

Rust is a System Programming Language  
System Prog Lang is by which we access hardware very easily.

Rust is known for Memory Management, concurrency, speed

Rust generally follows SnakeCase (not CamelCase)

snake\_case  
JS - camelCase

Error in rust if camel case is used

Cargo. tool for dependencies  
Toml means Tom's obvious minimal language

Data types -

1. Scaler (integer, floating point )
2. Compound (array, tuple, string, vector)

Signed Integer	Unsigned Integer
i8	u8
i16	u16
i32	u32
i64	u64
isize	usize

- 3.
4. In general, every datatype in Rust is immutable
5. To make it mutable add a mut keyword while declaring

Function return type should be mentioned using the arrow

Memory management has 3 approach

1. Control first approach - used in c/cpp  
Ismein pointer point karta jaha memory allocate hui hai  
Agar humne free kiya variable, toh pointer dangling ho jaata
2. Safety first approach - used in Python/java  
Garbage collection hota hai - here we don't know how this would work therefore there is an issue in this approach
3. Ownership - used in rust

Ownership rules apply only on both but the main purpose is to manage heap data (dynamic memory allocated data type)

Here we do not require a garbage collector

Rules

1. Each value in Rust has a variable that's called its Owner.
2. There can only be one Owner at a time
3. When the Owner goes out of Scope, the value will be dropped

Scope?

The Area where we can access the variable is the scope of that variable.

Note - const - naming only in capital letters and its data type should be mentioned(mandatory)

In Rust, if two dynamic data-typed variables are pointing to the same address then it performs moving of address to a recently accessed variable bcoz

If two variables are pointing to the same location then if both try to free up the space then this would be a double pointer Error.

```
5 main.rs
fn main() {
    let x:String = String::from("Hello");//x is the owner of Hello
    process_string(x);//transfer of ownership
    // println!("The value of x in main() is {}",x);
}

fn process_string(item:String){//Hello-new owner is item
    println!("The value of x in process_string() is {}",item);
}
```

Error in main function's last line - due to transfer of ownership.

Example to understand ownership - here ownership of string is transferred

```
fn main() {
    let s1:String = String::from("hello");//s1 owner
    let (s2,len) = calculate_length(s1);//ownership transfer, new own
    println!("The length of {} is {}",s2,len);
}

fn calculate_length(s:String)->(String,usize){//s will be the new c
    let length:usize = s.len();
    return (s,length);//return ownership transfer,5
}
```

The same example without the ownership concept (passing clone as a parameter)

```
src > main.rs
1  fn main() {
2      let s1:String = String::from("hello");//s1 owner
3      let len = calculate_length(s1.clone());
4      println!("The length of {} is {}",s1,len);
5  }
6
7  fn calculate_length(s:String)->usize{
8      let length:usize = s.len();
9      return length;//5
10 }
```

OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

The length of hello is 5

PS C:\Users\user\hello\_program>

**BORROW OPERATION** - We just use string s1 by passing it by reference and using it  
This does not transfer the owner and our work is done.

```

main.rs
fn main() {
    let s1:String = String::from("Hello");
    let len:usize = calculate_length(&s1); //borrow operation
    println!("The length of {} is {}",s1,len);
}

fn calculate_length(s2:&String)->usize{
    return s2.len();
}

```

When borrow takes place Rust makes sure that no mutable operations take place on the variables/parameters

Therefore variable borrowed cant be mutable

## MUTABLE REFERENCE

```

fn main() {
    let mut s1:String = String::from("Hello");
    append_string(&mut s1);
    println!("The new string is {}",s1);
    // let len:usize = calculate_length(&s1); //borrow operation
    // println!("The length of {} is {}",s1,len);
}

// fn calculate_length(s2:&String)->usize{
//     return s2.len();
// }

fn append_string(s3:&mut String){
    s3.push_str("World");
}

```

## RULES OF ABOVE CONCEPT

```

fn main() {
    let mut s1:String = String::from("Hello");
    let w1 = &mut s1;
    w1.push_str(" World");
    println!("{}",w1);

    let w2 = &mut s1;
    w2.push_str(" Code");
    println!("{}",w1);
}

```

This above code gives an error in a second print statement as w1 is printed after w2 begins modified

