VARDHAMAN COLLEGE OF ENGINEERING

(AUTONOMOUS)

B. Tech. CSE III Year II Sem **VCE-R22**

RUST Programming

Course Code: A8525

L	T	P	C
0	0	2	1

SYLLABUS

LIST OF EXPERIMENTS

1. Write a RUST program to display “hello world” message.
2. Write a RUST program to demonstrate variables, mutability and type references.
3. Write a program to demonstrate Rust Type Casting
4. Write a RUST program to perform basic arithmetic operations on two given numbers using arithmetic operators.
5. Write a RUST program to calculate student grades to a subject based on their overall score.
    a. if the score is above 90, assign grade A
    b. if the score is above 75, assign grade B
    c. if the score is above 65, assign grade C
6. Write a RUST program to print all prime numbers from 1 to n using for loop.
7. Write a program to demonstrate the usage of functions in RUST to find sum of two numbers. Pass two values as parameters.
8. Write a program to create a vector of strings and access the elements. Display all elements in sorted order.
9. Write a RUST program to demonstrate different ways of creating iterators.
10. Write a RUST program to perform all strings operations like creation of string, slicing of stirng etc.
11. Write a RUST program to demonstrate recoverable errors using panic, except.
12. Write a RUST program to demonstrate about result enum and option enum.
13. Write a RUST program to demonstrate data movement and ownership rules in Rust
14. Write a RUST program to demonstrate ownership in functions.
15. Demonstrate Building and Running Project with Cargo in Rust
16. Write a program to demonstrate Defining, Implementing and Using a Trait in Rust
17. Write a RUST program to demonstrate about pattern matching.
18. Implement Generic struct and Generic Functions in Rust.
19. Write a RUST program for performing following FILE operations:
    a) Opening a file
    b) Reading from a file
    c) Writing to a file
    d) Removing a file
    e) Appending to a file
20. Build a multithreaded web server using RUST.

VARDHAMAN COLLEGE OF ENGINEERING

(AUTONOMOUS)

Affiliated to **JNTUH**, Approved by **AICTE**, Accredited by **NAAC** with **A++** Grade, **ISO 9001:2015** Certified
Kacharam, Shamshabad, Hyderabad – 501218, Telangana, India

III B. TECH II SEMESTER

RUST Programming

Course Plan

Course Outcomes (COs)

After the completion of the course, the learner will be able to:

CO#	CO Statement	BL#
CO1	Create a Rust project and write basic rust programs, including proper Cargo configuration.	BL3
CO2	Translate a design into a working Rust program.	BL3
CO3	Use structs, enums ,files and traits as intended in the construction of Rust programs.	BL3
CO4	Handle different types of errors and use generic types, traits	BL3

Course Plan

Session #	Lecture/ Practice #	Topic	BL#	CO#
1	P1	Write a RUST program to display “hello world” message.	BL3	CO1
2	P2	Write a RUST program to demonstrate variables, mutability and type references.	BL3	CO1
3	P3	Write a program to demonstrate Rust Type Casting	BL3	CO1
4	P4	Write a RUST program to perform basic arithmetic operations on two given numbers using arithmetic operators.	BL3	CO1
5	P5	Write a RUST program to calculate student grades to a subject based on their overall score. a. if the score is above 90, assign grade A b. if the score is above 75, assign grade B c. if the score is above 65, assign grade C	BL3	CO2
6	P6	Write a RUST program to print all prime numbers from 1 to n using for loop.	BL3	CO2
7	P7	Write a program to demonstrate the usage of functions in RUST to find sum of two numbers. Pass two values as parameters.	BL3	CO2
8	P8	Write a program to create a vector of strings and access the elements. Display all elements in sorted order.	BL3	CO2
9	P9	Write a RUST program to demonstrate different ways of creating iterators.	BL3	CO2
10	P10	Write a RUST program to perform all strings operations like creation of string, slicing of stirng etc.	BL3	CO2
11	P11	Write a RUST program to demonstrate recoverable errors using panic, except.	BL3	CO4
12	P12	Write a RUST program to demonstrate about result enum and option enum.	BL3	CO3

Session #	Lecture/ Practice #	Topic	BL#	CO#
13	P13	Write a RUST program to demonstrate data movement and ownership rules in Rust.	BL3	CO3
14	P14	Write a RUST program to demonstrate ownership in functions.	BL3	CO4
15	P15	Demonstrate Building and Running Project with Cargo in Rust	BL3	CO1
16	P16	Write a program to demonstrate Defining, Implementing and Using a Trait in Rust.	BL3	CO4
17	P17	Write a RUST program to demonstrate about pattern matching.	BL3	CO2
18	P18	Implement Generic struct and Generic Functions in Rust.	BL3	CO3
19	P19	write a RUST program for performing following FILE operations: a) Opening a file b) Reading from a file c) Writing to a file d) Removing a file e) Appending to a file	BL3	CO3
20	P20	Build a multithreaded web server using RUST.	BL3	CO3

Week-1	• **Write a RUST program to display “hello world” message.** • **Write a RUST program to demonstrate variables, mutability and type references.**

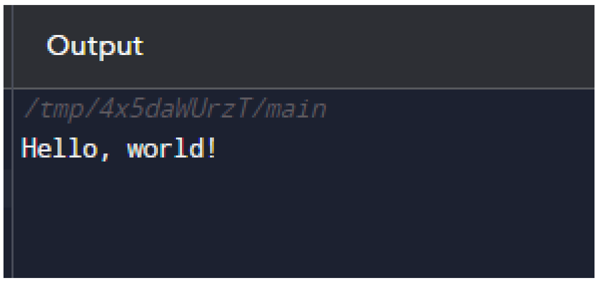
- **Write a RUST program to display “hello world” message.**

```
fn main() {
   println!("Hello, world!");
}
```

To run this program, you can follow these steps:

Install Rust from Rust's official website.
Create a new file named main.rs and paste the above code into it.
Open a terminal or command prompt and navigate to the directory where main.rs is located.
Run the following command to compile and execute the program:

```
$ rustc main.rs
$ ./main
```



```
Output
/tmp/4x5daWUrzT/main
Hello, world!
```

- **Write a RUST program to demonstrate variables, mutability and type references.**

```
fn main() {
   // Declaring a variable
   let x: i32 = 5; // Immutable variable
   // Printing the value of x
   println!("Value of x: {}", x);
   // Trying to change the value of x will result in a compilation error
   // x = 10; // Uncommenting this line will result in an error because x is
immutable
   // Declaring a mutable variable
   let mut y: i32 = 10; // Mutable variable

   // Printing the value of y
   println!("Value of y (before mutation): {}", y);
   // Mutating the value of y
   y = 15;
```

```
    // Printing the mutated value of y
    println!("Value of y (after mutation): {}", y);

    // Demonstrating type references
    let z: &i32 = &y; // Creating a reference to y

    // Printing the value of y and its reference
    println!("Value of y: {}", y);
    println!("Reference to y (z): {}", z);

    // Trying to mutate the value through the reference will result in a
compilation error
    // *z = 20; // Uncommenting this line will result in an error because z is an
immutable reference

    // Creating a mutable reference to y
    let z_mut: &mut i32 = &mut y;

    // Printing the value of y and its mutable reference
    println!("Value of y: {}", y);
    println!("Mutable reference to y (z_mut): {}", z_mut);

    // Mutating the value of y through the mutable reference
    *z_mut = 20;

    // Printing the mutated value of y
    println!("Value of y (after mutation through mutable reference): {}", y);
}
```

Certainly! Here's a Rust program that demonstrates variables, mutability, and type references:

This program demonstrates:

- Declaration of immutable and mutable variables (`x` and `y`).
- Attempting to mutate an immutable variable results in a compilation error.
- Demonstration of type references (`z`).
- Attempting to mutate the value through an immutable reference results in a compilation error.
- Declaration and usage of mutable references (`z_mut`).
- Mutating the value through a mutable reference.

