# **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.

// Project 44

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

    a + b

}

fn main() {

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

    println!("{}", func(3, 4));

}

Output:-

7

# # Ownership

Ownership is a set of rules that govern how a Rust program manages memory. All programs have to manage the way they use a computer’s memory while running. Some languages have garbage collection that regularly looks for no-longer-used memory as the program runs; in other languages, the programmer must explicitly allocate and free the memory. Rust uses a third approach: memory is managed through a system of ownership with a set of rules that the compiler checks. If any of the rules are violated, the program won’t compile. None of the features of ownership will slow down your program while it’s running.

Because ownership is a new concept for many programmers, it does take some time to get used to. The good news is that the more experienced you become with Rust and the rules of the ownership system, the easier you’ll find it to naturally develop code that is safe and efficient. Keep at it!

When you understand ownership, you’ll have a solid foundation for understanding the features that make Rust unique.

* The Stack and the Heap

Both the stack and the heap are parts of memory available to your code to use at runtime, but they are structured in different ways. The stack stores values in the order it gets them and removes the values in the opposite order. This is referred to as *last in, first out*. Think of a stack of plates: when you add more plates, you put them on top of the pile, and when you need a plate, you take one off the top. Adding or removing plates from the middle or bottom wouldn’t work as well! Adding data is called *pushing onto the stack*, and removing data is called *popping off the stack*. All data stored on the stack must have a known, fixed size. Data with an unknown size at compile time or a size that might change must be stored on the heap instead.

The heap is less organized: when you put data on the heap, you request a certain amount of space. The memory allocator finds an empty spot in the heap that is big enough, marks it as being in use, and returns a *pointer*, which is the address of that location. This process is called *allocating on the heap* and is sometimes abbreviated as just *allocating* (pushing values onto the stack is not considered allocating). Because the pointer to the heap is a known, fixed size, you can store the pointer on the stack, but when you want the actual data, you must follow the pointer. Think of being seated at a restaurant. When you enter, you state the number of people in your group, and the host finds an empty table that fits everyone and leads you there. If someone in your group comes late, they can ask where you’ve been seated to find you.

Pushing to the stack is faster than allocating on the heap because the allocator never has to search for a place to store new data; that location is always at the top of the stack. Comparatively, allocating space on the heap requires more work because the allocator must first find a big enough space to hold the data and then perform bookkeeping to prepare for the next allocation.

Accessing data in the heap is slower than accessing data on the stack because you have to follow a pointer to get there. Contemporary processors are faster if they jump around less in memory.

**Ownership Rules**

First, let’s take a look at the ownership rules. Keep these rules in mind as we work through the examples that illustrate them:

* Each value in Rust has an *owner*.
* There can only be one owner at a time.
* When the owner goes out of scope, the value will be dropped.

**Variable Scope**

As a first example of ownership, we’ll look at the *scope* of some variables. A scope is the range within a program for which an item is valid.

Let’s look at this example

// Project45

fn main() {

    {                          // s is not valid here

        let s: String = String::from("hello");       // s comes into scope here

        println!("{}", s);     // s can be used within this scope

    }                          // s goes out of scope here and is no longer valid

    // println!("{}", s);     // This would cause a compile-time error because s is out of scope

}

Output:-

hello

The variable s refers to a string literal, where the value of the string is hardcoded into the text of our program. The variable is valid from the point at which it’s declared until the end of the current *scope*.

In other words, there are two important points in time here:

* When s comes *into* scope, it is valid.
* It remains valid until it goes *out of* scope.

At this point, the relationship between scopes and when variables are valid is similar to that in other programming languages.

**Memory and Allocation**

In Rust, managing memory is a key feature that makes the language both **safe** and **efficient**. It does this without requiring a garbage collector (GC) like languages such as Java or Python. Instead, Rust uses **ownership rules** and an automatic process to allocate and deallocate memory.

**String Literals vs. String Type**

1. **String Literals**:
   * Known at compile time.
   * Hardcoded into the final executable.
   * Stored in the **stack** (a fast, limited area of memory).
   * Immutable (cannot be changed after creation).
2. **String Type**:
   * Used for mutable, growable text.
   * Size is not known at compile time.
   * Allocates memory on the **heap** (a slower but larger area of memory).
   * Requires runtime memory management.

