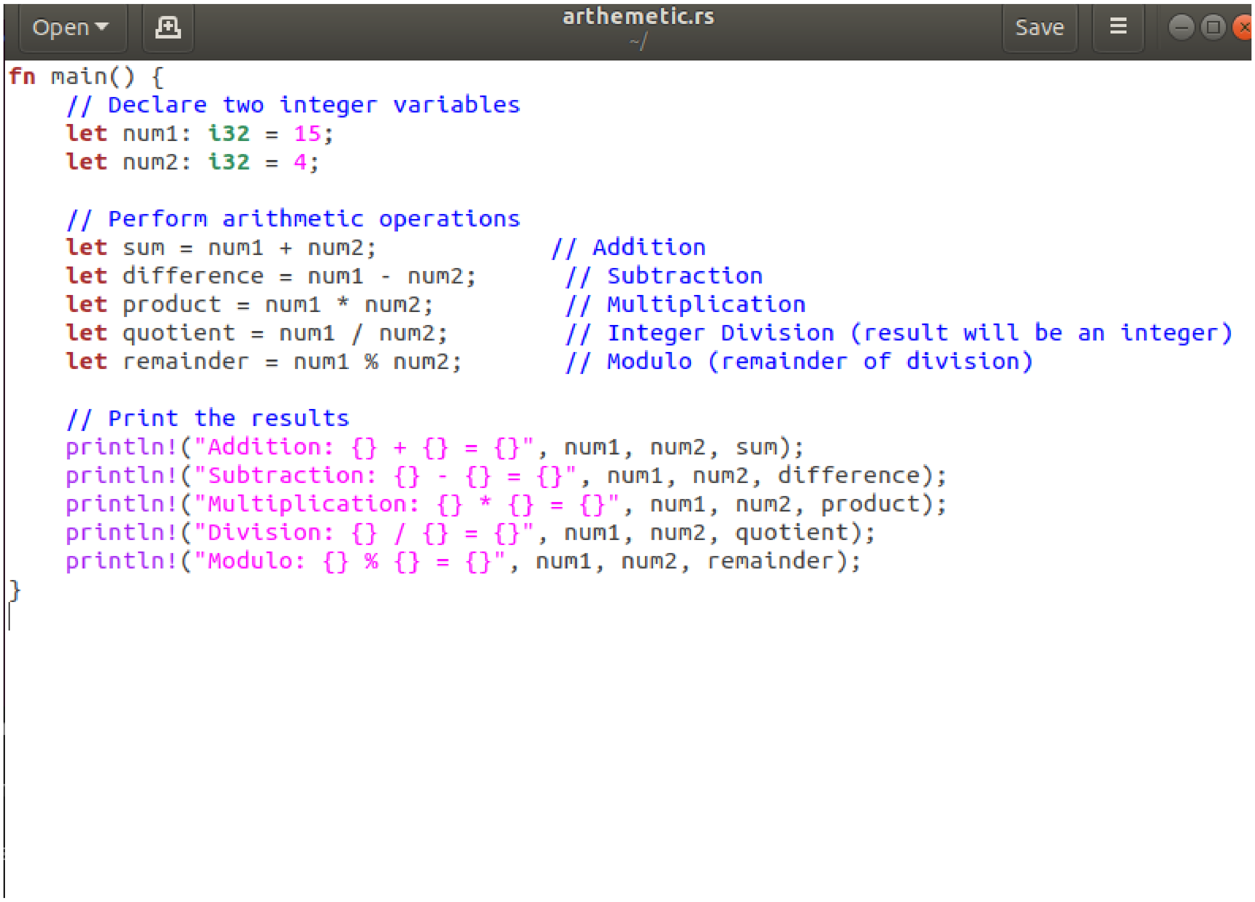
1. Declare two integer variables and perform the following operations:
   a. Addition
   b. Subtraction
   c. Multiplication

   d. Division
   e. Modulo
2. Print the result of each operation.

Objective: Perform **basic arithmetic operations** using integers.

- Learn how to use +, -, *, /, and % operators.
- Understand integer division (/ discards decimals).

Code:



```
fn main() {
    // Declare two integer variables
    let num1: i32 = 15;
    let num2: i32 = 4;

    // Perform arithmetic operations
    let sum = num1 + num2;            // Addition
    let difference = num1 - num2;      // Subtraction
    let product = num1 * num2;         // Multiplication
    let quotient = num1 / num2;        // Integer Division (result will be an integer)
    let remainder = num1 % num2;       // Modulo (remainder of division)

    // Print the results
    println!("Addition: {} + {} = {}", num1, num2, sum);
    println!("Subtraction: {} - {} = {}", num1, num2, difference);
    println!("Multiplication: {} * {} = {}", num1, num2, product);
    println!("Division: {} / {} = {}", num1, num2, quotient);
    println!("Modulo: {} % {} = {}", num1, num2, remainder);
}
```

Explanation:

