# **RUST**

# #️ The Basics

* To download the Rust package

$ cargo new Project01

* To build the package

$ cargo build

1. To run the package

$ cargo run

* "Hello, World!”

// Project01

fn main() {

    println!("Hello, world!");

}

Output:-

$ cargo run

Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.09s

Running `target\debug\Project01.exe`

Hello, world!

* Variables and Mutabilty
* By default, variables are immutable. When a variable is immutable, once a value is bound to a name, you can’t change that value.

fn main() {

    let x = 5;

    println!("The value of x is: {x}");

    x = 6;

    println!("The value of x is: {x}");

}

Output:-

$ cargo run

Compiling variables v0.1.0 (file:///projects/variables)

error[E0384]: cannot assign twice to immutable variable `x`

--> src/main.rs:4:5

|

2 | let x = 5;

| - first assignment to `x`

3 | println!("The value of x is: {x}");

4 | x = 6;

| ^^^^^ cannot assign twice to immutable variable

|

help: consider making this binding mutable

|

2 | let mut x = 5;

| +++

For more information about this error, try `rustc --explain E0384`.

error: could not compile `variables` (bin "variables") due to 1 previous error

* You received the error message cannot assign twice to immutable variable `x` because you tried to assign a second value to the immutable x variable.  
    
  Mutability can be very useful, and can make code more convenient to write. Although variables are immutable by default, you can make them mutable by adding mut in front of the variable name. Adding mut also conveys intent to future readers of the code by indicating that other parts of the code will be changing this variable’s value.
* For example:-

//Project02

fn main() {

    let mut x = 5;

    println!("The value of x is: {x}");

    x = 6;

    println!("The value of x is: {x}");

}

Output:-

$ cargo run

Compiling Project02 v0.1.0 (C:\Users\Harsh\Desktop\Rust\Project02)

Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.24s

Running `target\debug\Project02.exe`

The value of x is: 5

The value of x is: 6

* Like immutable variables, *constants* are values that are bound to a name and are not allowed to change, but there are a few differences between constants and variables.
* First, you aren’t allowed to use mut with constants. Constants aren’t just immutable by default—they’re always immutable. You declare constants using the const keyword instead of the let keyword, and the type of the value *must* be annotated.
* Constants can be declared in any scope, including the global scope, which makes them useful for values that many parts of code need to know about.
* The last difference is that constants may be set only to a constant expression, not the result of a value that could only be computed at runtime.
* For example:-

//Project02

fn main() {

    let mut x = 5;

    println!("The value of x is: {x}");

    x = 6;

    println!("The value of x is: {x}");

    const SECOND: i8 = 60;

    println!("The value of SECOND is: {SECOND}");

}

Output :-

The value of x is: 5

The value of x is: 6

The value of SECOND is: 60

# #️ Data Types

Keep in mind that Rust is a statically typed language, which means that it must know the types of all variables at compile time.

We’ll look at two data type subsets: scalar and compound.

* 1. Scalar Types

A scalar type represents a single value. Rust has four primary scalar types: integers, floating-point numbers, Booleans, and characters.

| **Length** | **Signed** | **Unsigned** |
| --- | --- | --- |
| 8-bit | i8 | u8 |
| 16-bit | i16 | u16 |
| 32-bit | i32 | u32 |
| 64-bit | i64 | u64 |
| 128-bit | i128 | u128 |
| arch | isize | usize |

1. Integer Types



* + **Integer Literals in Rust**

| **Number literals** | **Example** |
| --- | --- |
| Decimal | 98\_222 |
| Hex | 0xff |
| Octal | 0o77 |
| Binary | 0b1111\_0000 |
| Byte (u8 only) | b'A' |

* Code:-

//Project03

fn main() {

    let x: i8 = 10;

    println!("Value of x :- {}", x);

    let decimal = 02\_55;

    let hex = 0xff;

    let octal = 0o77;

    let binary = 0b1111\_1111;

    println!("Decimal :- {}", hex);

    println!("Octal :- {}", octal);

    println!("Binary :- {}", binary);

    println!("Decimal :- {}", decimal);