**Memory Management in Rust**

1. **Heap Allocation**:
   * When we use String::from("hello"), Rust requests memory on the heap to store the string data.
2. **Automatic Deallocation**:
   * When a variable goes **out of scope**, Rust automatically calls the drop function to free the allocated memory.
   * This avoids common memory bugs like **double free**, **dangling pointers**, and **memory leaks**.

**Key Concept: Ownership**

* A variable **owns** its memory, and only one variable can own a given piece of memory at a time.
* When the owner goes out of scope, Rust cleans up the memory automatically.

**Variables and Data Interacting with Move**

Multiple variables can interact with the same data in different ways in Rust. Let’s look at an example using an integer.

// Project46

fn main() {

    let x = 5;

    let y = x;                       // x is copied because integers implement the Copy trait

    println!("x: {}, y: {}", x, y);  // Both x and y are valid

}

Output:-

x: 5, y: 5

We can probably guess what this is doing: “bind the value 5 to x; then make a copy of the value in x and bind it to y.” We now have two variables, x and y, and both equal 5. This is indeed what is happening, because integers are simple values with a known, fixed size, and these two 5 values are pushed onto the stack.

**Copy for Stack-Only Data**:

* Simple data types like integers implement the Copy trait.
* Assignments like let y = x; create a direct copy of the value, and both variables remain valid.

In Rust, when assigning variables of types that allocate memory on the **heap**, such as String, the data is **moved** rather than **copied** to ensure memory safety. This behavior prevents common issues like **double freeing** memory or dangling pointers.

**Key Concepts**

1. **Move**:
   * When a variable is moved, ownership of the heap data is transferred to the new variable.
   * The original variable becomes invalid and cannot be used further.
   * Rust enforces this at compile time, preventing memory errors.
2. **Shallow Copy vs. Deep Copy**:
   * A **shallow copy** only copies the pointer, length, and capacity of a variable, not the actual data on the heap.
   * Rust performs a move instead of a shallow copy by invalidating the original variable after the assignment.
   * **Deep copies**, which involve duplicating the heap data, must be explicitly requested using .clone().
3. **Performance**:
   * Moves are efficient because they avoid copying potentially large amounts of data.
   * Explicit deep copies are performed only when necessary, giving you control over performance costs.

// Project47

fn main() {

    // Example 1: Move semantics with String

    let s1 = String::from("hello"); // s1 owns the String

    let s2 = s1; // Ownership is moved to s2; s1 is now invalid

    // println!("{}", s1);          // This would cause a compile-time error: "value borrowed here after move"

    println!("{}", s2); // This works because s2 owns the String

    // Example 2: Explicitly creating a deep copy

    let s3 = String::from("world"); // s3 owns another String

    let s4 = s3.clone(); // Deep copy of the String; both s3 and s4 are valid

    println!("s3: {}, s4: {}", s3, s4); // Both s3 and s4 can be used independently

}

Output:-

hello

s3: world, s4: world

**Ownership and Functions**

In Rust, passing variables to functions follows the same rules as assigning variables:

1. **Move**:
   * If you pass a variable to a function and the variable owns data on the heap (like a String), the ownership is transferred to the function parameter.
   * After the transfer, the original variable is no longer valid, and trying to use it will result in a compile-time error.
2. **Copy**:
   * If the variable implements the Copy trait (e.g., i32), the value is copied into the function parameter.
   * The original variable remains valid even after the function call.

This behavior ensures memory safety by clearly defining ownership and avoiding issues like dangling pointers or double frees.

// Project 48

fn main() {

    let s = String::from("hello");  // `s` owns the string "hello"

    takes\_ownership(s);             // Ownership of `s` moves to the function

    // println!("{s}");             // Error: `s` is no longer valid here

    let x = 42;                     // `x` owns the integer 42

    makes\_copy(x);                  // `x` is copied into the function

    println!("{x}");                // Valid: `x` is still accessible because it implements Copy

    let s1 = String::from("world");

    let s2 = return\_ownership(s1);  // Ownership of `s1` moves to the function,

                                    // and the function returns ownership to `s2`

    println!("{s2}");               // Valid: `s2` now owns the String

}

fn takes\_ownership(some\_string: String) {

    // This function takes ownership of the String passed to it

    println!("Owned string: {}", some\_string);

} // `some\_string` goes out of scope, and its memory is freed here.

fn makes\_copy(some\_integer: i32) {

    // This function receives a copy of the integer

    println!("Copied integer: {}", some\_integer);

} // No special action required; the integer is stack-allocated.

fn return\_ownership(some\_string: String) -> String {

    // This function returns ownership of the String

    some\_string

}

Output :-

Owned string: hello

Copied integer: 42

42

world

One more Example :-

// Project 49

fn main() {

    let s1 = String::from("hello");

    let (s2, len) = calculate\_length(s1);

    println!("The length of '{s2}' is {len}.");

}

fn calculate\_length(s: String) -> (String, usize) {

    let length = s.len(); // len() returns the length of a String

    (s, length)

}

Output:-

The length of 'hello' is 5.