Week-2	• **Write a program to demonstrate Rust Type Casting Write a RUST program to perform basic arithmetic operations on two given numbers using arithmetic operators.**

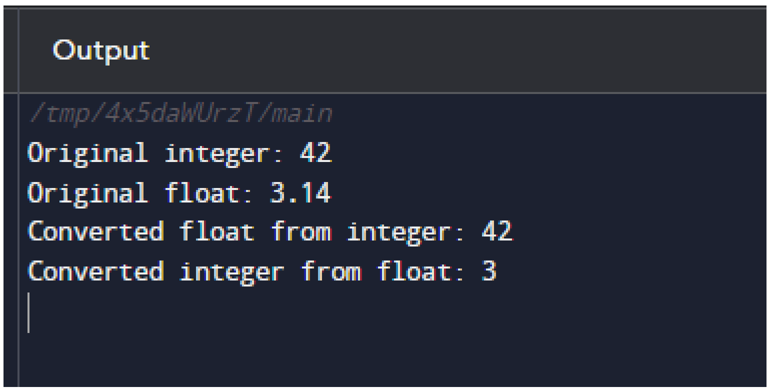
- **Write a program to demonstrate Rust Type Casting.**

```
fn main() {
   // Define variables of different types
   let integer_number: i32 = 42;
   let float_number: f64 = 3.14;

   // Type casting: From integer to float
   let float_from_int: f64 = integer_number as f64;

   // Type casting: From float to integer
   let int_from_float: i32 = float_number as i32;

   println!("Original integer: {}", integer_number);
   println!("Original float: {}", float_number);
   println!("Converted float from integer: {}", float_from_int);
   println!("Converted integer from float: {}", int_from_float);
}
```

In this program:

- We define an integer variable `integer_number` of type `i32` and a float variable `float_number` of type `f64`.
- We perform type casting using the `as` keyword. We cast `integer_number` to `f64` and `float_number` to `i32`.
- We then print out the original values and the values after type casting.

When you run this program, you'll see the original values along with the values obtained after type casting.



```
/tmp/4x5daWUrzT/main
Original integer: 42
Original float: 3.14
Converted float from integer: 42
Converted integer from float: 3
```

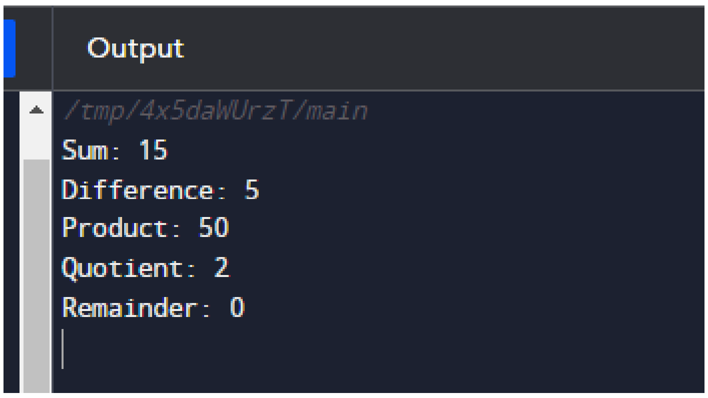
Write a RUST program to perform basic arithmetic operations on two given numbers using arithmetic operators.

fn main() {

~~~
```rust
fn main() {
    // Define two numbers
    let num1: i32 = 10;
    let num2: i32 = 5;

    // Addition
    let sum = num1 + num2;
    println!("Sum: {}", sum);

    // Subtraction
    let difference = num1 - num2;
    println!("Difference: {}", difference);

    // Multiplication
    let product = num1 * num2;
    println!("Product: {}", product);

    // Division
    let quotient = num1 / num2;
    println!("Quotient: {}", quotient);

    // Remainder (Modulus)
    let remainder = num1 % num2;
    println!("Remainder: {}", remainder);
}
```
~~~

In this program:
- We define two variables `num1` and `num2` of type `i32`.
- We perform addition, subtraction, multiplication, division, and modulus operations using the `+`, `-`, `*`, `/`, and `%` operators respectively.
- We print out the results of each operation.

When you run this program, you'll see the results of each arithmetic operation based on the given numbers.



```
/tmp/4x5daWUrzT/main
Sum: 15
Difference: 5
Product: 50
Quotient: 2
Remainder: 0
```

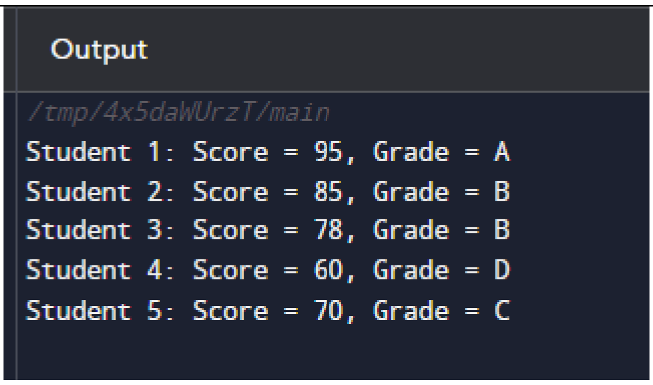
Week-3	• **Write a RUST program to calculate student grades to a subject based on their overall score.** **a. if the score is above 90, assign grade A** **b. if the score is above 75, assign grade B** **c. if the score is above 65, assign grade C** • **Write a RUST program to print all prime numbers from 1 to n using for loop.**

- **Write a RUST program to calculate student grades to a subject based on their overall score.**
  a. if the score is above 90, assign grade A
  b. if the score is above 75, assign grade B
  c. if the score is above 65, assign grade C

```rust
fn calculate_grade(score: u32) -> char {
    if score > 90 {
        'A'
    } else if score > 75 {
        'B'
    } else if score > 65 {
        'C'
    } else {
        'D'
    }
}

fn main() {
    let scores = vec![95, 85, 78, 60, 70];

    for (i, &score) in scores.iter().enumerate() {
        let grade = calculate_grade(score);
        println!("Student {}: Score = {}, Grade = {}", i + 1, score, grade);
    }
}
```

In this program:
- We define a function `calculate_grade` that takes a student's score as input and returns the corresponding grade based on the given criteria.
- Inside the `main` function, we have a vector `scores` containing the scores of different students.
- We iterate over each score using a `for` loop and calculate the grade for each score using the `calculate_grade` function.
- We then print out the student number, their score, and the grade they received.