}

Output:-

Value of x :- 10

Decimal :- 255

Octal :- 63

Binary :- 255

Decimal :- 255

1. Floating-Point Types

* Rust also has two primitive types for *floating-point numbers*, which are numbers with decimal points. Rust’s floating-point types are f32 and f64, which are 32 bits and 64 bits in size, respectively. The default type is f64 because on modern CPUs, it’s roughly the same speed as f32 but is capable of more precision. All floating-point types are signed.
* Code:-

//Project04

fn main() {

    let x = 2.0;

    let y: f32 = 3.0;

    println!("x :- {} , y :- {}", x, y);

}

Output:-

x :- 2 , y :- 3

1. The Boolean Type

* As in most other programming languages, a Boolean type in Rust has two possible values: true and false. Booleans are one byte in size. The Boolean type in Rust is specified using bool.
* Code

//Project05

fn main() {

    let t = true;

    let f: bool = false;

    println!("{}", t);

    println!("{}", f);

}

Output:-

True

false

1. **The Character Type**

* Code

//Project06

fn main() {

    let c = 'z';

    let z: char = 'ℤ'; // with explicit type annotation

    println!("c :- {} , z :- {}", c, z);

}

Output:-

c :- z , z :- ℤ

1. Numeric Operations

* Rust supports the basic mathematical operations you’d expect for all the number types: addition, subtraction, multiplication, division, and remainder.
* CODE

//Project07

fn main() {

    // addition

    let x: i32 = 10;

    let y: i32 = 20;

    let sum: i32 = x + y;

    println!("sum :- {}", sum);

    // subtraction

    let a: f32 = 3.0;

    let b: f32 = 5.1;

    let difference: f32 = a - b;

    println!("difference :- {}", difference);

    // multiplication

    let x: i32 = 5;

    let y: i32 = 7;

    let product1: i32 = x \* y;

    println!("integer product: {}", product1);

    let x: f32 = 5.1;

    let y: f32 = 7.2;

    let product2: f32 = x \* y;

    println!("float product: {}", product2);

    // division

    let quotient: f32 = 56.7 / 32.2;

    let truncated: i32 = -5 / 3; // Results in -1

    println!("float quotient :- {}", quotient);

    println!("integer quotient :- {}", truncated);

    // remainder

    let remainder: i32 = 43 % 5;

    println!("remainder :- {}", remainder);

}

Output:-

sum :- 30

difference :- -2.1

integer product: 35

float product: 36.719997

float quotient :- 1.7608695

integer quotient :- -1

remainder :- 3

* Standard Input in Rust
  + Taking input of String
* Taking input in Rust involves using the std::io module.

//Project08

use std::io;

fn main() {

    println!("Please enter your input:");

    let mut input = String::new();

    io::stdin()

        .read\_line(&mut input)

        .expect("Failed to read line");

    println!("You entered: {}", input.trim());

}

Output:-

Please enter your input:

Dev

You entered: Dev

* + Taking input of Integer Value

//Project09

use std::io;

fn main() {

    println!("Please enter a number:");

    let mut input = String::new();

    io::stdin()

        .read\_line(&mut input)

        .expect("Failed to read line");

    // Convert the input to a number

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

    println!("You entered the number: {}", number);

}

Output:-

Please enter a number:

3

You entered the number: 3

* + Taking input of Float and Character Value

//Project10

use std::io;

fn main() {

    // Taking float input

    println!("Please enter a float value:");

    let mut float\_input = String::new();

    io::stdin()

        .read\_line(&mut float\_input)

        .expect("Failed to read line");

    let float\_value: f64 = float\_input

        .trim()

        .parse()

        .expect("Please enter a valid float number");

    println!("You entered the float: {}", float\_value);

    // Taking character input

    println!("Please enter a single character:");

    let mut char\_input = String::new();

    io::stdin()

        .read\_line(&mut char\_input)

        .expect("Failed to read line");

    let char\_value: char = char\_input

        .trim()

        .chars()

        .next()

        .expect("Please enter a valid character");

    println!("You entered the character: {}", char\_value);