# # References and Borrowing

In Rust, a reference is a way to access data without taking ownership of it. This allows the original owner to retain control over the data while allowing others to read (or sometimes modify) it temporarily.

// Project50

fn main() {

    let s1 = String::from("hello");

    let len = calculate\_length(&s1); // Borrowing s1

    println!("The length of '{}' is {}.", s1, len);

}

fn calculate\_length(s: &String) -> usize {

    s.len() // Using the borrowed reference

}

Output :-

The length of 'hello' is 5.

**Explanation:**

1. **Borrowing: The &s1 syntax creates a reference to the string s1 without transferring ownership.**
2. **Function Signature: The function calculate\_length accepts a reference &String as its parameter. This indicates that the function will borrow the string rather than take ownership.**
3. **No Ownership Transfer: After the function call, s1 remains valid because its ownership was never transferred.**

**Immutable and Mutable References**

By default, references are **immutable**, meaning you cannot modify the data through the reference. If you want to modify the data, you need to use a **mutable reference**.

Code Example: Mutable References

// Project51

fn main() {

    let mut s = String::from("hello");

    change(&mut s); // Passing a mutable reference

    println!("{}", s); // Prints "hello, world"

}

fn change(some\_string: &mut String) {

    some\_string.push\_str(", world"); // Modifying the borrowed data

}

Output:-

hello, world

**Explanation:**

1. **Mutable Borrow:** The &mut s syntax allows the change function to modify s.
2. **Function Signature:** The parameter some\_string: &mut String indicates that the function expects a mutable reference.
3. **Ownership Rules:** Ownership remains with the original variable (s), but the function can modify its contents.

**Restriction: Single Mutable Reference**

Rust enforces the rule that you cannot have more than one mutable reference to a value at a time. This prevents data races and ensures safe concurrency.

Code Example: Invalid Multiple Mutable References

fn main() {

    let mut s = String::from("hello");

    let r1 = &mut s;

    let r2 = &mut s; // Error: Cannot borrow `s` as mutable more than once

    println!("{}, {}", r1, r2);

}

Error:-

second mutable borrow occurs here

**Explanation:**

* The compiler generates an error because r1 and r2 both try to borrow s mutably.
* Rust prevents multiple mutable references to ensure data consistency and avoid race conditions.

Code Example: Using Scopes to Resolve Multiple Mutable References

// Project52

fn main() {

    let mut s = String::from("hello");

    {

        let r1 = &mut s; // First mutable reference

        println!("{}", r1);

    } // r1 goes out of scope here

    let r2 = &mut s; // Second mutable reference

    println!("{}", r2);

}

Output:-

hello

hello

**Explanation:**

* By enclosing r1 in its own scope, we ensure it is dropped before r2 is created.
* This resolves the issue of overlapping mutable references.

**Combining Immutable and Mutable References**

Rust does not allow a mutable reference to coexist with immutable references. This ensures data integrity when multiple references are used.

Code Example: Invalid Combination

fn main() {

    let mut s = String::from("hello");

    let r1 = &s; // Immutable reference

    let r2 = &s; // Another immutable reference

    let r3 = &mut s; // Error: Cannot borrow `s` as mutable

    println!("{}, {}, {}", r1, r2, r3);

}

Error:-

mutable borrow occurs here

**Explanation:**

* The immutable references (r1 and r2) conflict with the mutable reference (r3).
* This restriction ensures that data being read (via immutable references) is not modified simultaneously (via a mutable reference).

Code Example: Valid Combination by Limiting Scope

// Project53

fn main() {

    let mut s = String::from("hello");

    let r1 = &s;

    let r2 = &s;

    println!("{} and {}", r1, r2); // Last use of r1 and r2

    let r3 = &mut s; // No conflict as r1 and r2 are no longer used

    println!("{}", r3);

}

Output:-

hello and hello

hello

**Dangling References**

Rust ensures that references are always valid. A common bug in other languages is creating a reference to memory that has been freed, called a dangling reference. Rust prevents this at compile time.