1. **Variable Declaration:**
   a. num1 and num2 are declared as i32 (32-bit signed integers).
2. **Arithmetic Operations:**
   a. + for addition
   b. - for subtraction

c. * for multiplication

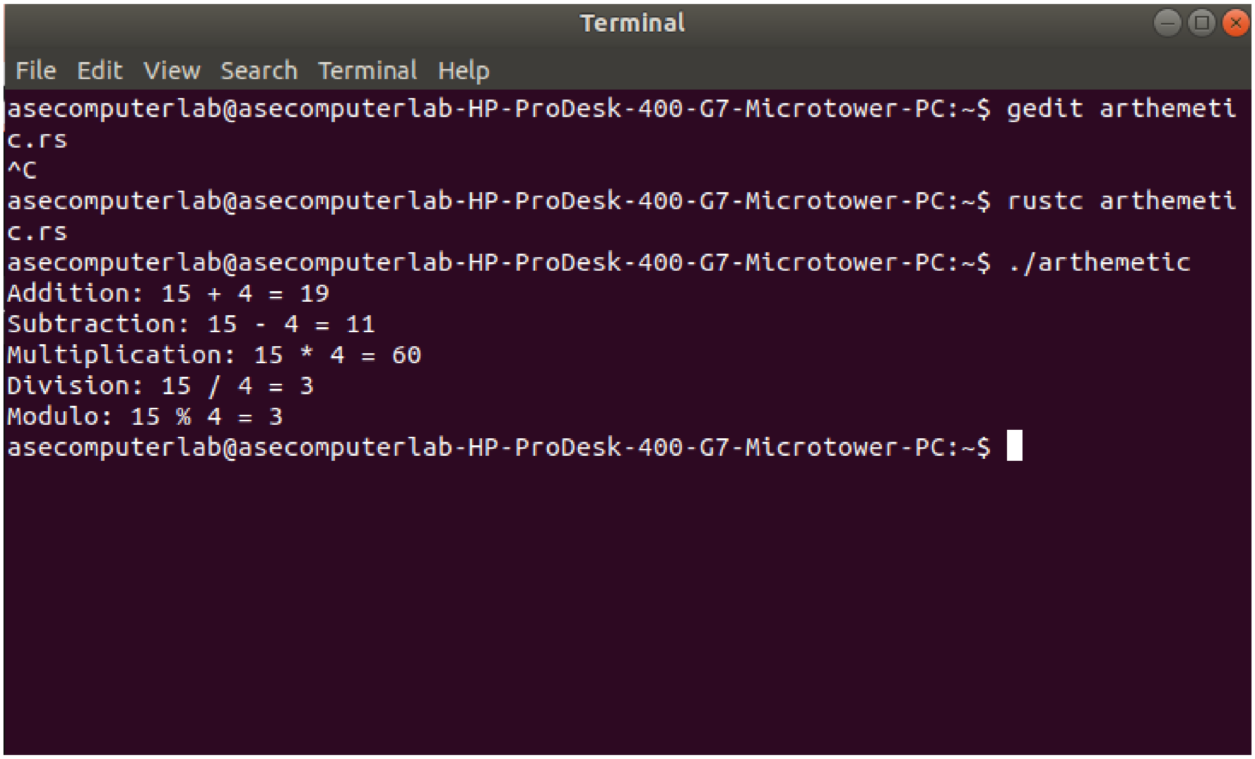
d. / for integer division (since num1 and num2 are integers, the result is also an integer)

e. % for modulo operation (gives the remainder of division)

3. **Printing the Results:**

   a. println! is used to format and display the output.

Output:



```
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ gedit arthemeti
c.rs
^C
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ rustc arthemeti
c.rs
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ ./arthemetic
Addition: 15 + 4 = 19
Subtraction: 15 - 4 = 11
Multiplication: 15 * 4 = 60
Division: 15 / 4 = 3
Modulo: 15 % 4 = 3
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ 
```

Conclusion:

- Rust supports basic arithmetic operations just like other programming languages.
- **Integer division** only returns the integer part of the quotient.
- **Modulo (%)** is useful for finding the remainder when dividing two numbers.