}

Output:-

Please enter a float value:

1.2

You entered the float: 1.2

Please enter a single character:

a

You entered the character: a

* Control Flow in Rust
* The ability to run some code depending on whether a condition is true and to run some code repeatedly while a condition is true are basic building blocks in most programming languages. The most common constructs that let you control the flow of execution of Rust code are if expressions and loops.
  + 1. if Expressions
* An if expression allows you to branch your code depending on conditions. You provide a condition and then state, “If this condition is met, run this block of code. If the condition is not met, do not run this block of code.”
  + Basic if-else

//Project15

fn main() {

    let number = 0;

    if number > 0 {

        println!("Number is positive");

    } else if number < 0 {

        println!("Number is negative");

    } else {

        println!("Number is zero");

    }

}

Output:-

Number is zero

* + if as an Expression
* In Rust, if can return a value because it’s an expression. This means you can assign the result of an if statement to a variable.

//Project16

fn main() {

    let number = 3;

    let result = if number > 0 {

        "positive"

    } else {

        "non-positive"

    };

    println!("The number is {}", result);

}

Output:-

The number is positive

* + 1. Repetition with Loops
* It’s often useful to execute a block of code more than once. For this task, Rust provides several *loops*, which will run through the code inside the loop body to the end and then start immediately back at the beginning.in
* Rust has three kinds of loops: loop, while, and for.
  + Returning Values from Loops

//Project17

fn main() {

    let mut counter = 0;

    let result = loop {

        counter += 1;

        if counter == 10 {

            break counter \* 2;

        }

    };

    println!("The result is {result}");

}

Output:-

The result is 20

* + Labeled Loops

//Project18

fn main() {

    let mut count = 0;

    'counting\_up: loop {

        println!("count = {count}");

        let mut remaining = 10;

        loop {

            println!("remaining = {remaining}");

            if remaining == 9 {

                break;

            }

            if count == 2 {

                break 'counting\_up;

            }

            remaining -= 1;

        }

        count += 1;

    }

    println!("End count = {count}");

}

Output:-

count = 0

remaining = 10

remaining = 9

count = 1

remaining = 10

remaining = 9

count = 2

remaining = 10

End count = 2

* + 1. while loop
* A while loop runs as long as its condition evaluates to true.
  + Basic while loop

//Project19

fn main() {

    let mut number = 0;

    while number < 5 {

        number += 1;

        println!("Number: {}", number);

    }

}

Output:-

Number: 1

Number: 2

Number: 3

Number: 4

Number: 5

* + while loop with continue and break

//Project20

fn main() {

    let mut number = 0;

    while number < 5 {

        number += 1;

        if number == 3 {

            println!("Skipping 3");

            continue;  // Skip the rest of this iteration

        }

        println!("Number: {}", number);

        if number == 4 {

            break;  // Exit the loop early

        }

    }

}

Output:-

Number: 1

Number: 2

Skipping 3

Number: 4

* + 1. for loop
* Basic for loop(Inclusive forLoop)

//Project21

fn main() {

    for number in 1..=5 {  // Inclusive range (1 to 5)

        println!("Number: {}", number);

    }

}

Output:-

Number: 1

Number: 2

Number: 3

Number: 4

Number: 5

* Basic for loop

//Project23

fn main() {

    for number in 1..5 {

        println!("Number: {}", number);

    }

}

Output:-

Number: 1

Number: 2

Number: 3

Number: 4

* Reverse for loop

//Project22

fn main() {

    for number in (1..5).rev() {

        println!("{number}!");

    }

    println!("LIFTOFF!!!");

}

Output:-

4!

3!

2!

1!

LIFTOFF!!!

* 2. Compound Types
* Compound types can group multiple values into one type. Rust has two primitive compound types: tuples and arrays.

1. The Tuple Type

* A tuple is a general way of grouping together a number of values with a variety of types into one compound type. Tuples have a fixed length: once declared, they cannot grow or shrink in size.
* We create a tuple by writing a comma-separated list of values inside parentheses. Each position in the tuple has a type, and the types of the different values in the tuple don’t have to be the same.
  + - Creating, Destructuring and Accessing Elements of Tuple
* Destructuring means extracting tuple elements into variables.