To avoid race Conditions like the above example

Once a variable is referred(mutably or immutably) it can't be used in any way after another variable is mutably referred

## REFERENCING, DEREFERENCING, AUTO-DEREFERENCING

```

fn main() {
    let x = 5;
    let y = &x; //y is reference to the value of x, value of x is 5
    println!("{}",y); //auto dereferencing
}

```

Reference is not a pointer (it has more data than a pointer)

Pointer is unsafe that's why reference is used

The below code will give an error as & the mut integer type is not the same as the integer type due to auto dereferencing.

```
fn main() {  
    let mut x = 5;  
    x=x+1;//6  
    let y = &mut x;//y is reference to the value of x, value of x is 6  
    y=y+1;//7  
    println!("x={}",y);  
}
```

This will work fine.

```
fn main() {  
    let mut x = 5;  
    x=x+1;//6  
    let y = &mut x;//y is reference to the value of x, value of x is 6  
    *y=*y+1;//7  
    println!("x={}",y);  
}
```

Dangling Reference -

This is due to the issue that the local variable has a reference but the local variable is killed beyond its scope.

An example is shown below:

```

c > main.rs
1 //Dangling Reference
2 fn main() {
3     let reference_to_nothing = create_string_ref();
4 }
5
6 fn create_string_ref()->&String {
7     let s:String = String::from("hello");
8     return &s;
9 }
10
11

```

Programming concepts  
Data type in detail-

## Float Type

```

fn main() {
    let float32_number: f32 = 3.14; // f32 floating-point number
    let float64_number = 6.28;      // f64 floating-point number (type inference)

    println!("Float32 number: {}", float32_number);
    println!("Float64 number: {}", float64_number);
}

```

## Bool Type

```

fn main() {
    let is_raining: bool = true;
    let is_sunny = false;

    let need_umbrella = is_raining && !is_sunny;
    let need_glasses = is_raining || is_sunny;

    println!("need umberella is {}, need glasses is {}", need_umbrella, need_glasses);
}

```

In Rust, the `bool` data type represents a boolean value which can either be `true` or `false`.

# Character Type

```
fn main() {  
    let letter_a = 'a'; // ASCII character  
    let emoji = '😄'; // Unicode emoji  
    let kanji = '漢'; // Unicode character representing a Kanji character  
  
    println!("Letter a: {}", letter_a);  
    println!("Emoji: {}", emoji);  
    println!("Kanji character: {}", kanji);  
}
```

A char in Rust is always 4 bytes in size and can represent characters from various languages, including ASCII characters, emojis, and characters from non-Latin scripts.

## ARRAY -

```
src > main.rs  
1 fn main(){  
2     let arr: [&str; 3] = ["Hello", "World", "Coders"];  
3     write_arr(arr); //array directly pass  
4     println!("arr={:?}", arr);  
5 }  
6  
7 fn write_arr(mut arr1: [&str; 3]) { //arr1 new copy of arr  
8     arr1[0] = "Fellow";  
9     println!("arr1={:?}", arr1);  
10 }
```

OUTPUT    DEBUG CONSOLE    **TERMINAL**    PORTS

Compiling hello\_program v0.1.0 (C:\Users\user\hello\_program)  
Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.22s  
Running `target\debug\hello\_program.exe`  
arr1=["Fellow", "World", "Coders"]  
arr=["Hello", "World", "Coders"]  
PS C:\Users\user\hello\_program>

## Dynamic Array - Vector

## Type inference



# Type Inference

```
fn main() {  
    // Type inference in Rust  
    let x = 5; // Compiler infers that x is of type i32  
    let y = 5.5; // Compiler infers that y is of type f64  
    let z = "Hello, world!"; // Compiler infers that z is of type &str  
  
    // Printing the types inferred by the compiler  
    println!("Type of x: {}", type_of(&x));  
    println!("Type of y: {}", type_of(&y));  
    println!("Type of z: {}", type_of(&z));  
}
```

Type inference is a feature in programming languages that allows the compiler to deduce the data type of a variable or expression without explicit type annotations from the programmer.