Code Example: Dangling Reference

fn main() {

    let reference\_to\_nothing = dangle();

}

fn dangle() -> &String {

    let s = String::from("hello"); // `s` is created

    &s // Returning a reference to `s`

} // `s` goes out of scope and is dropped here

Error:-

cannot return reference to local variable `s`

--> src/main.rs:19:5

|

19 | &s // Returning a reference to `s`

| ^^ returns a reference to data owned by the current function

Solution: Return Ownership

// Project54

fn main() {

    let no\_dangle = no\_dangle();

    println!("no\_dangle: {}", no\_dangle);

}

fn no\_dangle() -> String {

    let s = String::from("hello");

    s // Ownership is moved to the caller

}

Output:-

no\_dangle: hello

# # Structs

Structs in Rust allow you to group related data together into a single compound type. Each field in a struct is named and has a specific type. Structs are useful for creating more expressive types compared to tuples, as they associate names with data.

Defining a Struct

struct User {

    active: bool,

    username: String,

    email: String,

    sign\_in\_count: u64,

}

Here:

* struct User defines a blueprint for a User.
* It contains four fields:
  + active of type bool.
  + username of type String.
  + email of type String.
  + sign\_in\_count of type u64.

Instantiating a Struct

To create an instance of a struct, provide values for each field using **key-value pairs**.

// Project 55

struct User {

    active: bool,

    username: String,

    email: String,

    sign\_in\_count: u64,

}

fn main() {

    let user1 = User {

        active: true,

        username: String::from("someusername123"),

        email: String::from("someone@example.com"),

        sign\_in\_count: 1,

    };

    println!("Username: {}", user1.username);

}

Output:-

Username: someusername123

Mutability

If you need to modify a field, the entire instance must be mutable.

// Project56

struct User {

    active: bool,

    username: String,

    email: String,

    sign\_in\_count: u64,

}

fn main() {

    let mut user1 = User {

        active: true,

        username: String::from("someusername123"),

        email: String::from("someone@example.com"),

        sign\_in\_count: 1,

    };

    user1.email = String::from("newemail@example.com");

    println!("Updated Email: {}", user1.email);

}

Output:-

Updated Email: newemail@example.com

Using Functions for Struct Initialization

When initializing structs repeatedly, you can use functions for cleaner code.

// Project57

struct User {

    active: bool,

    username: String,

    email: String,

    sign\_in\_count: u64,

}

fn build\_user(email: String, username: String) -> User {

    User {

        active: true,

        username,

        email,

        sign\_in\_count: 1,

    }

}

fn main() {

    let user1 = build\_user(String::from("someone@example.com"), String::from("user123"));

    println!("User email: {}", user1.email);

}

Output:-

User email: someone@example.com

Struct Update Syntax

Rust allows you to create new instances based on existing ones, using the **struct update syntax**.

// Project58

struct User {

    active: bool,

    username: String,

    email: String,

    sign\_in\_count: u64,

}

fn main() {

    let user1 = User {

        active: true,

        username: String::from("user1"),

        email: String::from("user1@example.com"),

        sign\_in\_count: 1,

    };

    let user2 = User {

        email: String::from("user2@example.com"),

        ..user1

    };

    println!("User2 email: {}", user2.email);

    println!("User2 username: {}", user2.username);

}

Output:-

User2 email: user2@example.com

User2 username: user1

Tuple Structs

Tuple structs give a name to a tuple without named fields.

// Project59

struct Color(i32, i32, i32);

struct Point(i32, i32, i32);

fn main() {

    let black = Color(0, 0, 0);

    let origin = Point(0, 0, 0);

    println!("Black: ({}, {}, {})", black.0, black.1, black.2);

}

Output:-

Black: (0, 0, 0)

Unit-Like Structs

Unit-like structs have no fields and are useful for implementing traits without storing data.

// Project60

struct AlwaysEqual;

fn main() {

    let instance = AlwaysEqual;

}

Ownership in Structs

When defining structs, you usually use owned types (like String) rather than borrowed types (like &str). This ensures that the data lives as long as the struct.