//Project11

fn main() {

    let tup1: (i32, f64, u8) = (500, 6.4, 1);

    println!("The first value is: {}", tup1.0);

    println!("The second value is: {}", tup1.1);

    println!("The third value is: {}", tup1.2);

    let tup2 = (6.6, 9.9, 100);

    let (x, y, z) = tup2; // Destructuring

    println!("The value of x is: {}",x);

    println!("The value of y is: {y}");

    println!("The value of z is: {z}");

}

Output:-

The first value is: 500

The second value is: 6.4

The third value is: 1

The value of x is: 6.6

The value of y is: 9.9

The value of z is: 100

* + - Nested Tuple
* Access elements inside nested tuples.

//Project12

fn main() {

    let nested\_tuple = ((1, 2), (3, 4));

    println!("Accessing inner element: {}", (nested\_tuple.0).1);

}

Output:-

Accessing inner element: 2

* + - Tuple Comparison

Compare tuples (if their elements are comparable)

//Project13

fn main() {

    let tuple1 = (1, 2);

    let tuple2 = (1, 3);

    println!("Are they equal? {}", tuple1 == tuple2);

    println!("Is tuple1 less than tuple2? {}", tuple1 < tuple2);

}

Output:-

Are they equal? false

Is tuple1 less than tuple2? true

1. The Array Type

* Another way to have a collection of multiple values is with an *array*. Unlike a tuple, every element of an array must have the same type. Unlike arrays in some other languages, arrays in Rust have a fixed length.
* To create an array in Rust with a user-defined length, you'll need to use **vectors (Vec<T>)** instead of arrays because arrays in Rust must have their size fixed at compile time. Vectors are dynamically resizable and ideal for this scenario.
  + - Declaring, Accessing and Modifying Elements of Array in Rust

let a: [i32; 5] = [1, 2, 3, 4, 5];

* Here, i32 is the type of each element. After the semicolon, the number 5 indicates the array contains five elements.

let a = [3; 5];

* The array named “a” will contain 5 elements that will all be set to the value 3 initially. This is the same as writing let a = [3, 3, 3, 3, 3]; but in a more concise way.

//Project14

fn main() {

    //Declaring Arrays

    let array: [i32; 5] = [1, 2, 3, 4, 5];

    //Accessing Elements

    println!("First element: {}", array[0]);

    println!("Second element: {}", array[1]);

    //Declaring Arrays

    let mut array\_with\_same\_values: [i32; 5] = [0; 5];

    // Modifying elements of array

    array\_with\_same\_values[0] = 10;

    array\_with\_same\_values[1] = 20;

    array\_with\_same\_values[2] = 30;

    array\_with\_same\_values[3] = 40;

    array\_with\_same\_values[4] = 50;

    //Accessing Elements

    println!("Array with same values: {:?}", array\_with\_same\_values);

}

Output:-

First element: 1

Second element: 2

Array with same values: [10, 20, 30, 40, 50]

* + - Array length and Slicing

//Project24

fn main() {

    let array = [5, 10, 15];

    // Length

    println!("Length of array: {}", array.len());

    // Slicing

    for i in 0..array.len() {

        println!("Element at index {}: {}", i, array[i]);

    }

}

Output:-

Length of array: 3

Element at index 0: 5

Element at index 1: 10

Element at index 2: 15

* + - Copying and Cloning

//Project25

fn main() {

    let array1 = [1, 2, 3];

    let array2 = array1; // This creates a copy of `array1`

    println!("array1: {:?}", array1);

    println!("array2: {:?}", array2);

}

Output:-

array1: [1, 2, 3]

array2: [1, 2, 3]

* + - Reversing an Array

//Project26

fn main() {

    let mut array = [1, 2, 3, 4];

    array.reverse();

    println!("Reversed array: {:?}", array);

}

Output :-

Reversed array: [4, 3, 2, 1]

* + - Sorting Array

//Project27

fn main() {

    let mut array = [5, 3, 8, 1];

    array.sort();

    println!("Sorted array: {:?}", array);