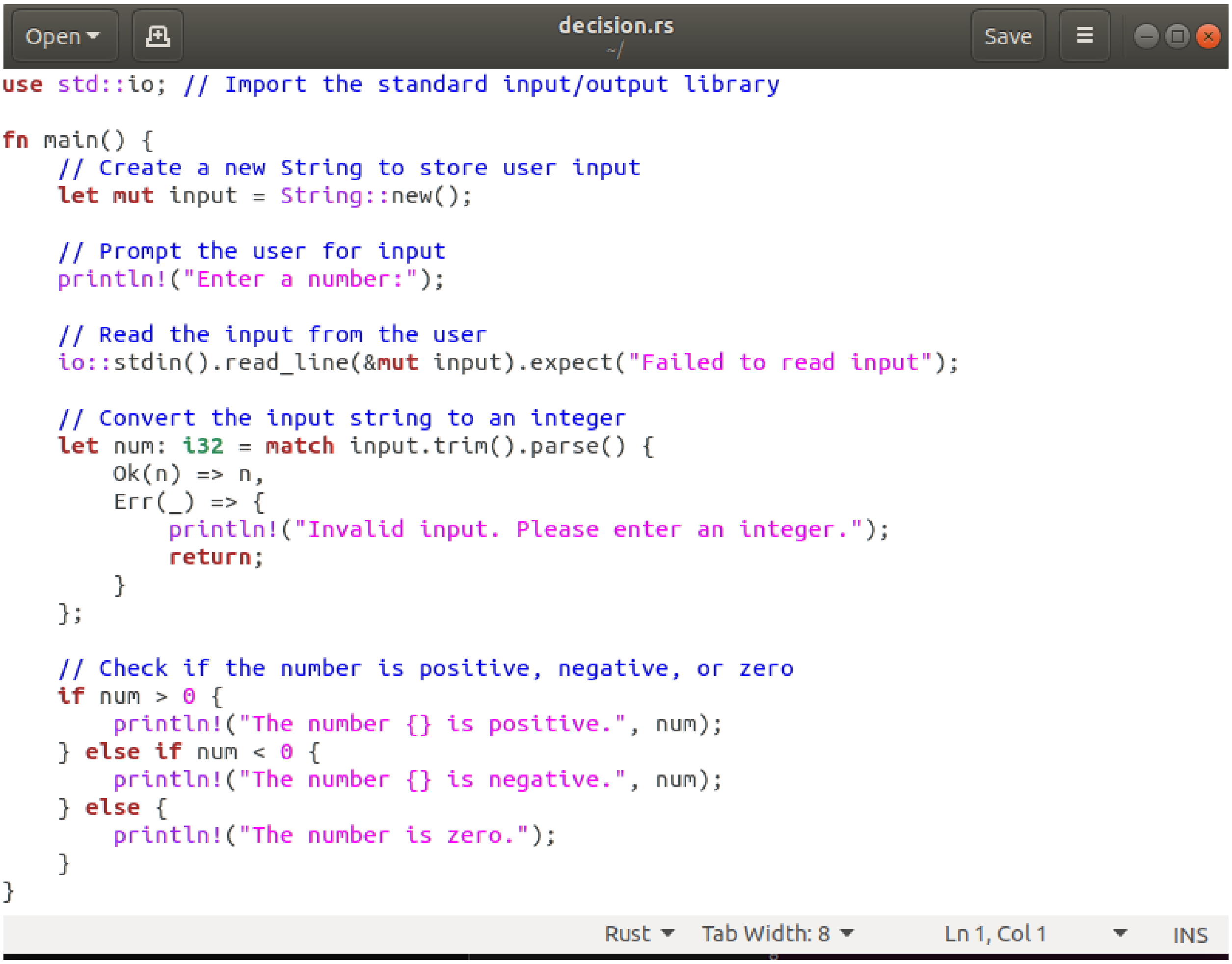
Task 3: If-Else Decision Making

1. Write a program that:
    a. Takes a number as input.
    b. Checks whether the number is positive, negative, or zero using an if-else statement.
    c. Print a message based on the result.

Objective: Learn **conditional statements** (if-else).

- Practice taking **user input** and handling errors.
- Understand decision-making by checking if a number is **positive, negative, or zero.**

Code:



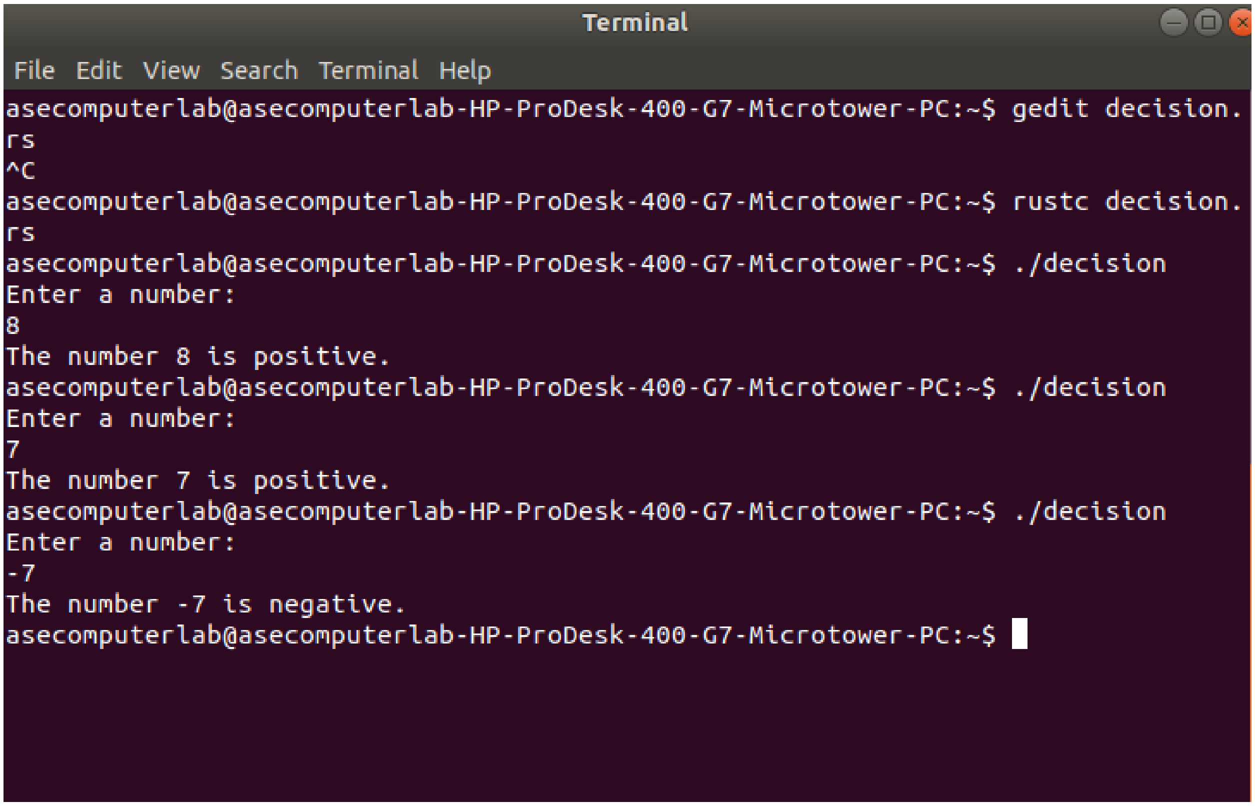
```rust
use std::io; // Import the standard input/output library

fn main() {
    // Create a new String to store user input
    let mut input = String::new();

    // Prompt the user for input
    println!("Enter a number:");

    // Read the input from the user
    io::stdin().read_line(&mut input).expect("Failed to read input");

    // Convert the input string to an integer
    let num: i32 = match input.trim().parse() {
        Ok(n) => n,
        Err(_) => {
            println!("Invalid input. Please enter an integer.");
            return;
        }
    };

    // Check if the number is positive, negative, or zero
    if num > 0 {
        println!("The number {} is positive.", num);
    } else if num < 0 {
        println!("The number {} is negative.", num);
    } else {
        println!("The number is zero.");
    }
}
```