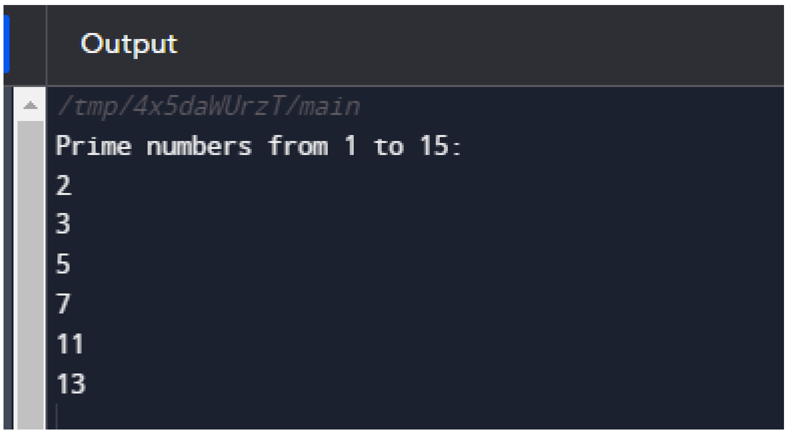
When you run this program, it will output the student number, their score, and the corresponding grade based on the given criteria for each student in the vector `scores`.



```
Output
/tmp/4x5daWUrzT/main
Student 1: Score = 95, Grade = A
Student 2: Score = 85, Grade = B
Student 3: Score = 78, Grade = B
Student 4: Score = 60, Grade = D
Student 5: Score = 70, Grade = C
```

Write a RUST program to print all prime numbers from 1 to n using for loop.

```
fn is_prime(num: u32) -> bool {
    if num <= 1 {
        return false;
    }
    for i in 2..=(num / 2) {
        if num % i == 0 {
            return false;
        }
    }
    true
}

fn print_primes_up_to_n(n: u32) {
    println!("Prime numbers from 1 to {}:", n);
    for num in 2..=n {
        if is_prime(num) {
            println!("{}", num);
        }
    }
}

fn main() {
    let n = 15; // Change this value to set the upper limit
    print_primes_up_to_n(n);
}
```



```
Output
/tmp/4x5daWUrzT/main
Prime numbers from 1 to 15:
2
3
5
7
11
13
```

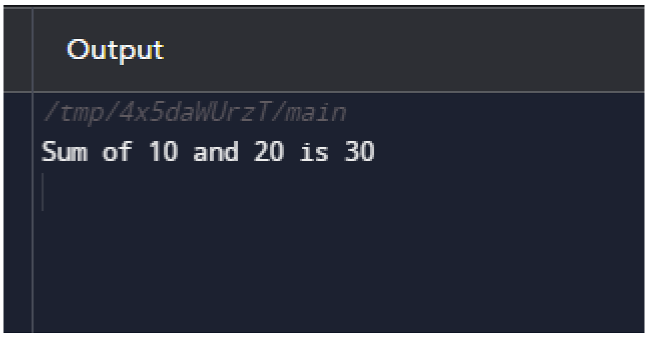
Week-4

- **Write a program to demonstrate the usage of functions in RUST to find sum of two numbers. Pass two values as parameters.**
- **Write a program to create a vector of strings and access the elements. Display all elements in sorted order.**

- **Write a program to demonstrate the usage of functions in RUST to find sum of two numbers. Pass two values as parameters.**

```
fn sum_of_two_numbers(num1: i32, num2: i32) -> i32 {
    num1 + num2
}

fn main() {
    let number1 = 10;
    let number2 = 20;

    let sum = sum_of_two_numbers(number1, number2);

    println!("Sum of {} and {} is {}", number1, number2, sum);
}
```

We define a function sum_of_two_numbers that takes two parameters (num1 and num2) of type i32 and returns their sum.
Inside the main function, we define two variables number1 and number2.
We then call the sum_of_two_numbers function with number1 and number2 as arguments and store the result in the variable sum.
Finally, we print out the sum along with the original numbers.


Output

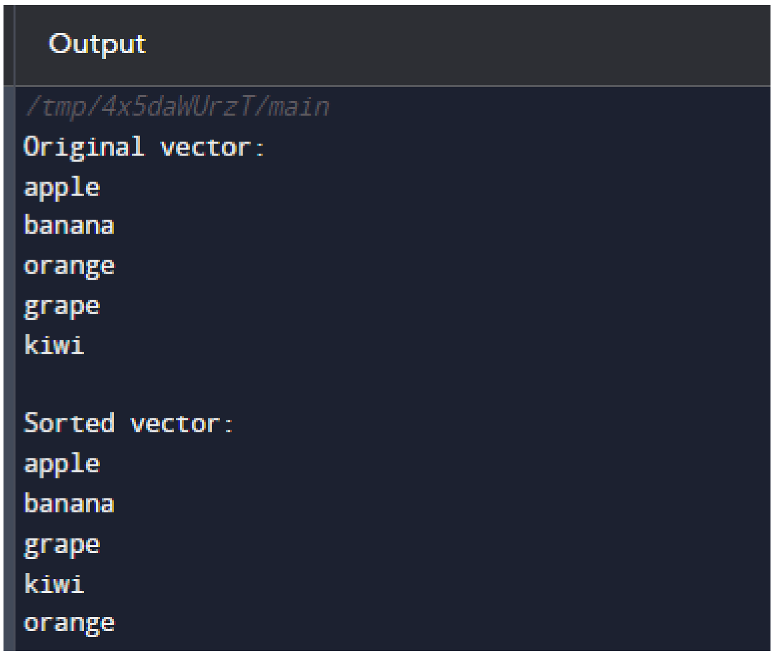
```
/tmp/4x5daWUrzT/main
Sum of 10 and 20 is 30
```

Write a program to create a vector of strings and access the elements. Display all elements in sorted order.

```
fn main() {
    // Create a vector of strings
    let mut strings = vec!["apple", "banana", "orange", "grape", "kiwi"];

    // Accessing elements
    println!("Original vector:");
    for string in &strings {
        println!("{}", string);
    }

    // Sort the vector
    strings.sort();

    // Display elements in sorted order
    println!("\nSorted vector:");
    for string in &strings {
        println!("{}", string);
    }
}
```

In this program:

We first create a vector of strings named strings.
We access the elements of the vector using a for loop and print each element.
Then, we sort the vector using the sort method.
Finally, we again iterate through the vector and print each element, which now appears in sorted order.



```
/tmp/4x5daWUrzT/main
Original vector:
apple
banana
orange
grape
kiwi

Sorted vector:
apple
banana
grape
kiwi
orange
```

Week-5	• Write a RUST program to demonstrate different ways of creating iterators.

