# MASTERING THE RUST LANGUAGE

A LoopCursive Book

ABHISHEK Building System Reliable

Applications LoopCursive Robust Scalable

Optimized Maintainable Flexible

# CONTENTS

# RUST INTRODUCTION

### WHAT IS PROGRAMMING

Programming is the process of creating instructions for computers to execute tasks or solve problems.

### WHAT IS PROGRAMMING LANGUAGE

A programming language is a set of instructions used to communicate with computers.

### WHAT IS RUST

Rust is an open-source programming language widely used to build operating system kernels, game engines, browser engines like those in Firefox, and command-line tools. It is known for its speed, memory efficiency, and focus on safety and concurrency.

### USE OF RUST

Rust is used for developing systems software, such as operating systems, game engines, web servers, and other performance-critical applications. It's valued for its efficiency, memory safety, and concurrency features, making it suitable for projects where performance and reliability are crucial.

### RUST INSTALLATION

- For Windows - visit **rustup.rs**, download and run rustup-init.exe
- Linux/macOS - open terminal and run `curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh`

## ENVIRONMENT VARIABLE

You may need to add Rust to your %PATH% on Windows. Visit 'create and modify environment variables'

## VERIFY INSTALLATION

```
rustc --version
```

## FEATURES OF RUST

1) **Memory Safety**: Rust prevents memory errors (like crashes) without needing garbage collection.
2) **Ownership System**: Rust controls how data is used to avoid mistakes and ensure efficient memory use.
3) **Concurrency Safety**: Rust helps you write safe multi-threaded code, avoiding issues like data races.
4) **Performance**: Rust runs as fast as C/C++ while keeping memory management safe.
5) **Error Handling**: Rust uses special types to make you handle errors clearly, reducing crashes.

## FIRST RUST PROGRAM

Create a file named main.rs and add the following lines. Rust files use the .rs extension.

## CODE

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

## COMPILE RUST PROGRAM

Open your terminal and run these commands to compile and execute your Rust program.

## WINDOWS

```
> rustc main.rs
> ./main.exe
```

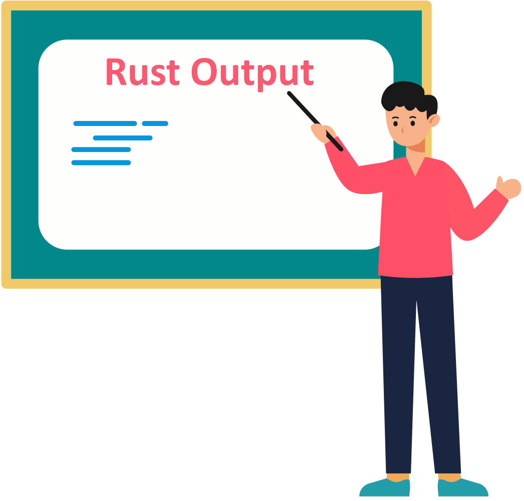
## LINUX/MAC

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

## NOTE

rustc main.rs compiles your Rust program
./main or ./main.exe runs the program

# RUST OUTPUT



### HELLO WORLD

A "Hello, World!" program prints the text "Hello, World!" to the screen. It's a simple program commonly used to introduce beginners to a new programming language.

### CODE

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

### WORKING : HELLO WORLD

- fn main(): Declares the main function, the program's entry point.
- println!("Hello, World!");: Prints "Hello, World!" to the console, followed by a newline.

### RUST PRINT OUTPUT

In Rust, we actually use the println! macro to print strings, numbers, and variables on the output screen.

### CODE

```
fn main() {
    print!("Hello, World!");
}
```

### VARIATIONS OF THE PRINT MACRO

1. print!: Prints without a newline
2. println!: Prints with a newline

### NOTE

print! is used when you want to print multiple items on the same line.
while println! is used when you want to print each item on a new line.

### RUST PRINT! MACRO

The `print!` macro prints text inside double quotes.

### CODE

```
fn main() {
    print!("Hello, World!");
}
```

### RUST PRINTLN! MACRO

The `println!` macro prints text inside double quotes and adds a newline at the end.

### CODE

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

### RUST MACRO

A Rust macro is a piece of code that generates other code.

### PRACTISE PROBLEMS

1) What is the difference between print! and println! in Rust?
2) How do you print the string "Hello, world!" to the console in Rust using println!?