Explanation:

1. **User Input Handling:**
    a. use std::io; is used to import the input/output library.
    b. let mut input = String::new(); creates a mutable string to store user input.
    c. io::stdin().read_line(&mut input).expect("Failed to read input"); reads the user input.
2. **Parsing the Input:**
    a. .trim().parse() converts the input string into an integer (i32).
    b. match is used to handle parsing errors gracefully. If the user enters a non-integer, it prints an error message and exits the program.
3. **If-Else Condition:**
    a. if num > 0 → Prints **"positive"** if the number is greater than zero.

  b. else if num < 0 → Prints **"negative"** if the number is less than zero.
  c. else → Prints **"zero"** if the number is exactly zero.

Output:



```
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ gedit decision.
rs
^C
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ rustc decision.
rs
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ ./decision
Enter a number:
8
The number 8 is positive.
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ ./decision
Enter a number:
7
The number 7 is positive.
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ ./decision
Enter a number:
-7
The number -7 is negative.
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ 
```

Conclusion:

- This program successfully demonstrates **decision-making** using an **if-else** structure in Rust.
- It includes **user input handling**, **error checking**, and **conditional statements** to classify the number.
- Rust's **strong type safety** ensures that invalid inputs are properly handled.

Task 4: Checking for Even or Odd

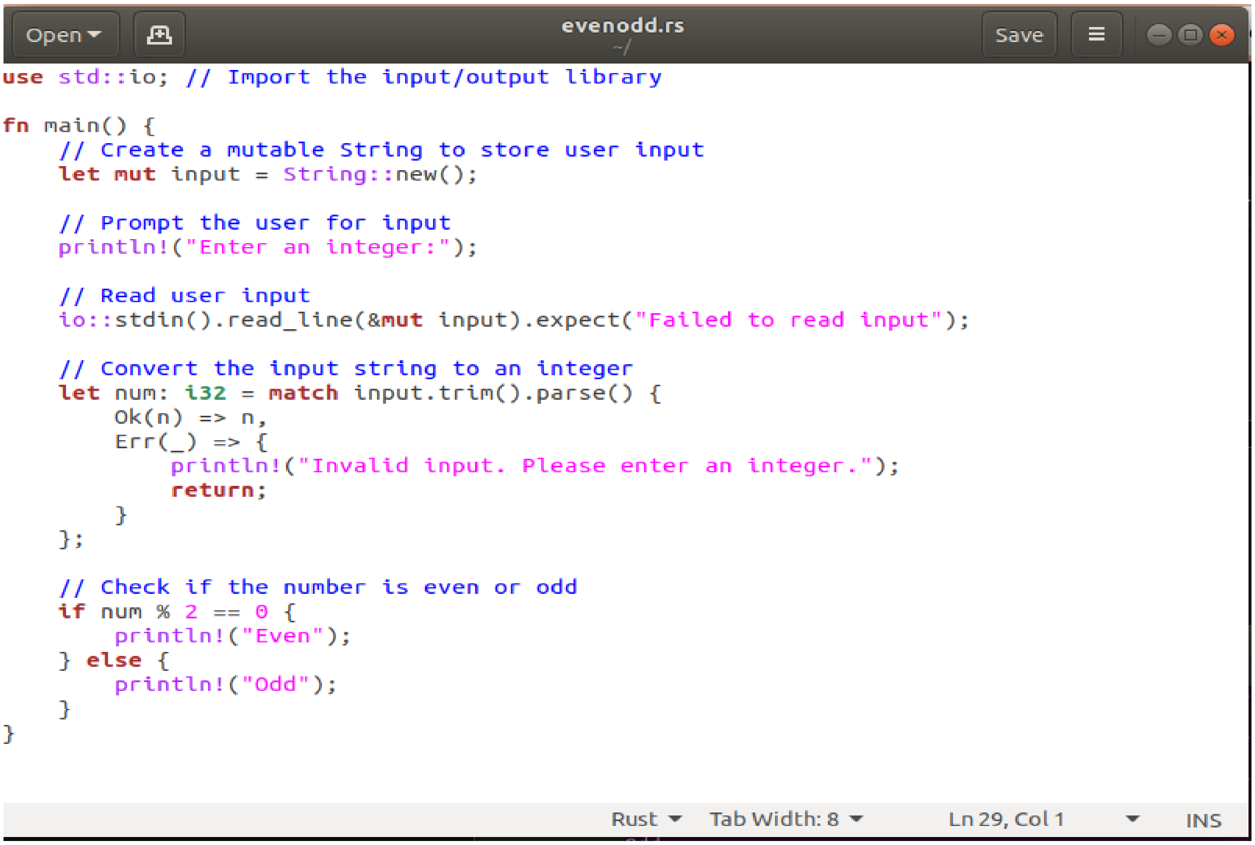
1. Write a program that:
   a. Takes an integer as input.
   b. Uses an if-else statement to check if the number is even or odd.

c. Print "Even" if the number is even and "Odd" if the number is odd.