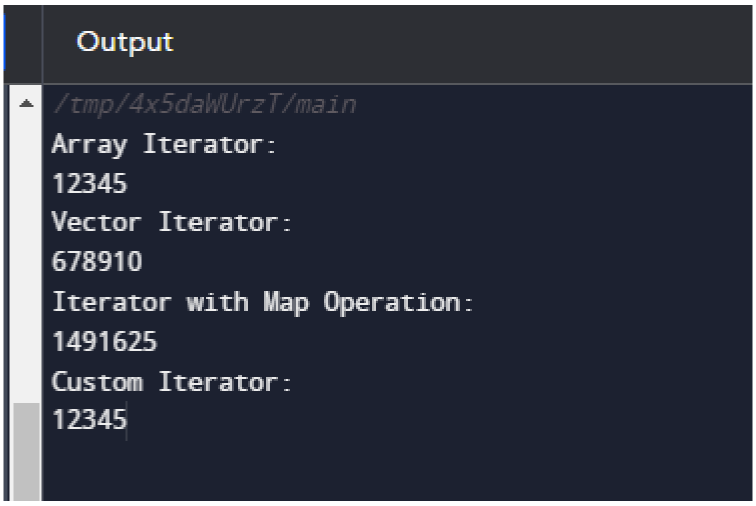
• **Write a RUST program to demonstrate different ways of creating iterators.**

```
fn main() {
  // 1. Creating an iterator using an array
  let array = [1, 2, 3, 4, 5];
  let array_iter = array.iter();
  println!("Array Iterator:");
  for &num in array_iter {
    println!("{}", num);
  }

  // 2. Creating an iterator using a range
  let range_iter = (1..=5).iter(); // Inclusive range
  println!("\nRange Iterator:");
  for num in range_iter {
    println!("{}", num);
  }

  // 3. Creating an iterator using a vector
  let vector = vec![6, 7, 8, 9, 10];
  let vector_iter = vector.iter();
  println!("\nVector Iterator:");
  for &num in vector_iter {
    println!("{}", num);
  }

  // 4. Creating an iterator with a map operation
  let squares_iter = (1..=5).map(|x| x * x);
  println!("\nIterator with Map Operation:");
  for num in squares_iter {
    println!("{}", num);
  }

  // 5. Creating a custom iterator
  struct Counter {
    count: u32,
  }

  impl Iterator for Counter {
    type Item = u32;

    fn next(&mut self) -> Option<Self::Item> {
      if self.count < 5 {
        self.count += 1;
        Some(self.count)
```

```
            } else {
                None
            }
        }
    }

    let mut counter = Counter { count: 0 };
    println!("\nCustom Iterator:");
    for num in counter {
        println!("{}", num);
    }
}
```

In this program:

We create an iterator from an array using iter().
We create an iterator from a range using iter().
We create an iterator from a vector using iter().
We create an iterator with a map operation to calculate squares.
We define a custom iterator Counter and implement the Iterator trait for it, providing a custom next method.

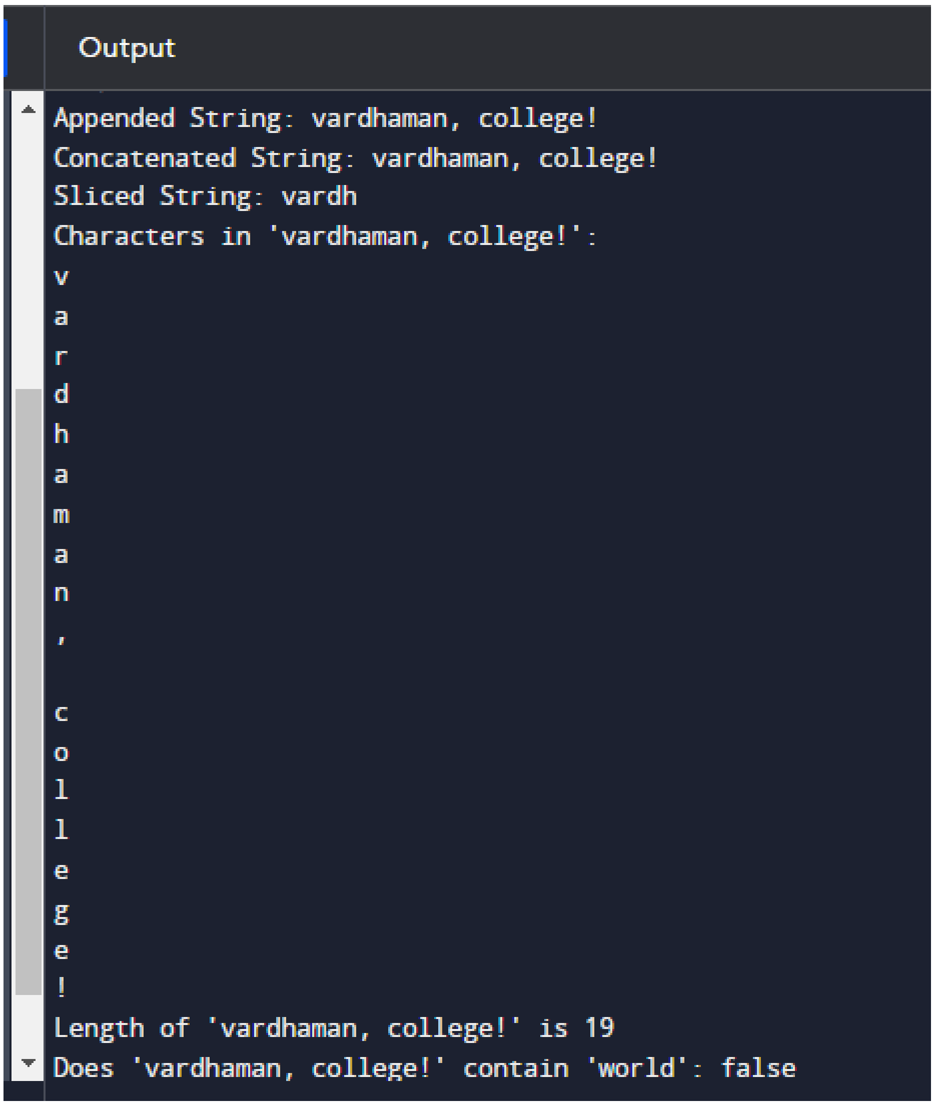


```
/tmp/4x5daWUrzT/main
Array Iterator:
12345
Vector Iterator:
678910
Iterator with Map Operation:
1491625
Custom Iterator:
12345
```

Week-6	• **Write a RUST program to perform all strings operations like creation of string, slicing of string etc.**

•Write a RUST program to perform all strings operations like creation of string, slicing of string etc.

```
fn main() {
    // Create a string
    let mut s = String::from("vardhaman");

    // Append to a string
    s.push_str(", college!");
    println!("Appended String: {}", s);

    // Concatenate strings
    let s1 = String::from("vardhaman, ");
    let s2 = String::from("college!");
    let s3 = s1 + &s2; // s1 is moved here and can no longer be used
    println!("Concatenated String: {}", s3);

    // Slicing a string
    let word = "vardhaman college";
    let sliced = &word[0..5];
    println!("Sliced String: {}", sliced);

    // Iterating over characters in a string
    println!("Characters in '{}':", s3);
    for c in s3.chars() {
        println!("{}", c);
    }

    // String length
    println!("Length of '{}' is {}", s3, s3.len());

    // Check if a string contains a substring
    let check_substring = "world";
    println!("Does '{}' contain '{}': {}", s3, check_substring, s3.contains(check_substring));

    // Replace substring
    let replaced_string = s3.replace("world", "Rust");
    println!("Replaced String: {}", replaced_string);
}
```



```
Appended String: vardhaman, college!
Concatenated String: vardhaman, college!
Sliced String: vardh
Characters in 'vardhaman, college!':
v
a
r
d
h
a
m
a
n
,
 
c
o
l
l
e
g
e
!
Length of 'vardhaman, college!' is 19
Does 'vardhaman, college!' contain 'world': false
```

This Rust program demonstrates:

Creating a string using String::from.
Appending to a string using push_str.
Concatenating strings using the + operator (using the String::from method and borrowing with &).
Slicing a string using the index notation.
Iterating over characters in a string using the chars method.
Getting the length of a string using the len method.
Checking if a string contains a substring using the contains method.
Replacing a substring within a string using the replace method.