# RUST COMMENTS

## COMMENTS

Comments are notes in your code that the computer ignores. They're for humans to understand.

## TYPES OF COMMENTS

1) Single-Line Comment //
2) Multi-line Comments /* */

### SINGLE LINE COMMENT

Starts with // and everything after it on the same line is ignored by the compiler

### CODE

```
fn main () {
    // declare a variable
    let x = 1;
    println!("x = {}", x);
}
```

### MULTI LINE COMMENT

Starts with /* and ends with */. Everything between is ignored by the compiler and can span multiple lines.

### CODE

```
fn main() {
    /*
    declare a variable
    and assign value to it
    */
    let x = 1;
    println!("x = {}", x);
}
```

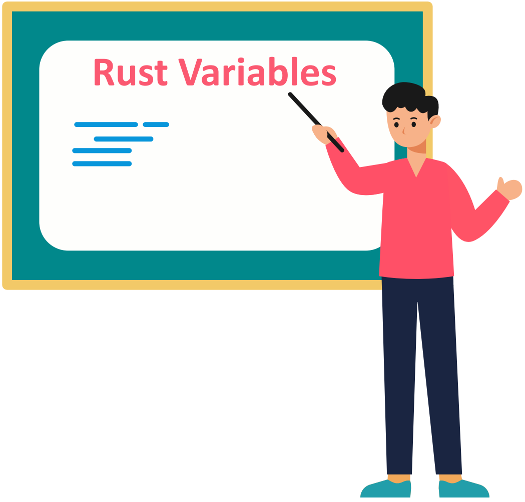
**Note**

Use comments to explain complex code, describe functions, leave notes, disable code, and provide context, while keeping them short, clear, and up-to-date.

PRACTISE PROBLEMS

1) Write a simple Rust program that prints "Hello, World!" to the console, and add a comment explaining what the program does.
2) Write a Rust program to print a message, then comment out the print statement. What happens when you run it?

# RUST VARIABLES



## RUST VARIABLES

A variable is a storage for a value, which can be a string, a number, or something else. Every variable has a name (or an identifier) to distinguish it from other variables. You can access a value by the name of the variable.

## VARIABLE DECLARATION

Use the let keyword to declare a variable in Rust

## CODE

```
fn main() {
    let x = 5;
}
```

## PRINT VARIABLES

We use print! and println! macros to print variables.

## CODE

```
fn main() {
    let x = 5; // Immutable variable
    println!("x = {}", x);
}
```

## PLACEHOLDER

1) {} is a placeholder which is replaced by the value of the variable after the comma.
2) We can use multiple placeholders in the same `print!` or `println!` macro to print multiple variable values.
3) "Placeholders `{first}`, `{second}`, etc., are replaced by the corresponding variable values."

## RUST VARIABLE NAMING RULES

1) Snake Case: Use lowercase letters with underscores (e.g., my_variable).
2) Start with Letter/Underscore: Begin with a letter or _, not a number.
3) Allowed Characters: Use letters, digits, and underscores.
4) No Special Characters: Avoid symbols like @, #, or -.
5) No Reserved Keywords: Don’t use Rust’s keywords (e.g., fn, let).
6) Case Sensitive: my_variable is different from My_Variable.
7) Be Descriptive: Use meaningful names for clarity.

## KEYWORDS

Keywords are reserved words that have a special meaning to the language. They are used to define the structure and syntax of the language.

## KEYWORDS LIST

| | | | | | |
|---|---|---|---|---|---|
| as | async | await | break | const | continue |
| crate | dyn | else | enum | extern | fn |
| for | if | impl | in | let | loop |
| match | mod | move | mut | pub | ref |
| return | Self | self | struct | super | trait |
| type | unsafe | use | where | while | yield |

## NOTE

This list might not be complete, as new keywords might be added in future versions of Rust. However, this list covers all the current keywords in Rust.

## MUTABLE VARIABLE

A variable whose value can be changed after it's set.

## IMMUTABLE VARIABLE

A variable whose value cannot be changed after it's set. By default rust variables are immutable.

## MODIFY VARIABLE

By default, variables are immutable. To make a variable mutable, add mut

### MUTABILITY IN RUST

We use the `mut` keyword to make a variable mutable.