Objective: Apply **if-else conditions** with the **modulo operator (%)**.

- Learn how to check for **even or odd** numbers.
- Understand how the % 2 operation works in Rust.

Code:



```rust
use std::io; // Import the input/output library

fn main() {
    // Create a mutable String to store user input
    let mut input = String::new();

    // Prompt the user for input
    println!("Enter an integer:");

    // Read user input
    io::stdin().read_line(&mut input).expect("Failed to read input");

    // Convert the input string to an integer
    let num: i32 = match input.trim().parse() {
        Ok(n) => n,
        Err(_) => {
            println!("Invalid input. Please enter an integer.");
            return;
        }
    };

    // Check if the number is even or odd
    if num % 2 == 0 {
        println!("Even");
    } else {
        println!("Odd");
    }
}
```

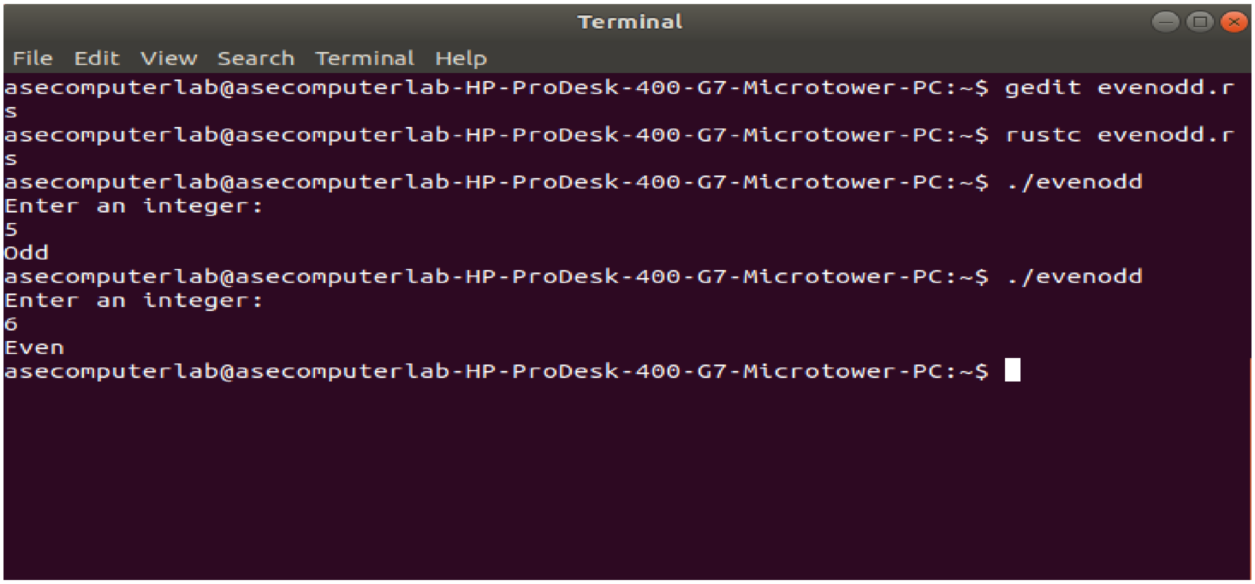
Explanation:

1. **User Input Handling:**
    a. use std::io; imports Rust's standard I/O library.
    b. let mut input = String::new(); creates a mutable string to store user input.

   c. io::stdin().read_line(&mut input).expect("Failed to read input"); reads the input from the user.

2. **Parsing Input to Integer:**
   a. .trim().parse() converts the input string to an i32 integer.
   b. The match statement handles errors if the user enters a non-integer.
3. **Checking Even or Odd:**
   a. **Even number:** If num % 2 == 0, the number is divisible by 2, so it's **even**.
   b. **Odd number:** Otherwise, it's **odd**.

Output:



```
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ gedit evenodd.r
s
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ rustc evenodd.r
s
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ ./evenodd
Enter an integer:
5
Odd
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ ./evenodd
Enter an integer:
6
Even
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ 
```

Conclusion:

- The program successfully checks whether a number is **even or odd** using an **if-else statement**.
- It includes **user input handling**, **error checking**, and **modulo operator (%)** for checking even/odd numbers.
- Rust’s strong type system ensures that only valid integers are processed.

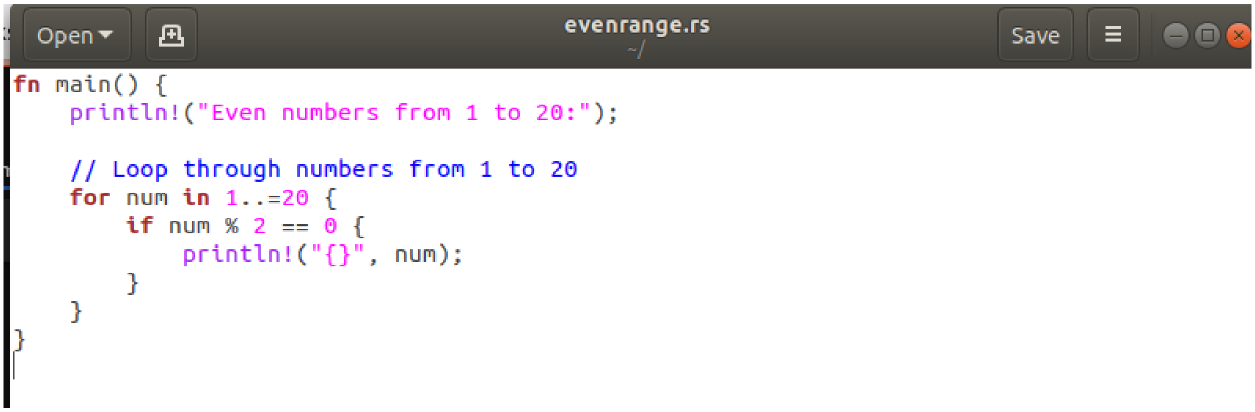
Task 5: Using a Loop to Print Numbers

2. Write a program that uses a for loop to print the even numbers from the range 1 to 20.

Objective: Understand **for loops** and iteration over a **range**.