Example: Struct with String Fields

// Project61

struct User {

    username: String,

    email: String,

}

fn main() {

    let user = User {

        username: String::from("user123"),

        email: String::from("user@example.com"),

    };

    println!("Username: {}", user.username);

}

Output:-

Username: user123

Example: Struct with References (Requires Lifetimes)

To store references, you need **lifetime specifiers** to ensure that the referenced data outlives the struct.

// Project62

struct User<'a> {

    username: &'a str,

    email: &'a str,

}

fn main() {

    let username = "user123";

    let email = "user@example.com";

    let user = User {

        username,

        email,

    };

    println!("Username: {}", user.username);

}

Output:-

Username: user123

**An Example Program Using Structs**

Step 1: Basic Implementation Using Individual Variables

Here, we calculate the area of a rectangle using individual variables for width and height. This approach works but lacks clarity and structure.

// Project63

fn main() {

    let width1 = 30;

    let height1 = 50;

    println!(

        "The area of the rectangle is {} square pixels.",

        area(width1, height1)

    );

}

fn area(width: u32, height: u32) -> u32 {

    width \* height

}

Output:-

The area of the rectangle is 1500 square pixels.

Step 2: Refactoring with Tuples

We improve clarity slightly by grouping width and height into a tuple.

// Project64

fn main() {

    let rect1 = (30, 50);

    println!(

        "The area of the rectangle is {} square pixels.",

        area(rect1)

    );

}

fn area(dimensions: (u32, u32)) -> u32 {

    dimensions.0 \* dimensions.1

}

Output:-

The area of the rectangle is 1500 square pixels.

Step 3: Refactoring with Structs

Now, we introduce structs for better clarity and structure. Structs allow us to assign meaningful names to the dimensions of the rectangle.

// Project65

struct Rectangle {

    width: u32,

    height: u32,

}

fn main() {

    let rect1 = Rectangle {

        width: 30,

        height: 50,

    };

    println!(

        "The area of the rectangle is {} square pixels.",

        area(&rect1)

    );

}

fn area(rectangle: &Rectangle) -> u32 {

    rectangle.width \* rectangle.height

}

Output:-

The area of the rectangle is 1500 square pixels.

Step 4: Debugging with the Debug Trait

To print a struct for debugging, we derive the Debug trait.

// Project66

#[derive(Debug)]

struct Rectangle {

    width: u32,

    height: u32,

}

fn main() {

    let rect1 = Rectangle {

        width: 30,

        height: 50,

    };

    println!("rect1 is {:?}", rect1);

    println!("rect1 (pretty) is {:#?}", rect1);

}

Output:-

rect1 is Rectangle { width: 30, height: 50 }

rect1 (pretty) is Rectangle {

width: 30,

height: 50,

}

**Explanation**

1. **Deriving Debug**:
   * #[derive(Debug)] adds the ability to print structs using {:?} or {:#?}.
2. **Debugging Output**:
   * {:?} gives a compact view: Rectangle { width: 30, height: 50 }.
   * {:#?} provides a prettier, indented view.

Step 5: Using the dbg! Macro

The dbg! macro is another debugging tool. It prints file and line information, along with the value of an expression.

// Project67

#[derive(Debug)]

struct Rectangle {

    width: u32,

    height: u32,

}

fn main() {

    let scale = 2;

    let rect1 = Rectangle {

        width: dbg!(30 \* scale),

        height: 50,

    };

    dbg!(&rect1);

}

Output:-

[src/main.rs:10:16] 30 \* scale = 60

[src/main.rs:14:5] &rect1 = Rectangle {

width: 60,

height: 50,

}

Step 6: Turning Functions into Methods

To tie the area function directly to the Rectangle struct, we use methods.

// Project68

#[derive(Debug)]

struct Rectangle {

    width: u32,

    height: u32,

}

impl Rectangle {

    fn area(&self) -> u32 {

        self.width \* self.height

    }

}

fn main() {

    let rect1 = Rectangle {

        width: 30,

        height: 50,

    };

    println!(

        "The area of the rectangle is {} square pixels.",

        rect1.area()

    );

    dbg!(&rect1);

}

Output:-

The area of the rectangle is 1500 square pixels.