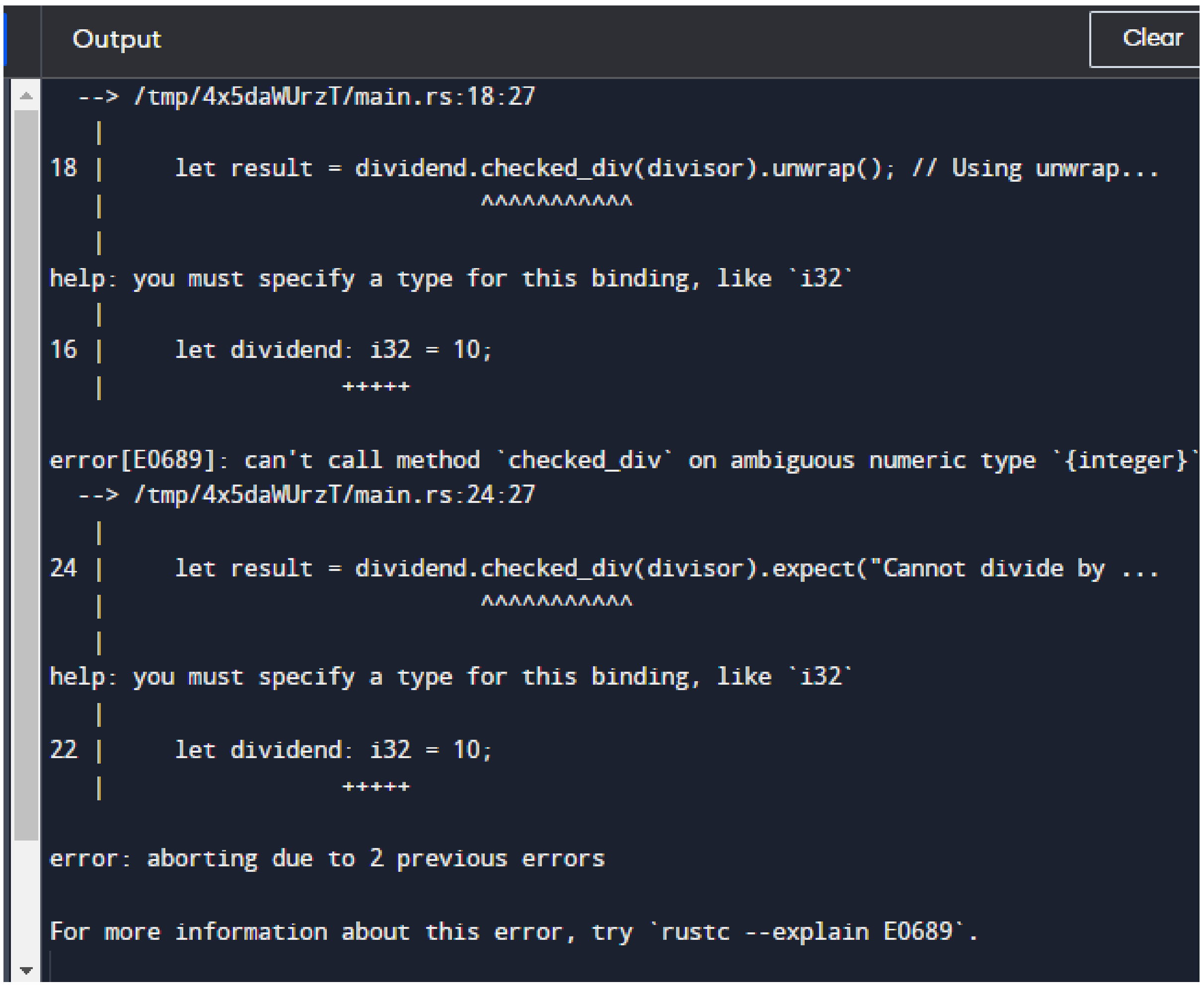
Week-7

- **Write a RUST program to demonstrate recoverable errors using panic, except.**

- Write a RUST program to demonstrate recoverable errors using panic, except.

```
// A function that divides two numbers
fn divide(dividend: i32, divisor: i32) -> i32 {
    if divisor == 0 {
        // If divisor is zero, panic with a custom error message
        panic!("Cannot divide by zero!");
    }
    dividend / divisor
}

fn main() {
    // Example of using panic
    let result = divide(10, 0); // This will cause a panic
    println!("Result: {}", result); // This line will not be executed

    // Example of using Result with unwrap
    let dividend = 10;
    let divisor = 5;
    let result = dividend.checked_div(divisor).unwrap(); // Using unwrap to handle Result
    println!("Result: {}", result);

    // Example of using Result with expect
    let dividend = 10;
    let divisor = 0;
    let result = dividend.checked_div(divisor).expect("Cannot divide by zero!"); // Using
expect to handle Result
    println!("Result: {}", result); // This line will not be executed
}
```

he `divide` function tries to perform division but panics if the divisor is zero.
In the `main` function, we demonstrate using `panic` by calling `divide(10, 0)`, which will cause a panic.
We then demonstrate using `Result` with `unwrap` by using `checked_div` method, which returns a `Result` type. We use `unwrap` to extract the result. If the result is `Ok`, it returns the value, otherwise, it panics with a default error message.
Similarly, we demonstrate using `Result` with `expect` by using the `expect` method instead of `unwrap`. It allows us to provide a custom error message.



```
  --> /tmp/4x5daWUrzT/main.rs:18:27
   |
18 |     let result = dividend.checked_div(divisor).unwrap(); // Using unwrap...
   |                           ^^^^^^^^^^^
   |
help: you must specify a type for this binding, like `i32`
   |
16 |     let dividend: i32 = 10;
   |                 +++++

error[E0689]: can't call method `checked_div` on ambiguous numeric type `{integer}`
  --> /tmp/4x5daWUrzT/main.rs:24:27
   |
24 |     let result = dividend.checked_div(divisor).expect("Cannot divide by ...
   |                           ^^^^^^^^^^^
   |
help: you must specify a type for this binding, like `i32`
   |
22 |     let dividend: i32 = 10;
   |                 +++++

error: aborting due to 2 previous errors

For more information about this error, try `rustc --explain E0689`.
```

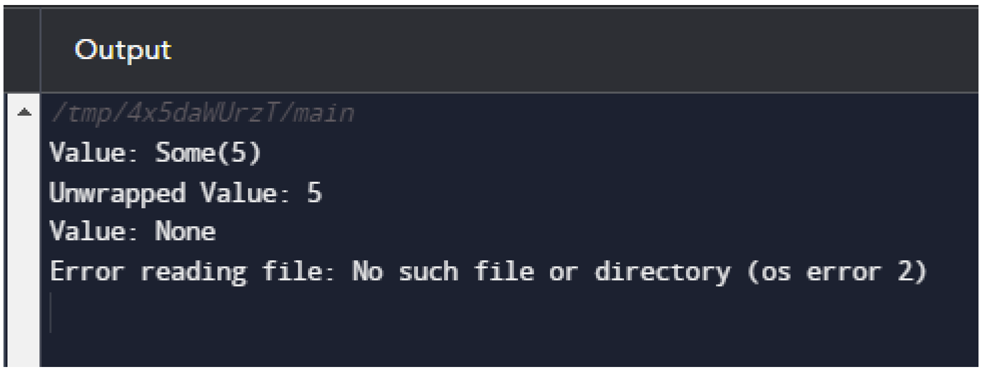
Week-8	• **Write a RUST program to demonstrate about result enum and option enum** • **Write a RUST program to demonstrate data movement and ownership rules in Rust**

