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 CamalCase)

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)
3.

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

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.

```rust
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 owr
  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
fn main() {
  let s1:String = String::from("hello");//s1 owner
  let len = calculate_length(s1.clone());
  println!("The length of {} is {}",s1,len);
}

fn calculate_length(s:String)->usize{
   let length:usize = s.len();
   return length;//5
```

```
OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS                powershe

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

```rust
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

```rust
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

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

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

```rust
main.rs
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.

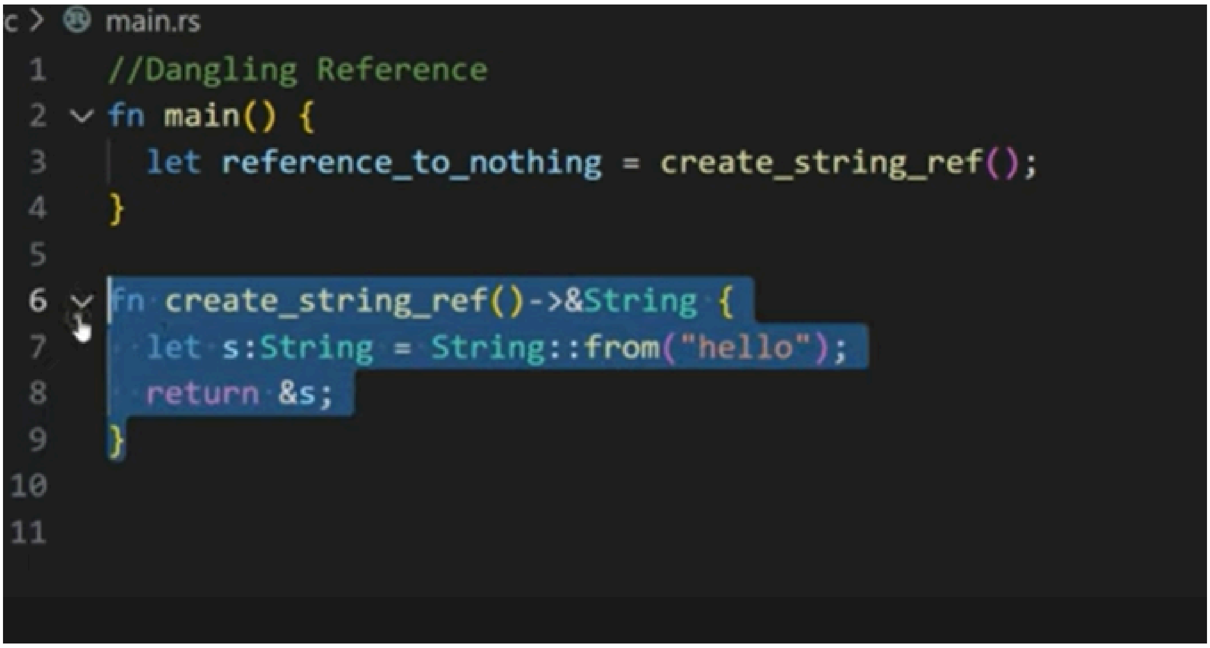
```
fn main() {
    let mut x = 5;
    x=x+1;//6
    let y = &mut x;//y is reference to the value of x, value of x i
    y=y+1;//7
    println!("x={}",y);
}
```

This will work fine.

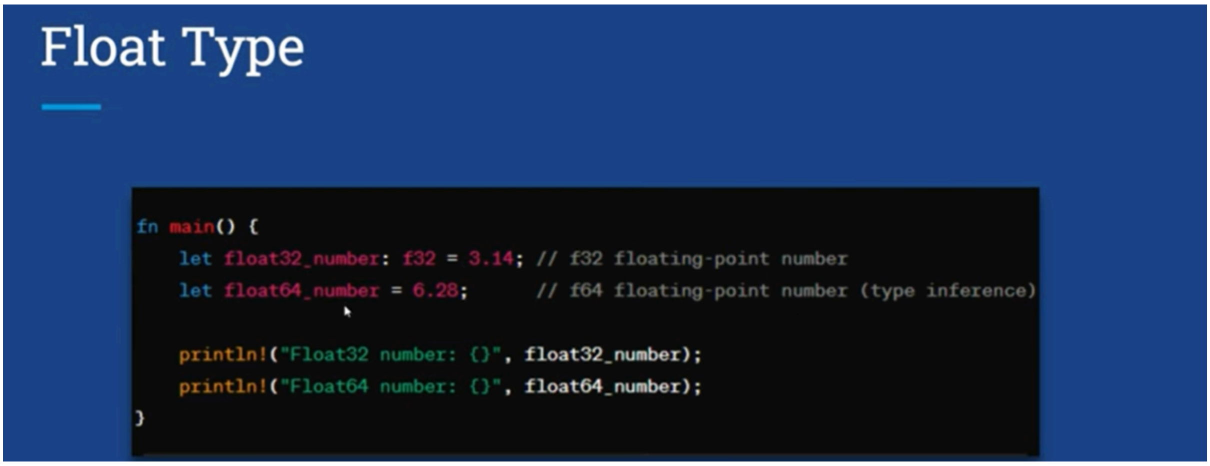
```
fn main() {
    let mut x = 5;
    x=x+1;//6
    let y = &mut x;//y is reference to the value of x, value of x i
    *y=*y+1;//7
    println!("x={}",y);
}
```