[src/main.rs:24:5] &rect1 = Rectangle {

width: 30,

height: 50,

}

Step 7: **Combining Everything**

// Project69

#[derive(Debug)]

struct Rectangle {

    width: u32,

    height: u32,

}

impl Rectangle {

    // Method to calculate area

    fn area(&self) -> u32 {

        self.width \* self.height

    }

    // Method to check if one rectangle can hold another

    fn can\_hold(&self, other: &Rectangle) -> bool {

        self.width > other.width && self.height > other.height

    }

    // Associated function to create a square

    fn square(size: u32) -> Self {

        Self {

            width: size,

            height: size,

        }

    }

}

fn main() {

    // Creating instances of Rectangle

    let rect1 = Rectangle {

        width: 30,

        height: 50,

    };

    let rect2 = Rectangle {

        width: 10,

        height: 40,

    };

    let rect3 = Rectangle {

        width: 60,

        height: 45,

    };

    // Displaying the area of rect1

    println!(

        "The area of the rectangle is {} square pixels.",

        rect1.area()

    );

    // Checking if rect1 can hold rect2 and rect3

    println!("Can rect1 hold rect2? {}", rect1.can\_hold(&rect2));

    println!("Can rect1 hold rect3? {}", rect1.can\_hold(&rect3));

    // Creating a square using the associated function

    let square = Rectangle::square(20);

    println!("Square: {:?}", square);

}

Output:-

The area of the rectangle is 1500 square pixels.

Can rect1 hold rect2? true

Can rect1 hold rect3? false

Square: Rectangle { width: 20, height: 20 }

# # Enums

Enums in Rust are used to define a type that can have different variants, and each variant can hold different kinds of data. This is helpful when we want to represent multiple possible options but we want them to be logically tied together under a single type.

// Project70

// Define an enum for IP addresses

enum IpAddr {

    V4(u8, u8, u8, u8),

    V6(String),

}

struct IpAddrInfo {

    kind: IpAddr,

    address: String,

}

impl IpAddrInfo {

    fn display(&self) {

        match &self.kind {

            IpAddr::V4(a, b, c, d) => {

                println!("IPv4 Address: {}.{}.{}.{}", a, b, c, d);

            },

            IpAddr::V6(addr) => {

                println!("IPv6 Address: {}", addr);

            },

        }

    }

}

enum Message {

    Quit,

    Move { x: i32, y: i32 },

    Write(String),

    ChangeColor(i32, i32, i32),

}

impl Message {

    fn call(&self) {

        match self {

            Message::Quit => println!("Quit message received"),

            Message::Move { x, y } => println!("Move to ({}, {})", x, y),

            Message::Write(s) => println!("Write: {}", s),

            Message::ChangeColor(r, g, b) => println!("Change color to RGB({}, {}, {})", r, g, b),

        }

    }

}

fn main() {

    let home = IpAddrInfo {

        kind: IpAddr::V4(127, 0, 0, 1),

        address: String::from("127.0.0.1"),

    };

    let loopback = IpAddrInfo {

        kind: IpAddr::V6(String::from("::1")),

        address: String::from("::1"),

    };

    home.display();

    loopback.display();

    let msg = Message::Write(String::from("Hello, world!"));

    msg.call();

    let some\_number = Some(5);

    match some\_number {

        Some(value) => println!("The value is {}", value),

        None => println!("No value"),

    }

}

Output:-

IPv4 Address: 127.0.0.1

IPv6 Address: ::1

Write: Hello, world!

The value is 5

**Conclusion**

* **Enums** in Rust help group related data together and allow you to store data in various ways within each variant.
* You can define **methods** on enums, similar to structs, to work with their data.
* **Option<T>** is a special enum for handling optional values, eliminating the need for null pointers and reducing errors in your code.

# # Generic Types, Traits, and Lifetimes

1. **Generic Data Types**

Generic Functions

Generics in Rust enable developers to write more flexible and reusable code without sacrificing performance. Instead of writing functions, structs, enums, and methods for each possible type, you can define them generically and specify the exact type at the time of calling.

Generics allow functions to accept arguments of any type. In the example you gave, two functions largest\_i32 and largest\_char were defined separately for finding the largest number and largest character in a slice, respectively. We can consolidate these two functions into a single generic function.

The two functions largest\_i32 and largest\_char essentially perform the same task, differing only in the type of the input (i32 vs. char). This results in code duplication, which we can avoid by using generics.