- Learn to loop through numbers using for num in 1..=20.
- Use **conditional statements** to filter **even numbers**.

Code:



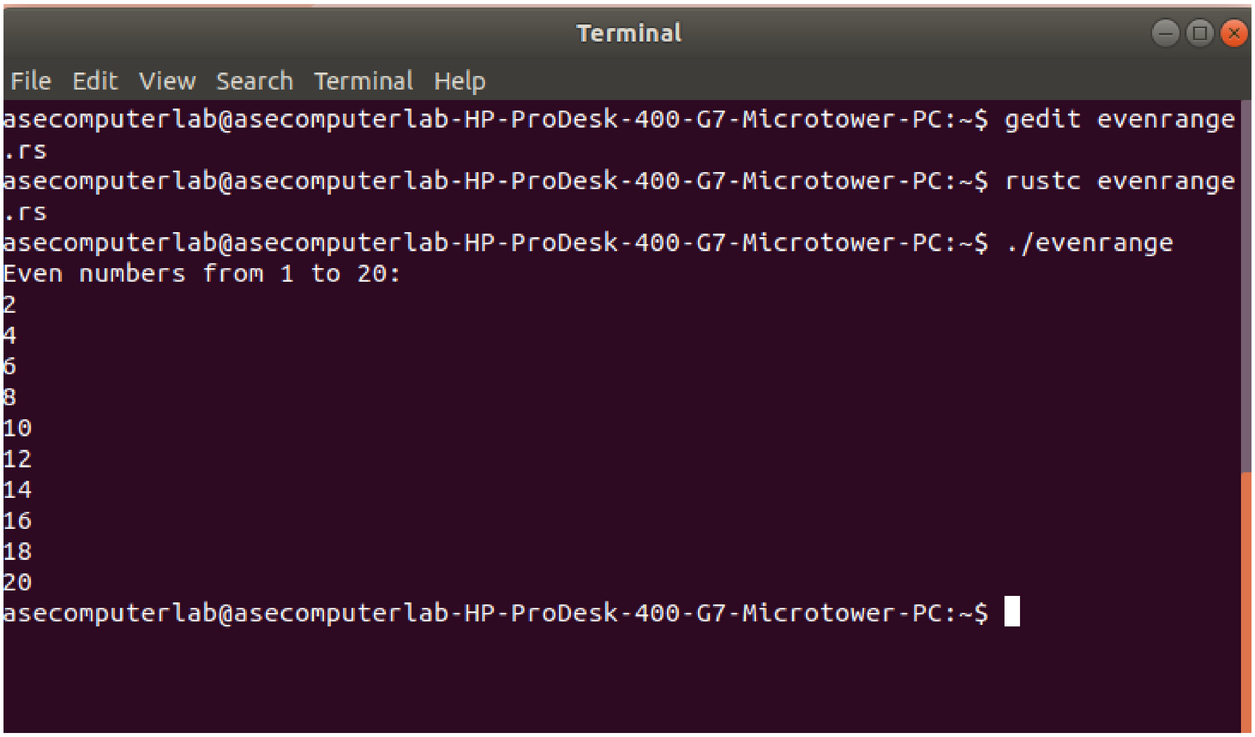
```
fn main() {
    println!("Even numbers from 1 to 20:");

    // Loop through numbers from 1 to 20
    for num in 1..=20 {
        if num % 2 == 0 {
            println!("{}", num);
        }
    }
}
```

Explanation:

1. **Looping Through the Range**:
   a. The for num in 1..=20 loop iterates through numbers **from 1 to 20**.
   b. The ..= operator ensures **inclusive range** (includes 20).
2. **Checking Even Numbers**:
   a. Inside the loop, if num % 2 == 0 checks if the number is divisible by 2 (even).

b. If true, the number is printed.

Output:



```
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ gedit evenrange
.rs
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ rustc evenrange
.rs
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ ./evenrange
Even numbers from 1 to 20:
2
4
6
8
10
12
14
16
18
20
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ 
```

Conclusion:

- **For loops** in Rust are powerful and can iterate over ranges.
- The **modulo operator (%)** helps filter even numbers.
- The **step_by(2) method** offers an optimized way to iterate over even numbers directly.