}

Output:-

Sorted array: [1, 3, 5, 8]

1. The Vector Type

* Vectors in Rust are dynamic arrays that can grow or shrink in size. They are part of Rust's standard library and are implemented as the Vec<T> type. Vectors are widely used because of their flexibility and ease of use.
* Declaring, Accessing and Modifying Elements of Array in Rust

//Project28

fn main() {

    // Using the vec! macro with initial values

    let mut v1 = vec![10, 20, 30, 40, 50];

    // Vector with repeated values

    let v2 = vec![0; 5]; // A vector of 5 zeros: [0, 0, 0, 0, 0]

    v1[0] = 11;

    println!("First element: {}", v1[0]);

    println!("First element: {}", v2[0]);

}

Output:-

First element: 11

First element: 0

2nd Approach to define Vector(Best) 🡺

* Declaring, Accessing and Modifying Elements of Array in Rust
* Length of Vector
* Sorting of Vector
* Check if vector is empty or not
* Clearing a vector

//Project29

use std::io;

fn main() {

    let mut input = String::new();

    // Take the length of the vector from the user

    println!("Enter the length of the vector:");

    io::stdin()

        .read\_line(&mut input)

        .expect("Failed to read input");

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

    // Create an empty vector

    let mut v: Vec<i32> = Vec::with\_capacity(length);

    // Populate the vector with user input

    for i in 0..length {

        input.clear();

        println!("Enter element {}: ", i + 1);

        io::stdin()

            .read\_line(&mut input)

            .expect("Failed to read input");

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

        v.push(element);

    }

    // Print all elements using a for loop

    println!("The elements of the vector are:");

    for i in 0..length {

        println!("Element {}: {}", i + 1, v[i]);

    }

    // Modifying elements

    v[0] = 1000;

    println!("Modified elements :- {:?}", v[0]);

    // Length of vector

    println!("Length: {}", v.len());

    // Checking if a Vector is Empty

    println!("Is vector empty? {}", v.is\_empty());

    // Sorting a Vector

    v.sort();

    println!("{:?}", v);

    // Clearing a Vector

    v.clear();

    println!("{:?}", v);

}

Output:-

Enter the length of the vector:

4

Enter element 1:

1

Enter element 2:

2

Enter element 3:

8

Enter element 4:

3

The elements of the vector are:

Element 1: 1

Element 2: 2

Element 3: 8

Element 4: 3

Modified elements :- 1000

Length: 4

Is vector empty? false

[2, 3, 8, 1000]

[]

* Pop and remove in Vector
* **pop**: When you want to remove the last element of a vector, as it is faster (O (1)) and doesn't require an index.
* **remove**: When you need to remove an element at a specific position (by index) in the vector, but be aware that it will potentially require shifting elements (O(n)).

// Project30

use std::io;

fn main() {

    let mut input = String::new();

    println!("Enter the length of the vector:");

    io::stdin()

        .read\_line(&mut input)

        .expect("Failed to read input");

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

    let mut v: Vec<i32> = Vec::with\_capacity(length);

    for i in 0..length {

        input.clear();

        println!("Enter element {}: ", i + 1);

        io::stdin()

            .read\_line(&mut input)

            .expect("Failed to read input");

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

        v.push(element);

    }

    println!("The elements of the vector before deletion are: {:?}", v);

    v.pop();

    println!("The elements of the vector after poping are: {:?}", v);

    v.remove(0);

    println!("The elements of the vector after removing are: {:?}", v);

}

Output:- Enter the length of the vector:

5

Enter element 1:

1

Enter element 2:

2

Enter element 3:

3

Enter element 4:

4

Enter element 5:

5

The elements of the vector before deletion are: [1, 2, 3, 4, 5]

The elements of the vector after poping are: [1, 2, 3, 4]

The elements of the vector after removing are: [2, 3, 4]

* Reverse Vector

// Project31

fn main() {

    let mut v = vec![1, 2, 3, 4, 5];

    // Reversing the vector in place

    v.reverse();

    // Printing the reversed vector

    println!("{:?}", v); // Outputs: [5, 4, 3, 2, 1]

}

Output:-

[5, 4, 3, 2, 1]

* Slicing in Rust

A **slice** in Rust is a dynamically-sized view into a sequence of elements of an array, vector, or other data structure. Slices provide safe and efficient access to portions of a collection without copying the data.