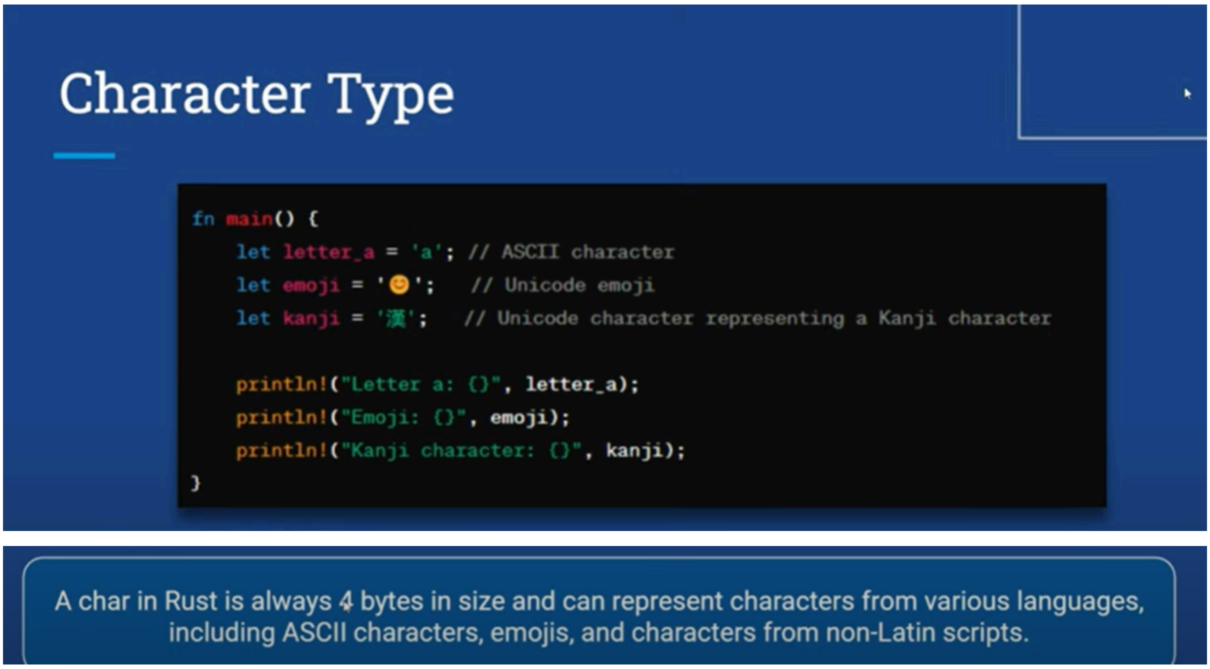
Dangling Reference -
This is due to the issue that the local variable has a reference but the local variable is killed beyond its scope.

An example is shown below:

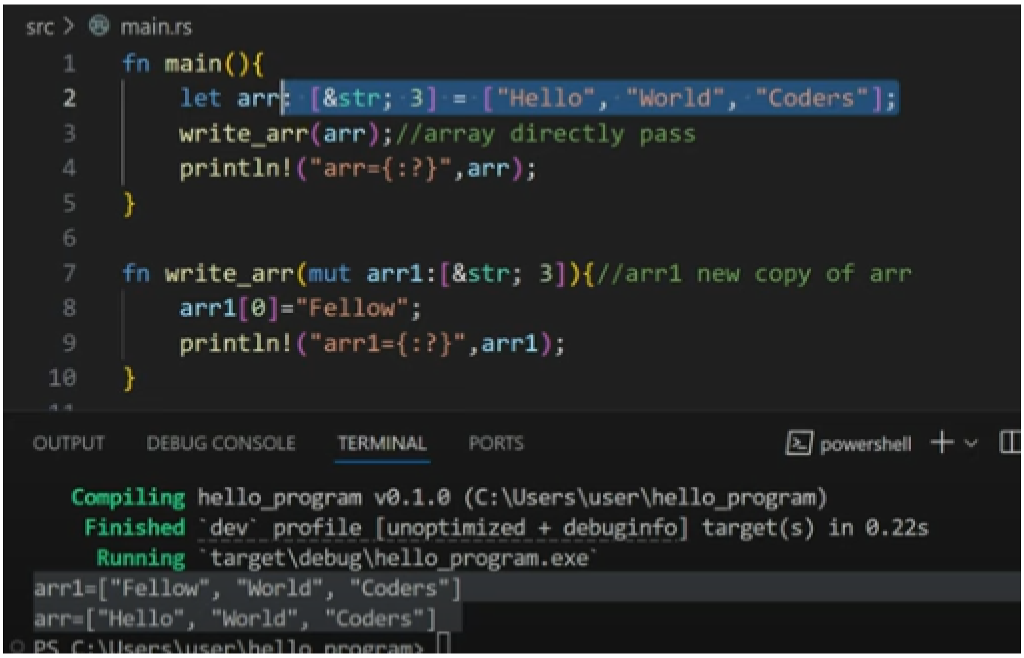


Programming concepts
Data type in detail-



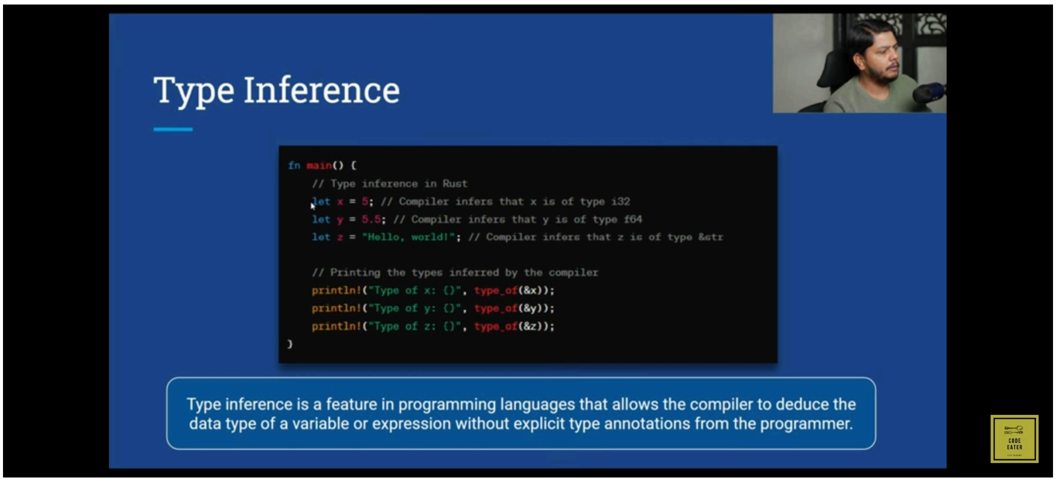


ARRAY -

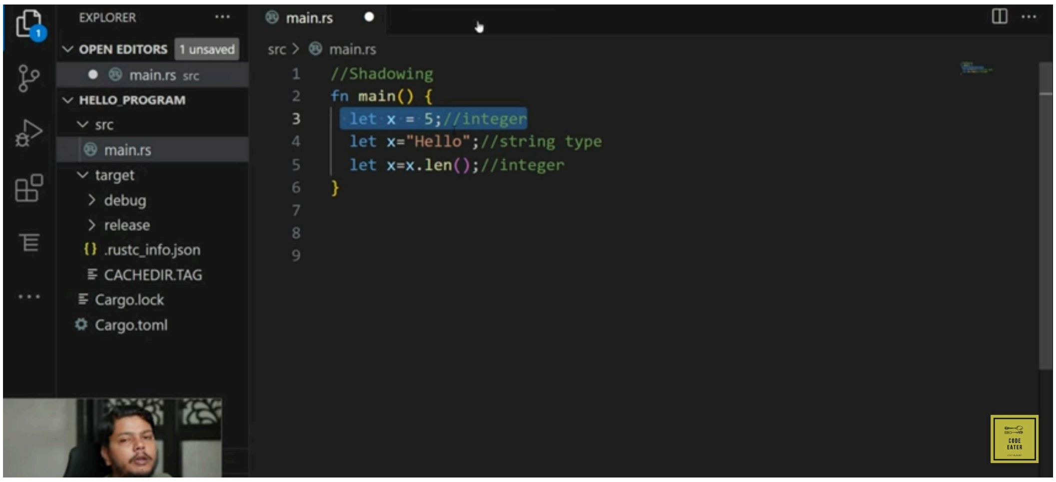


Dynamic Array - Vector

Type inference



Shadowing concept



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 -

```rust
//match
fn main(){
    let number = 5;

    match number {
        1=>println!("Number is one"),
        2=>println!("Number is two"),
        5=>println!("Number is five"),
        _=>println!("Number is not recongnizible")
    }
}
```

```
OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS                    powershell

   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
```

```rust
fn main(){

    fn is_even(num:i8)->bool{
        if num%2==0{
            return true;
        }
        return false;
    }

    let number = 5;
    match number {
        x if is_even(x)=>println!("Even"),
        x if !is_even(x)=>println!("Odd"),
        _=>println!("Number is not recongnizible")
    }
}
```

Input from user -

```
use std::io;

fn main(){
    let mut input = String::new();
    println!("Please input your name:");
    io::stdin()
        .read_line(&mut input)
        .expect("Input Failed");

    println!("User input:{}",input);
}
```