• Write a RUST program to demonstrate about result enum and option enum

```
use std::fs::File;
use std::io::{self, Read};

// Function to read a file and return a Result containing the file content or an error
fn read_file_contents(filename: &str) -> Result<String, io::Error> {
   let mut file = File::open(filename)?; // ? operator returns early if there's an error
   let mut contents = String::new();
   file.read_to_string(&mut contents)?;
   Ok(contents)
}

fn main() {
   // Example of using Option enum
   let mut optional_value: Option<i32> = Some(5);
   println!("Value: {:?}", optional_value);

   // Unwrapping Option
   let unwrapped_value = optional_value.unwrap(); // Unwrapping Some variant
   println!("Unwrapped Value: {}", unwrapped_value);

   // Changing to None
   optional_value = None;
   println!("Value: {:?}", optional_value);

   // Unwrapping None - this will cause a panic
   // let unwrapped_value = optional_value.unwrap(); // Uncommenting this line will
cause a panic

   // Example of using Result enum
   let filename = "example.txt";
   let file_contents_result = read_file_contents(filename);
   match file_contents_result {
      Ok(contents) => println!("File contents: {}", contents),
      Err(err) => println!("Error reading file: {}", err),
   }
}
```

In this program:

We define a function read_file_contents that tries to read the contents of a file and returns a Result containing the file content or an error.
In the main function, we demonstrate the usage of Option enum by creating an optional value and then unwrapping it.
We then change the optional value to None and demonstrate how unwrapping None results in a panic.
We also demonstrate the usage of Result enum by trying to read the contents of a file.
We match on the result, handling the Ok variant by printing the file contents and the Err variant by printing the error message.



```
/tmp/4x5daWUrzT/main
Value: Some(5)
Unwrapped Value: 5
Value: None
Error reading file: No such file or directory (os error 2)
```

Write a RUST program to demonstrate data movement and ownership rules in Rust

```
fn main() {
   // Example 1: Moving ownership
   let s1 = String::from("hello");
   let s2 = s1; // s1 is moved to s2
   // println!("s1: {}", s1); // This line will cause a compile error because s1 has been
moved
   println!("s2: {}", s2);

   // Example 2: Cloning data
   let s3 = String::from("world");
   let s4 = s3.clone(); // Cloning data to create a new owner
   println!("s3: {}", s3); // This is valid because we've cloned the data, not moved it
   println!("s4: {}", s4);

   // Example 3: Borrowing data
   let s5 = String::from("foo");
   let len = calculate_length(&s5); // Borrowing s5 rather than moving it
   println!("Length of '{}': {}", s5, len);

   // Example 4: Mutable borrowing
   let mut s6 = String::from("bar");
   change_string(&mut s6); // Mutably borrowing s6
```

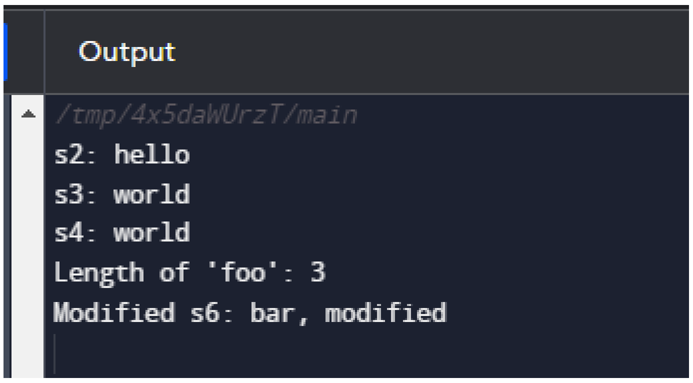
```
    println!("Modified s6: {}", s6);
}

// Function to calculate the length of a string without taking ownership
fn calculate_length(s: &String) -> usize {
    s.len()
}

// Function to modify a string without taking ownership
fn change_string(s: &mut String) {
    s.push_str(", modified");
}
```

In this program:

We demonstrate moving ownership from s1 to s2. After the move, s1 is no longer valid to use.
We show how to clone data from s3 to s4, creating a separate owner for the data.
We demonstrate borrowing data with the calculate_length function, which takes a reference to a String.
We demonstrate mutable borrowing with the change_string function, which takes a mutable reference to a String and modifies it.
These examples illustrate Rust's ownership rules:

Each value in Rust has a variable that's its owner.
There can only be one owner at a time.
When the owner goes out of scope, the value will be dropped.
Additionally, Rust enforces the borrowing rules:

You can have either one mutable reference or any number of immutable references to a piece of data at any given time.
References must always be valid and non-null.
References are automatically checked at compile time to ensure these rules are followed, preventing data races and other memory safety issues.


Output

```
/tmp/4x5daWUrzT/main
s2: hello
s3: world
s4: world
Length of 'foo': 3
Modified s6: bar, modified
```

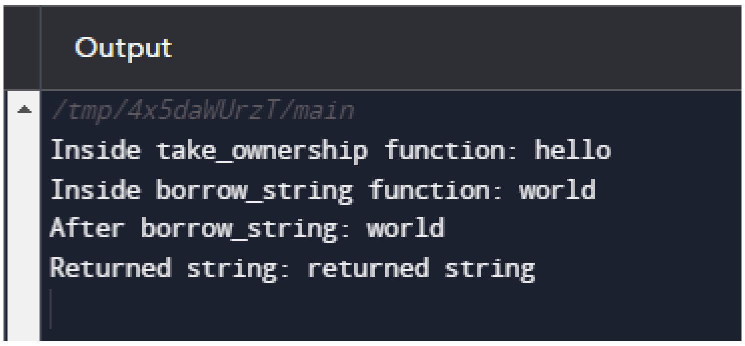
Week-9	• **Write a RUST program to demonstrate ownership in functions.**

Write a RUST program to demonstrate ownership in functions.

```
// Function that takes ownership of a String
fn take_ownership(s: String) {
    println!("Inside take_ownership function: {}", s);
} // When the function ends, `s` goes out of scope and its memory is freed

// Function that borrows a String
fn borrow_string(s: &String) {
    println!("Inside borrow_string function: {}", s);
} // Here, `s` is a reference to a String, so the memory it refers to isn't freed when the function ends

fn main() {
    // Creating a String
    let s1 = String::from("hello");

    // Passing ownership to a function
    take_ownership(s1);
    // println!("After take_ownership: {}", s1); // This line will cause a compile error since ownership was transferred to the function

    // Borrowing a String
    let s2 = String::from("world");
    borrow_string(&s2); // Passing a reference to the String
    println!("After borrow_string: {}", s2); // This is valid because we only borrowed s2, ownership wasn't transferred

    // Function returning a String
    let returned_string = return_string();
    println!("Returned string: {}", returned_string);
}

// Function returning a String
fn return_string() -> String {
    let s = String::from("returned string");
    s // Ownership is transferred to the caller when s is returned
}
```

In this program:

We define a function take_ownership that takes ownership of a String and prints its value. After the function ends, the memory allocated for the String is freed.
We define a function borrow_string that borrows a reference to a String and prints its value. The memory isn't freed when the function ends because it only borrowed the

	reference.
	In the main function, we demonstrate passing ownership to take_ownership, which means we can't use s1 afterward since ownership was transferred.
	We demonstrate borrowing s2 in borrow_string, allowing us to use s2 afterward.
	We define a function return_string that creates and returns a String, transferring ownership to the caller.
	