### CODE

```
fn main() {
    let mut x = 5;  variable
    println!("Initial value of {}", x);

    x = 10; // Modify the value
    println!("Modified value {}", x);
}
```

### RUST CONSTANTS

A constant is a special type of variable whose value cannot be changed. In Rust, we use the const keyword to create constants.

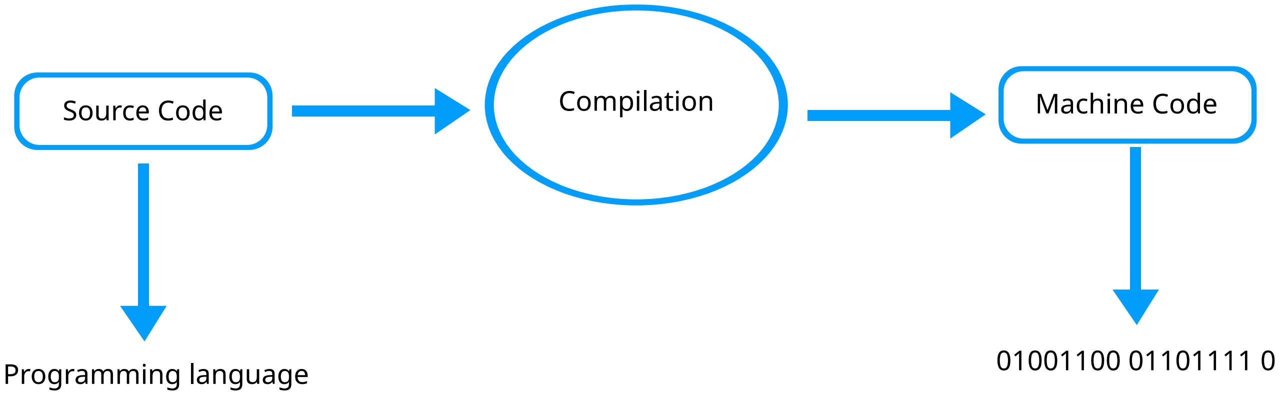
it is required to specify the data type when defining constants

### CODE

```
const PI: f64 = 3.14159;

fn main() {
    println!("The value of PI is: {}",
PI);
}
```

### KEY TERMS

1) Inferred: Rust automatically figures out the type of a variable from its value.
2) Explicitly: You manually specify the type of a variable or constant.
3) Runtime: The time when the program is running and executing code.
4) Compile Time: The time when the code is being compiled before it runs.
5) Scope: The region in the code where a variable or constant is accessible and can be used.

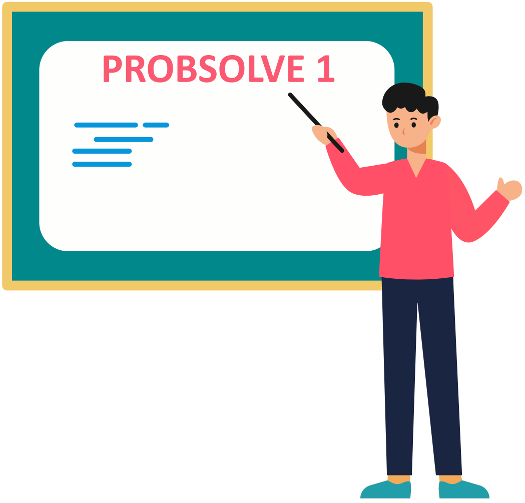


| CONSTANTS | V/S | IMMUTABLE VARIABLE |
|---|---|---|
| 1) A fixed value that never changes | | 1) A value that can't be changed after being set |
| 2) `let` (e.g., `let x = 5;`) | | 2) `let` (e.g., `let x = 5;`) |
| 3) Must be explicitly declared | | 3) Can be inferred or explicitly declared |
| 4) Globally accessible based on visibility | | 4) Limited to the block or function where it's defined |
| 5) Value is set at compile time | | 5) Value is set at runtime |
| 6) Can be used anywhere in the module or crate | | 6) Can be used within its scope |

## PRACTISE PROBLEMS

1) How do you declare an immutable variable in Rust, and what happens if you try to change its value later?
2) What is a mutable variable in Rust, and how do you declare it?
3) What is the purpose of the mut keyword in Rust variable declarations?
4) How do you declare a constant in Rust, and what are the rules for naming constants?
5) In Rust, what is the preferred case style for variable names, and how does it differ from other programming languages?
6) Can you use keywords like let, fn, and mut as variable names in Rust? Why or why not?