// Project71

use std::cmp::PartialOrd;

fn largest<T: PartialOrd>(list: &[T]) -> &T {

    let mut largest = &list[0];

    for item in list {

        if item > largest {

            largest = item;

        }

    }

    largest

}

fn main() {

    let number\_list = vec![34, 50, 25, 100, 65];

    let result = largest(&number\_list);

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

    let char\_list = vec!['y', 'm', 'a', 'q'];

    let result = largest(&char\_list);

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

}

Output:-

The largest number is 100

The largest char is y

Generic Structs

Generics can also be used in structs, allowing you to define a struct that can hold values of different types.

// Project72

struct Point<T> {

    x: T,

    y: T,

}

fn main() {

    let integer\_point = Point { x: 5, y: 10 };

    let float\_point = Point { x: 1.0, y: 4.0 };

    println!("Integer point: ({}, {})", integer\_point.x, integer\_point.y);

    println!("Float point: ({}, {})", float\_point.x, float\_point.y);

}

Output:-

Integer point: (5, 10)

Float point: (1, 4)

Handling Different Types for x and y

// Project73

struct Point<T, U> {

    x: T,

    y: U,

}

fn main() {

    let point1 = Point { x: 5, y: 10.5 }; // x is i32, y is f64

    let point2 = Point { x: "Hello", y: 'c' }; // x is &str, y is char

    println!("Point1: ({}, {})", point1.x, point1.y);

    println!("Point2: ({}, {})", point2.x, point2.y);

}

Output:-

Point1: (5, 10.5)

Point2: (Hello, c)

Generic Enums

You can use generics in enums as well. A good example is the Option<T> enum, which can hold an optional value of any type:

// Projecct74

enum Option<T> {

    Some(T),

    None,

}

fn main() {

    let integer\_option = Option::Some(5);

    let string\_option = Option::Some("Hello");

    match integer\_option {

        Option::Some(x) => println!("Integer value: {}", x),

        Option::None => println!("No value"),

    }

    match string\_option {

        Option::Some(x) => println!("String value: {}", x),

        Option::None => println!("No value"),

    }

}

Output:-

Integer value: 5

String value: Hello

Generic Methods on Structs

You can define methods on structs that also use generics. In the case of the Point<T> struct, you might want a method that computes the distance from the origin:

// Project75

struct Point<T> {

    x: T,

    y: T,

}

impl<T> Point<T> {

    fn x(&self) -> &T {

        &self.x

    }

    fn y(&self) -> &T {

        &self.y

    }

}

impl Point<f32> {

    fn distance\_from\_origin(&self) -> f32 {

        (self.x.powi(2) + self.y.powi(2)).sqrt()

    }

}

fn main() {

    let point = Point { x: 3.0, y: 4.0 };

    println!("x: {}", point.x());

    println!("y: {}", point.y());

    println!("Distance from origin: {}", point.distance\_from\_origin());

}

Output:-

x: 3

y: 4

Distance from origin: 5

1. **Traits**

In Rust, **traits** are a powerful way to define shared behavior across types, much like interfaces in other languages, though with some differences. Traits allow you to abstract common functionality and ensure that multiple types adhere to a set of behaviors, which is important when working with generics.

Defining a Trait

//Project76

trait Overview {

    fn overview(&self) -> String {

        String::from("This is default implementation for trait")

    }

}

struct Course {

    headline: String,

    author: String,

}

struct AnotherCourse {

    headline: String,

    author: String,

}

impl Overview for Course {

    fn overview(&self) -> String {

        format!("{} by {}", self.headline, self.author)

    }

}

impl Overview for AnotherCourse {}

fn main() {

    let course1 = Course {

        headline: String::from("Headline!"),

        author: String::from("Harsh"),

    };

    let course2 = AnotherCourse {

        headline: String::from("AnotherHeadline!"),

        author: String::from("AnotherHarsh"),

    };

    println!("{}", course1.overview());

    println!("{}", course2.overview());

}

Output:-

Headline! by Harsh

This is default implementation for trait

# # Modules

See project77

$ cargo new Project77 –lib

$ cd Project77

$ cargo install cargo-modules

To see structure

$ cargo modules structure

# # Pointers

In Rust, **pointers** are used to reference memory locations. They allow us to handle data indirectly, which is essential for managing ownership, borrowing, and sharing data across scopes. Let’s delve into various types of pointers and their features.

**1. Pointers in Rust**

Rust's pointers include:

* **References (& and &mut)**: Immutable and mutable references.
* **Raw pointers (\*const T and \*mut T)**: Unsafe, for low-level memory operations.
* **Smart pointers**: Data structures like Box, Rc, Arc, and RefCell.