Task 6: While Loop Example

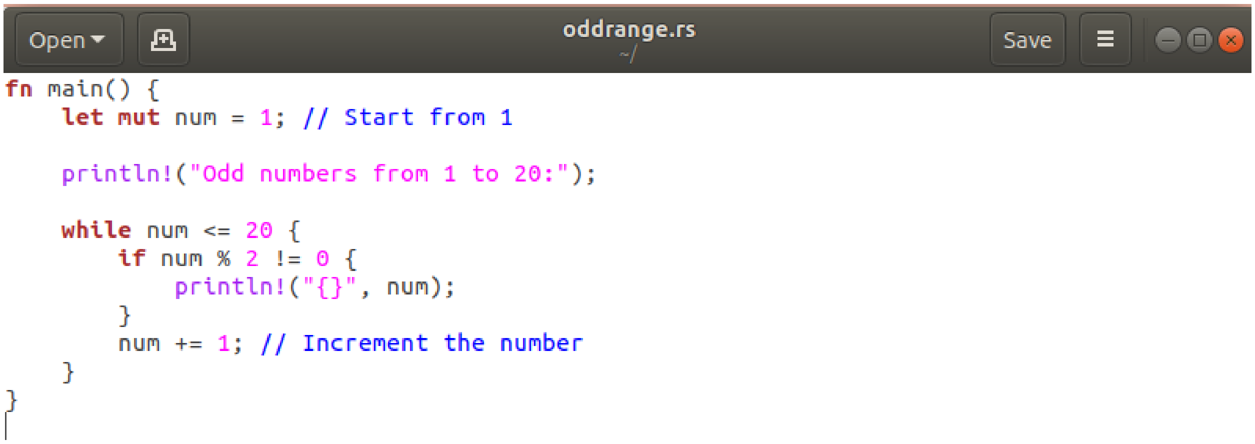
3. Write a program that uses a while loop to print odd numbers from the range 1 to 20.

Objective: Learn how to use **while loops** for iteration.

- Practice **loop control** using while condition.

- Understand **incrementing values** (num += 1 vs. num += 2 for efficiency).

Code:

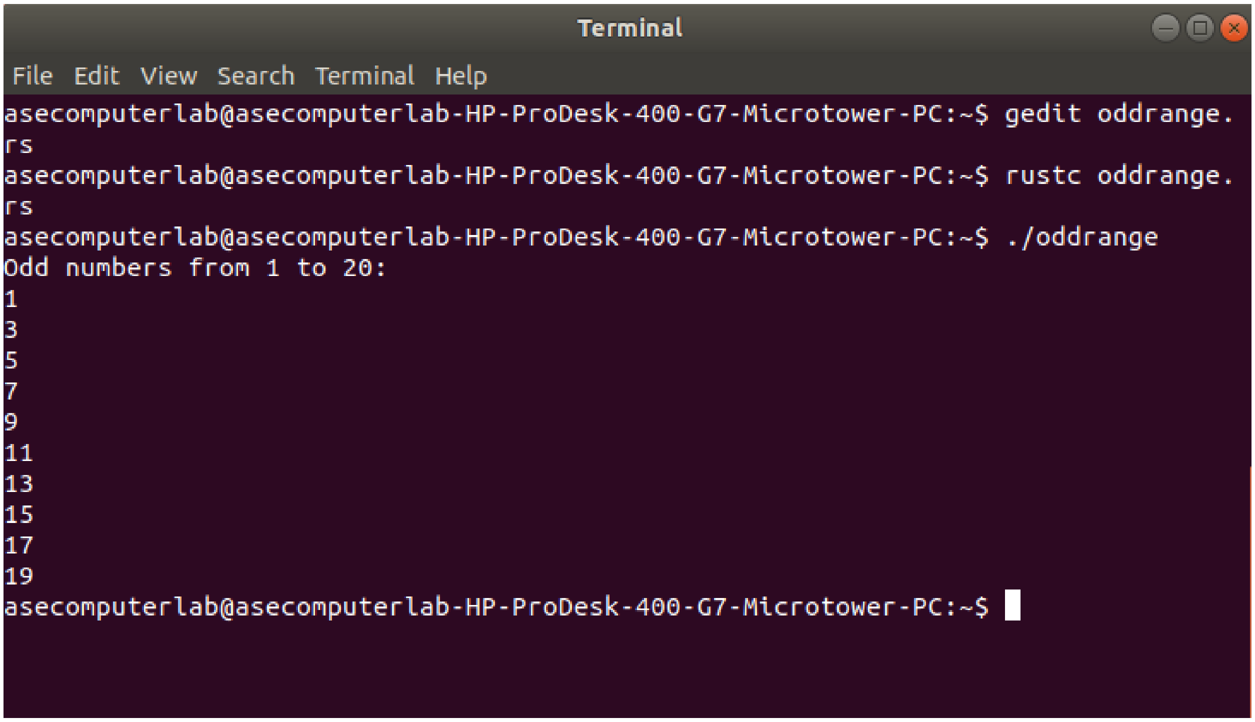


```
fn main() {
    let mut num = 1; // Start from 1

    println!("Odd numbers from 1 to 20:");

    while num <= 20 {
        if num % 2 != 0 {
            println!("{}", num);
        }
        num += 1; // Increment the number
    }
}
```

Explanation:

1. **Initialize the Counter**:
    a. let mut num = 1; starts at **1** (first odd number).
    b. mut keyword allows modification of num inside the loop.
2. **Using a While Loop**:
    a. while num <= 20 ensures the loop runs while num is **less than or equal to 20**.
    b. Inside the loop, if num % 2 != 0 checks if the number is **odd**.
    c. If true, the number is printed.
3. **Incrementing the Counter**:
    a. num += 1; ensures the loop moves to the next number.

Output:



```
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ gedit oddrange.
rs
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ rustc oddrange.
rs
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ ./oddrange
Odd numbers from 1 to 20:
1
3
5
7
9
11
13
15
17
19
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ 
```

Conclusion:

- While loops are useful when the number of iterations is not fixed.
- Modulo (% 2 != 0) helps identify odd numbers.
- Directly incrementing by 2 is a more efficient way to iterate over odd numbers.