	Output
	`/tmp/4x5daWUrzT/main`
	`Inside take_ownership function: hello`
	`Inside borrow_string function: world`
	`After borrow_string: world`
	`Returned string: returned string`

Week-10	• Demonstrate Building and Running Project with Cargo in Rust

Demonstrate Building and Running Project with Cargo in Rust

Certainly! Cargo is Rust's package manager and build system. It makes it easy to create, build, and manage Rust projects. Here's a step-by-step guide to demonstrate building and running a project with Cargo:

1. **Create a New Project**: Open a terminal or command prompt, navigate to the directory where you want to create your project, and run the following command:

```
cargo new my_project_name
```

Replace `my_project_name` with the desired name of your project.

2. **Navigate to the Project Directory**: After creating the project, navigate to its directory:

```
cd my_project_name
```

3. **Edit `src/main.rs`**: Open the `src/main.rs` file in your preferred text editor and write your Rust code.

4. **Build the Project**: To build the project, run the following command:

```
cargo build
```

This command will compile the project and generate an executable binary in the `target/debug` directory.

5. **Run the Project**: To run the project, use the following command:

```
cargo run
```

This command will compile the project if necessary and then run the generated executable.

6. **Run Tests**: If you have tests in your project, you can run them using the following command:

```
cargo test
```

This command will compile the tests and run them.

7. **Release Build**: For a release build with optimizations, use the following command:

```
cargo build --release
```

This command will create a release build with optimizations in the `target/release` directory.

8. **Clean Build**: To clean the build artifacts, use the following command:

```
cargo clean
```

This command will remove the `target` directory and its contents.

That's it! You've successfully built and run a Rust project using Cargo. You can now continue to develop your project, add dependencies, write tests, and more, all with the help of Cargo.

Week-11	• **Write a program to demonstrate Defining, Implementing and Using a Trait in Rust** • **Write a RUST program to demonstrate about pattern matching.**

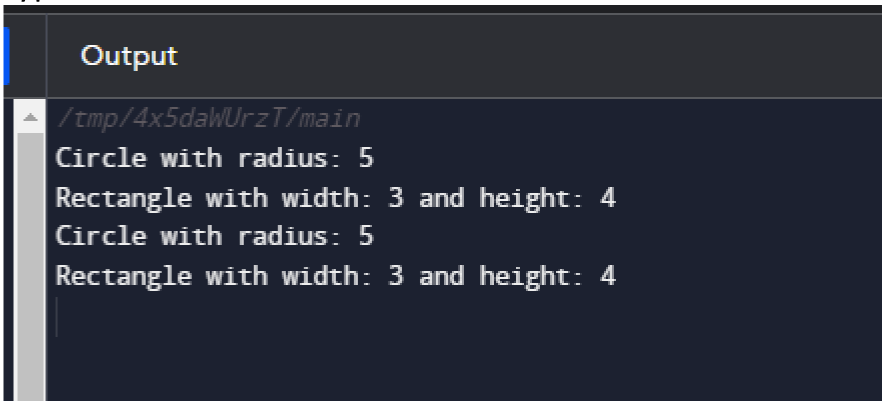
- **Write a program to demonstrate Defining, Implementing and Using a Trait in Rust**

```
// Define a trait named `Printable`
trait Printable {
   // Method signature without default implementation
   fn print(&self);
}

// Implement the `Printable` trait for the `Circle` struct
struct Circle {
   radius: f64,
}

impl Printable for Circle {
   // Implement the `print` method for the `Circle` struct
   fn print(&self) {
      println!("Circle with radius: {}", self.radius);
   }
}

// Implement the `Printable` trait for the `Rectangle` struct
struct Rectangle {
   width: f64,
   height: f64,
}

impl Printable for Rectangle {
   // Implement the `print` method for the `Rectangle` struct
   fn print(&self) {
      println!("Rectangle with width: {} and height: {}", self.width,
self.height);
   }
}
```

```
// Function that takes any type that implements the `Printable` trait
fn print_shape<T: Printable>(shape: T) {
    shape.print();
}

fn main() {
    let circle = Circle { radius: 5.0 };
    let rectangle = Rectangle { width: 3.0, height: 4.0 };

    // Call the `print` method on `Circle` and `Rectangle` instances
    circle.print();
    rectangle.print();

    // Use the `print_shape` function to print any shape
    print_shape(circle);
    print_shape(rectangle);
}
```

In this program:

We define a trait named Printable, which has a method named print.
We implement the Printable trait for two different structs: Circle and Rectangle. Each implementation provides its own implementation for the print method.
We define a generic function named print_shape, which takes any type that implements the Printable trait and calls its print method.
In the main function, we create instances of Circle and Rectangle, and then call their print methods directly. We also use the print_shape function to print shapes without knowing their concrete types.



```
/tmp/4x5daWUrzT/main
Circle with radius: 5
Rectangle with width: 3 and height: 4
Circle with radius: 5
Rectangle with width: 3 and height: 4
```

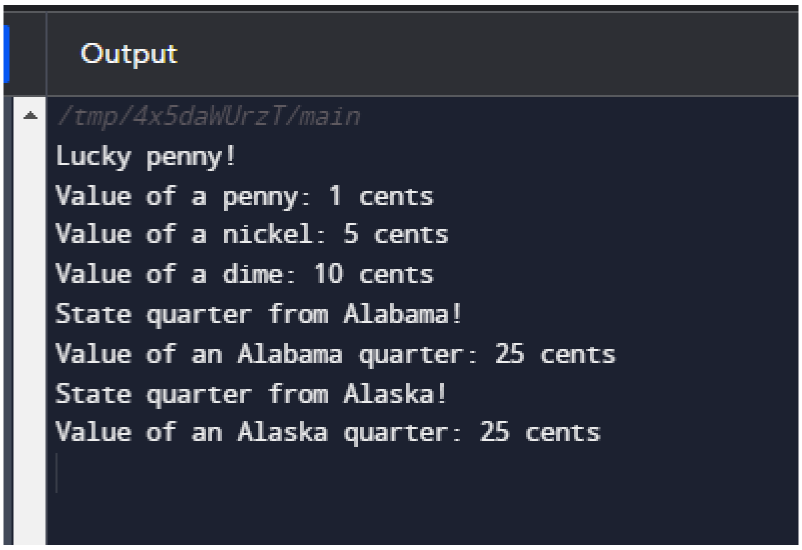
Write a RUST program to demonstrate about pattern matching.

```
#[derive(Debug)]
enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter(UsState),
}

#[derive(Debug)]
enum UsState {
    Alabama,
    Alaska,
    // Add more states if needed
}

fn value_in_cents(coin: Coin) -> u8 {
    match coin {
        Coin::Penny => {
            println!("Lucky penny!");
            1
        },
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter(state) => {
            println!("State quarter from {:?}!", state);
            25
        },
    }
}

fn main() {
    let penny = Coin::Penny;
    let nickel = Coin::Nickel;
    let dime = Coin::Dime;
    let alabama_quarter = Coin::Quarter(UsState::Alabama);
    let alaska_quarter = Coin::Quarter(UsState::Alaska);

    println!("Value of a penny: {} cents", value_in_cents(penny));
    println!("Value of a nickel: {} cents", value_in_cents(nickel));
    println!("Value of a dime: {} cents", value_in_cents(dime));
    println!("Value of an Alabama quarter: {} cents",
value_in_cents(alabama_quarter));
```

```
    println!("Value of an Alaska quarter: {} cents", value_in_cents(alaska_quarter));
}
```