* Declaring a Slice with Array and Vector
* Length of a Slice
* Access Elements in a Slice

// Project32

fn main() {

    let arr = [10, 20, 30, 40, 50];

    // Slice of the array

    let slice1 = &arr[1..4];

    println!("{:?}", slice);

    println!("Length: {}", slice1.len());

    println!("Accessing first element of slice :- {}", slice1[0]);

    let vec = vec![1, 2, 3, 4, 5];

    // Slice of the vector

    let slice2 = &vec[0..3];

    println!("{:?}", slice);

    println!("Length: {}", slice2.len());

}

Output:-

[20, 30, 40]

Length: 3

Accessing first element of slice :- 20

[1, 2, 3]

Length: 3

* Mutable Slices

You can create a mutable slice to modify elements in the original collection.

fn main() {

    let mut arr = [1, 2, 3, 4, 5];

    {

        let slice = &mut arr[1..4];

        slice[0] = 20; // Modifies arr[1]

        slice[1] = 30; // Modifies arr[2]

    }

    println!("{:?}", arr); // Outputs: [1, 20, 30, 4, 5]

}

Output:-

[1, 20, 30, 4, 5]

# #️ String

* Creating a String from a Literal
* Concatenation of String
* Iterate over characters
* Length of String
* Appending a character to a String
* Appending a String to String
* Slicing Substring
* Check if String is empty or not
* Check if String Contains a Substring
* Finding the index of substring in String
* Replacing parts of a String
* Trim WhiteSpace
* Convert to Uppercase or Lowercase
* Check whether two string is equal or not

// Project34

fn main() {

    // Creating a String from a Literal

    let s1 = String::from("Hello, ");

    let s2 = String::from("world!");

    // Concatenation of String

    let s3 = s1 + &s2; // `s1` is moved, so it's no longer accessible

    println!("{}", s3);

    // Iterate over characters

    let mut s3 = String::from("Hello");

    for c in s3.chars() {

        println!("{}", c); // Prints each character

    }

    // Length of String

    let len: usize = s3.len();

    println!("Length of String :- {}", len);

    // Appending a character to a String

    s3.push('!');

    println!("Appended character :- {}", s3);

    // Appending a String to String

    s3.push\_str(" World");

    println!("Appended String :- {}", s3);

    // Slicing Substring

    let slice = &s3[0..5]; // Includes indices 0 to 4

    println!("Slicing Substring :- {}", slice);

    // Check if String is empty or not

    println!(

        "Checking whether string is empty or not :- {}",

        s3.is\_empty()

    );

    // Check if String Contains a Substring

    println!("Checking for substring :- {}", s3.contains("Hello"));

    // Finding the index of subString in String

    let s4 = String::from("Hello, world!");

    if let Some(index) = s4.find("world") {

        println!("Found at index: {}", index);

    }

    // Replacing parts of a String

    let s5 = String::from("Hello, world!");

    let replaced = s5.replace("world", "Rust");

    println!("{}", replaced);

    // Trim WhiteSpace

    let s6 = String::from("   Hello, world!   ");

    println!("{}", s6.trim());

    // Convert to Uppercase or Lowercase

    let s7 = String::from("Hello");

    println!("{}", s7.to\_uppercase()); // Outputs: HELLO

    println!("{}", s7.to\_lowercase()); // Outputs: hello

}

Output:-

Hello, world!

H

e

l

l

o

Length of String :- 5

Appended character :- Hello!

Appended String :- Hello! World

Slicing Substring :- Hello

Checking whether string is empty or not :- false

Checking for substring :- true

Found at index: 7

Hello, Rust!

Hello, world!