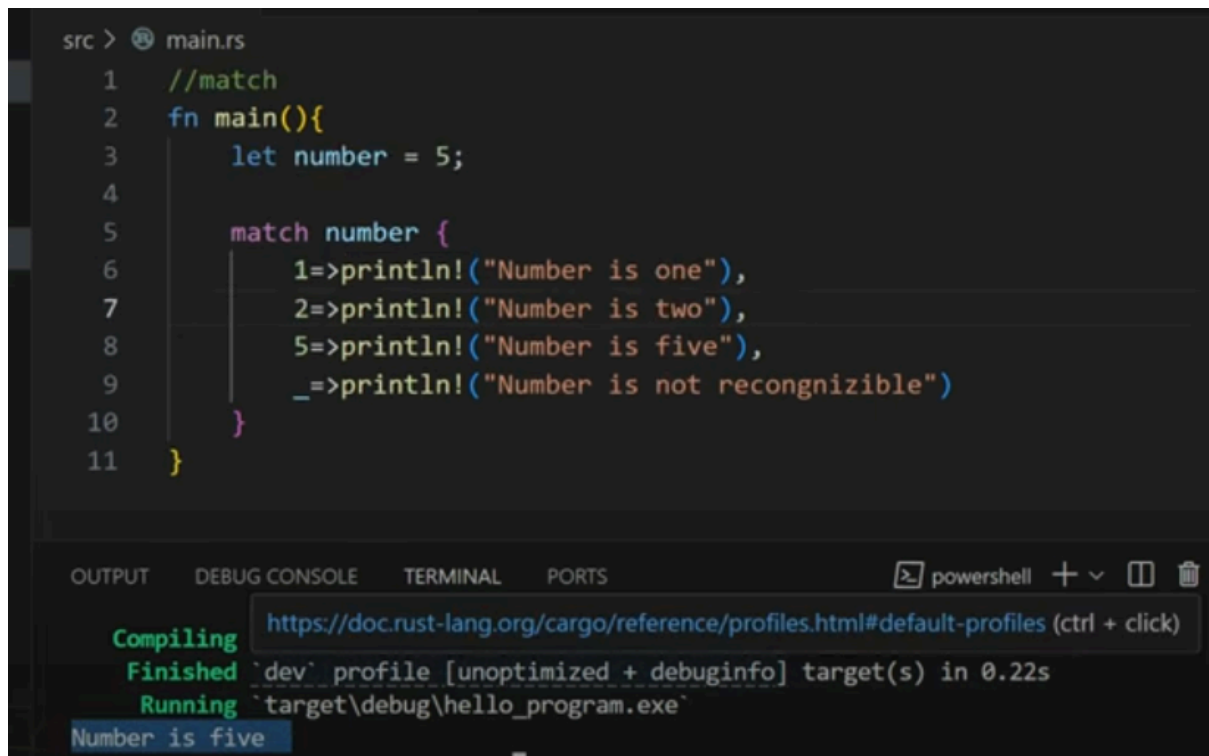
## Shadowing concept

The screenshot shows a code editor with a file explorer on the left and a code editor on the right. The file explorer shows a project structure with a 'src' directory containing 'main.rs'. The code editor shows the following Rust code:

```
1 //Shadowing  
2 fn main() {  
3     let x = 5; //integer  
4     let x = "Hello"; //string type  
5     let x = x.len(); //integer  
6 }  
7  
8  
9
```

This above code won't give any error, because here concept of shadowing is applied. Shadowing states that - if a variable name is used again and again and redeclared, so the compiler consider it a new variable and doesnot give any error.