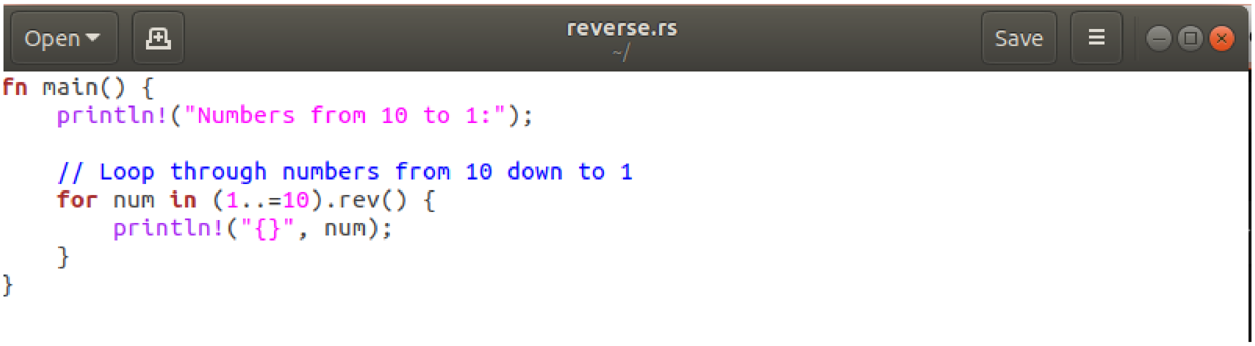
Task 7: Using a For Loop with a Range

4. Write a program that uses a for loop to print the numbers from 10 to 1 in reverse order (10, 9, 8, ..., 1).

Objective: Learn how to iterate in reverse order.

- Use the .rev() method to reverse a range.
- Understand counting down in a for loop.

Code:

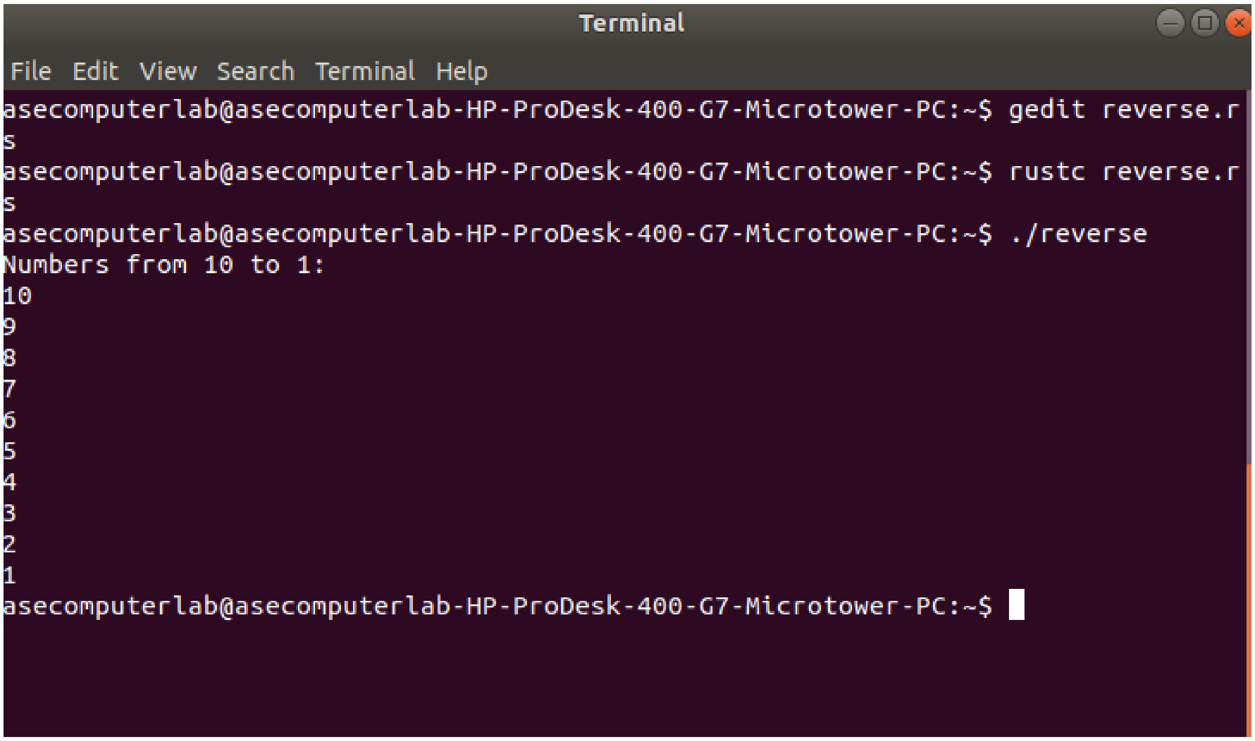


```rust
fn main() {
    println!("Numbers from 10 to 1:");

    // Loop through numbers from 10 down to 1
    for num in (1..=10).rev() {
        println!("{}", num);
    }
}
```

Explanation:

1. **Using a Reverse Range**:
    a. (1..=10).rev() creates a range from 1 to 10 and then **reverses** it.
    b. The .rev() method efficiently reverses the sequence.
2. **For Loop Iteration**:
    a. for num in (1..=10).rev() iterates **backwards** from 10 to 1.
    b. Each value is printed.

Output:



```
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ gedit reverse.r
s
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ rustc reverse.r
s
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ ./reverse
Numbers from 10 to 1:
10
9
8
7
6
5
4
3
2
1
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ 
```

Conclusion:

- **For loops** with .rev() are a simple way to iterate in **reverse order**.
- Rust provides **efficient iteration methods** like .rev() to handle reverse sequences.
- This program efficiently prints numbers **from 10 down to 1**.