In this program:

We define an enum Coin representing different types of coins and an enum UsState representing US states.
We implement a function value_in_cents that calculates the value of a coin in cents using pattern matching with the match keyword.
Inside the match expression, we match each variant of the Coin enum and destructure it to extract associated values.
We also demonstrate matching nested enums by pattern matching on the Coin::Quarter variant, which contains a UsState enum.
In the main function, we create instances of different coins and call the value_in_cents function to calculate their values.



```
/tmp/4x5daWUrzT/main
Lucky penny!
Value of a penny: 1 cents
Value of a nickel: 5 cents
Value of a dime: 10 cents
State quarter from Alabama!
Value of an Alabama quarter: 25 cents
State quarter from Alaska!
Value of an Alaska quarter: 25 cents
```

Week-12	• **Implement Generic struct and Generic Functions in Rust.**

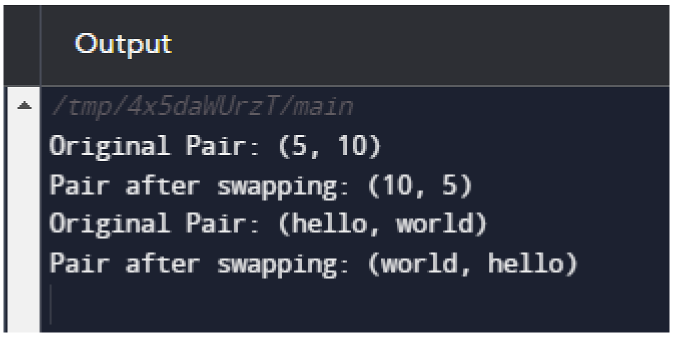
Implement Generic struct and Generic Functions in Rust.

```
// Define a generic struct named `Pair`
struct Pair<T> {
   first: T,
   second: T,
}

// Implement methods for the generic struct
impl<T> Pair<T> {
   // Constructor method to create a new Pair
   fn new(first: T, second: T) -> Self {
      Pair { first, second }
   }

   // Method to get the first element of the Pair
   fn get_first(&self) -> &T {
      &self.first
   }

   // Method to get the second element of the Pair
   fn get_second(&self) -> &T {
      &self.second
   }
}

// Define a generic function named `swap_pair_elements`
fn swap_pair_elements<T>(pair: &mut Pair<T>) {
   std::mem::swap(&mut pair.first, &mut pair.second);
}

fn main() {
   // Create a Pair of integers
   let mut pair_of_ints = Pair::new(5, 10);

   // Print the original Pair
   println!("Original Pair: ({}, {})", pair_of_ints.get_first(),
pair_of_ints.get_second());
```

```
    // Swap elements of the Pair
    swap_pair_elements(&mut pair_of_ints);

    // Print the Pair after swapping
    println!("Pair after swapping: ({}, {})", pair_of_ints.get_first(), pair_of_ints.get_second());

    // Create a Pair of strings
    let mut pair_of_strings = Pair::new("hello", "world");

    // Print the original Pair
    println!("Original Pair: ({}, {})", pair_of_strings.get_first(), pair_of_strings.get_second());

    // Swap elements of the Pair
    swap_pair_elements(&mut pair_of_strings);

    // Print the Pair after swapping
    println!("Pair after swapping: ({}, {})", pair_of_strings.get_first(), pair_of_strings.get_second());
}
```

In this program:

We define a generic struct named Pair<T> that holds two elements of the same type T.
We implement methods for the Pair<T> struct, including a constructor method new, and methods to get the first and second elements.
We define a generic function named swap_pair_elements that takes a mutable reference to a Pair<T> and swaps its elements.
In the main function, we demonstrate creating pairs of integers and strings, swapping their elements, and printing the results.



```
/tmp/4x5daWUrzT/main
Original Pair: (5, 10)
Pair after swapping: (10, 5)
Original Pair: (hello, world)
Pair after swapping: (world, hello)
```

Week-13	• **Write a RUST program for performing following FILE operations:** **a) Opening a file b) Reading from a file** **c) Writing to a file** **d) Removing a file e) Appending to a file**

Write a RUST program for performing following FILE operations:

```
use std::fs::{File, OpenOptions};
use std::io::{self, Read, Write};
fn main() -> io::Result<()> {
   // File paths
   let file_path = "example.txt";
   let append_file_path = "append_example.txt";
   // Opening a file
   let mut file = File::open(file_path)?;
   // Reading from a file
   let mut contents = String::new();
   file.read_to_string(&mut contents)?;
   println!("Contents of the file:");
   println!("{}", contents);
   // Writing to a file
   let mut write_file = File::create(file_path)?;
   write_file.write_all(b"Hello, world!\n")?;
   // Appending to a file
   let mut append_file =
OpenOptions::new().append(true).open(append_file_path)?;
   append_file.write_all(b"Appending content\n")?;

   // Removing a file
   std::fs::remove_file(file_path)?;

   Ok(())
}
```

In this Rust program:

We use File::open to open a file, File::create to create a file, and OpenOptions::new().append(true).open to open a file in append mode.
We use file.read_to_string to read from a file and write_file.write_all and append_file.write_all to write to files.
Finally, we use std::fs::remove_file to remove a file.

Week-14	• **Build a multithreaded web server using RUST.**

Build a multithreaded web server using RUST

```
use std::io::{Read, Write};
use std::net::{TcpListener, TcpStream};
use std::thread;
fn handle_client(mut stream: TcpStream) {
   let mut buffer = [0; 1024];
   stream.read(&mut buffer).unwrap();
   println!("Received request: {}", String::from_utf8_lossy(&buffer[..]));
   let response = "HTTP/1.1 200 OK\r\nContent-Type: text/plain\r\n\r\nHello, world!";
   stream.write(response.as_bytes()).unwrap();
   stream.flush().unwrap();
}
fn main() {
   let listener = TcpListener::bind("127.0.0.1:8080").unwrap();
   println!("Server listening on port 8080...");

   for stream in listener.incoming() {
      let stream = stream.unwrap();
      thread::spawn(move || {
         handle_client(stream);
      });
   }
}
```

In this example:

We create a function handle_client that will handle each incoming client connection. It reads the incoming HTTP request, prints it to the console, and sends back a simple HTTP response.
In the main function, we bind the server to the address 127.0.0.1:8080 and start listening for incoming connections.
For each incoming connection, we spawn a new thread to handle it. This allows multiple connections to be processed concurrently.
Inside each thread, we call the handle_client function to handle the incoming connection.
To run this code, you'll need to add hyper crate to your Cargo.toml file:

```
[dependencies]
hyper = "0.14.11"
```

Then, run cargo build and cargo run to build and run the server.