HELLO

hello

* Input on String

// Project35

use std::io;

fn main() {

    // Create a mutable string to store the input

    let mut input = String::new();

    // Print a prompt for the user

    println!("Enter a string:");

    // Read the input from the user

    io::stdin()

        .read\_line(&mut input) // Append the input into the `input` variable

        .expect("Failed to read input");

    // Remove the newline character

    let input = input.trim().to\_string();

    // Print the input

    println!("You entered: {}", input);

}

Output:-

Enter a string:

hello dev!

You entered: hello dev!

# # Functions in Rust

Functions in Rust are a fundamental building block for organizing and reusing code. Here’s a detailed explanation of all features of functions in Rust.

Rust code uses *snake case* as the conventional style for function and variable names, in which all letters are lowercase and underscores separate words.

We define a function in Rust by entering fn followed by a function name and a set of parentheses. The curly brackets tell the compiler where the function body begins and ends.

1. Defining and Calling a Function in Rust

// Project36

fn greet() {

    println!("Hello, Rust!");

}

fn main() {

    greet();

}

Output:-

Hello, Rust!

1. Function Parameters and Return values

// Project37

fn greet(name: &str) {

    println!("Hello, {}!", name);

}

fn add(a: i32, b: i32) -> i32 {

    a + b

}

fn square(num: i32) -> i32 {

    num \* num // No semicolon means this is the return value

}

fn cube(num: i32) -> i32 {

    return num \* num \* num;

}

fn main() {

    greet("Harsh");

    let result = add(5, 7);

    println!("The sum is: {}", result);

    let squareresult = square(4);

    println!("The square is: {}", squareresult);

    let cuberesult = cube(2);

    println!("The cube is: {}", cuberesult);

}

Output:-

Hello, Harsh!

The sum is: 12

The square is: 16

The cube is: 8

1. Function Overloading

Rust does not support function overloading (multiple functions with the same name but different parameters). However, you can use traits or enums to achieve similar behavior.

1. Inline Functions

Use the #[inline] attribute to hint the compiler to inline a function, potentially improving performance.

// Project38

#[inline]

fn add(a: i32, b: i32) -> i32 {

    a + b

}

fn main() {

    let result = add(15,16);

    println!("The result is: {}", result);

}

Output:-

The result is: 31

1. Generic Functions

In Rust, Generic functions are very useful. Generic make code more flexible and provide more functionality to the callers of the function. It prevents code duplication as it is not required to define different functions of different types. Generics are specified in the signature of function where we actually specify the datatype of parameters and return type.

Rust allows you to write generic functions that work with multiple types.

// Project39

use std::fmt::Debug;

fn print\_value<T: Debug>(value: T) {

    println!("{:?}", value);

}

fn main() {

    print\_value(42);        // Works with integers

    print\_value("Rust!");   // Works with strings

}

Output:-

42

"Rust!"

1. Function as First-Class Citizens

Functions in Rust can be assigned to variables, passed as arguments, and returned from other functions.

* Assign to variables:

fn add(a: i32, b: i32) -> i32 {

    a + b

}

fn main() {

    let operation = add;

    println!("{}", operation(2, 3)); // Outputs: 5

}

Output:-

5

* Pass as Arguments:

fn operate(a: i32, b: i32, func: fn(i32, i32) -> i32) -> i32 {

    func(a, b)

}

fn multiply(a: i32, b: i32) -> i32 {

    a \* b

}

fn main() {

    let result = operate(3, 4, multiply);

    println!("{}", result); // Outputs: 12

}

Output:-

12

1. Recursion

// Project42

fn factorial(n: u32) -> u32 {

    if n == 0 {

        1

    } else {

        n \* factorial(n - 1)

    }

}

fn main() {

    println!("{}", factorial(5));

}

Output:-

120

1. Nested Functions

Functions can be defined within other functions for better encapsulation.

// Project43

fn main() {

    fn inner\_function() {

        println!("Inner function");

    }

    inner\_function();

}

1. Function Pointers

You can define a function pointer type for referencing functions.

fn add(a: i32, b: i32) -> i32 {

    a + b

}

fn main() {

    let func: fn(i32, i32) -> i32 = add;

    println!("{}", func(3, 4));

}

Output:-

7