Match -

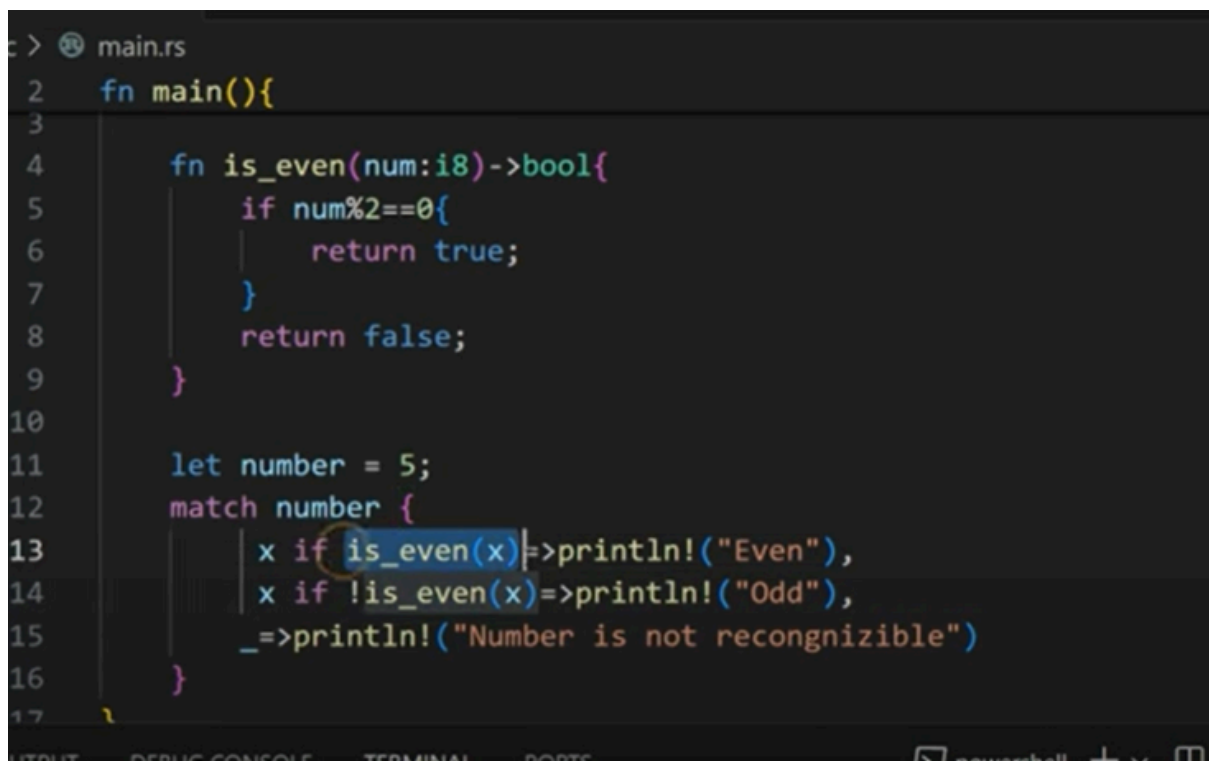


The screenshot shows an IDE with a Rust file named `main.rs`. The code defines a `main` function that uses a `match` statement to print the value of a variable `number` (set to 5). The output window shows the compilation and execution process, resulting in the text "Number is five".

```
src > main.rs
1 //match
2 fn main(){
3     let number = 5;
4
5     match number {
6         1=>println!("Number is one"),
7         2=>println!("Number is two"),
8         5=>println!("Number is five"),
9         _=>println!("Number is not recongnizable")
10    }
11 }
```

OUTPUT DEBUG CONSOLE TERMINAL PORTS powershell + - [X]

Compiling <https://doc.rust-lang.org/cargo/reference/profiles.html#default-profiles> (ctrl + click)  
Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.22s  
Running `target\debug\hello\_program.exe`  
Number is five



The screenshot shows an IDE with a Rust file named `main.rs`. The code defines a custom `is_even` function and uses it within a `match` statement to check if the variable `number` (set to 5) is even or odd. The output window shows the compilation and execution process, resulting in the text "Odd".

```
c > main.rs
2 fn main(){
3
4     fn is_even(num:i8)->bool{
5         if num%2==0{
6             return true;
7         }
8         return false;
9     }
10
11     let number = 5;
12     match number {
13         x if is_even(x)=>println!("Even"),
14         x if !is_even(x)=>println!("Odd"),
15         _=>println!("Number is not recongnizable")
16     }
17 }
```

OUTPUT DEBUG CONSOLE TERMINAL PORTS powershell + - [X]

Input from user -

```
1 use std::io;
2
3 fn main(){
4     let mut input = String::new();
5     println!("Please input your name:");
6     io::stdin()
7         .read_line(&mut input)
8         .expect("Input Failed");
9     |
10    println!("User input:{}",input);
11 }
12
```