**2. Box Pointer (Heap Allocation)**

Box<T> is a smart pointer for heap allocation. It is used when the size of the value is unknown or large. It provides ownership of the value it points to and allows accessing it through dereferencing.

**Key Features:**

* Single ownership.
* Allocates data on the heap.

**Code Example:**

fn main() {

let x = Box::new(5); // Allocates 5 on the heap

println!("Value of x: {}", \*x); // Dereferencing the box

}

**Explanation:**

* Box::new(5) allocates 5 on the heap.
* \*x dereferences the Box to access the underlying value.

**3. Dereferencing**

Dereferencing is the process of accessing the value pointed to by a pointer. Rust uses the \* operator for dereferencing.

**Code Example:**

fn main() {

let y = 10;

let z = &y; // Reference to y

println!("z points to: {}", \*z); // Dereferencing z to access y

}

**Explanation:**

* &y creates a reference.
* \*z dereferences the reference to access y.

**4. Reference Counting (Rc)**

Rc<T> is a smart pointer that enables multiple ownership of the same data. It keeps a reference count, and the data is dropped only when the count reaches zero.

**Key Features:**

* Used for shared ownership in single-threaded scenarios.
* Immutable access to the data.

**Code Example:**

use std::rc::Rc;

fn main() {

let a = Rc::new(5);

let b = Rc::clone(&a); // Creates another pointer to the same data

let c = Rc::clone(&a);

println!("a: {}, b: {}, c: {}", \*a, \*b, \*c);

println!("Reference count: {}", Rc::strong\_count(&a)); // Count of references

}

**Explanation:**

* Rc::clone creates new references.
* Rc::strong\_count tracks the number of active references.

**5. Atomic Reference Counting (Arc)**

Arc<T> is like Rc<T> but is thread-safe, allowing shared ownership across multiple threads. It uses atomic operations to ensure safety.

**Key Features:**

* Used for shared ownership in multi-threaded contexts.
* Immutable access to the data.

**Code Example:**

use std::sync::Arc;

use std::thread;

fn main() {

let value = Arc::new(10);

let handles: Vec<\_> = (0..3).map(|\_| {

let value = Arc::clone(&value);

thread::spawn(move || {

println!("Value: {}", \*value);

})

}).collect();

for handle in handles {

handle.join().unwrap();

}

}

**Explanation:**

* Arc::clone creates references for threads.
* Safe to share Arc across threads.

**6. Interior Mutability (RefCell)**

RefCell<T> allows mutable access to data even when it’s borrowed as immutable. It enforces borrowing rules at runtime, unlike compile-time checks for &mut.

**Key Features:**

* Single-threaded interior mutability.
* Borrowing rules enforced at runtime (panics if violated).

**Code Example:**

use std::cell::RefCell;

fn main() {

let data = RefCell::new(5);

{

let mut borrowed\_data = data.borrow\_mut(); // Mutable borrow

\*borrowed\_data += 1;

} // Mutable borrow ends here

println!("Updated value: {}", data.borrow()); // Immutable borrow

}

**Explanation:**

* RefCell::borrow\_mut allows mutable access.
* RefCell::borrow allows immutable access.
* Panics if you try to borrow mutably and immutably at the same time.

**7. Combining Rc and RefCell**

We can combine Rc<T> and RefCell<T> to achieve shared ownership and interior mutability.

**Code Example:**

use std::cell::RefCell;

use std::rc::Rc;

fn main() {

let data = Rc::new(RefCell::new(5));

let data\_clone = Rc::clone(&data);

\*data\_clone.borrow\_mut() += 1;

println!("Updated value: {}", \*data.borrow());

}

**Explanation:**

* Rc enables shared ownership.
* RefCell enables mutation even when Rc is used.

**8. Comparison Table**

| **Feature** | **Box<T>** | **Rc<T>** | **Arc<T>** | **RefCell<T>** |
| --- | --- | --- | --- | --- |
| Ownership | Single | Shared | Shared | Single |
| Thread Safety | No | No | Yes | No |
| Mutability | Compile-time | Immutable only | Immutable only | Runtime check |
| Use Case | Heap storage | Shared in single thread | Shared across threads | Interior mutability |

These tools provide flexibility for managing ownership and data in Rust while enforcing safety and performance. Choose based on your needs: single ownership (Box), shared ownership (Rc, Arc), or interior mutability (RefCell).