# PROB-SOLVE 1



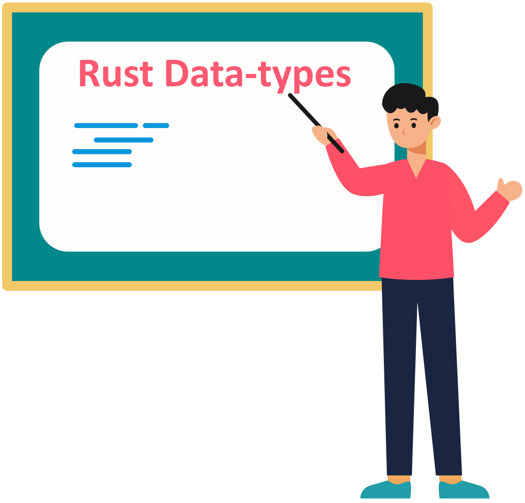
**PROBE SOLVE 1 PRACTISE PROBLEMS**

1) What will be the output of the following Rust code?

```
fn main() {
   println!("{} + {} = {}", 5, 10, 5 + 10);
}
```

2) What are the two types of comments in Rust? Provide an example of each.
3) What is the difference between mutable and immutable variables in Rust? Provide an example.
4) What will happen if you try to modify an immutable variable in Rust?

# RUST DATA TYPES



## RUST DATA TYPES

A data type specifies the kind of value a variable can hold.

## MAIN DATA TYPES IN RUST

1) Scalar types
2) Compound types
3) Custom Types
4) Special types

## SCALAR TYPES IN RUST

- A scalar type represents a single value.
- Rust offers four primary scalar types
    - Integer
    - Floating-point
    - Boolean
    - Character

## INTEGER TYPES

In Rust, we use integer types to store whole numbers and are categorized by their size and whether they are signed or unsigned.Integers are classified into signed and unsigned types, each with different sizes.

## CATEGORIES OF INTEGER DATA TYPES

| Size | Signed | Unsigned |
|---|---|---|
| 8-bit | i8 | u8 |
| 16-bit | i16 | u16 |
| 32-bit | i32 | u32 |
| 64-bit | i64 | u64 |
| 128-bit | i128 | u128 |
| Platform-dependent | isize | usize |

## SIGNED INTEGER

In Rust, a signed integer is a type of integer that can store both positive and negative whole numbers.

## CODE

```
fn main() {
    // Define some signed integers
    let x: i32 = 42;
    let y: i32 = -7;

    // Print the values
    println!("x: {}", x);
    println!("y: {}", y);
}
```

## NOTE

In Rust, `i32` is the default integer type for literals because it handles both positive and negative values and offers a good balance of range and performance.

## Formulas for Calculating the Range of Signed Integers.

### Formula

For an n-bit signed integer

- Minimum Value : $-2^{(n-1)}$
- Maximum Value : $2^{(n-1)} - 1$

### Example

For an `i8` (8 bit signed integer)

- Minimum Value: $-2^{(8-1)} = -2^{7} = -128$
- Maximum Value : $2^{(8-1)} – 1 = 2^{7} – 1 = 127$

## Formulas for Calculating the Range of Unsigned Integers.

| Formula | Example |
|---|---|
| For an n-bit unsigned integer<br>- Minimum Value : 0<br>- Maximum Value : $2^n$ - 1 | For an \`u8\` (8 bit unsigned integer)<br>- Minimum Value: 0<br>- Maximum Value : $2^8$ – 1 = $2^8$ – 1 = 256 – 1 = 255 |

## UNSIGNED INTEGER

In Rust, a unsigned integer is a type of integer that can store only positive whole numbers.

## CODE

```rust
fn main() {
    // Define some unsigned integers
    let a: u32 = 42;
    let b: u32 = 7;

    // Print the values
    println!("Value of a: {}", a);
    println!("Value of b: {}", b);
}
```

## VALUE RANGES FOR SIGNED INTEGER

| Type | Range |
|---|---|
| i8 | -128 to 127 |
| i16 | -32,768 to 32,767 |
| i32 | -2,147,483,648 to 2,147,483,647 |
| i64 | -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 |
| i128 | -170,141,183,460,469,231,731,687,303,715,884,105,727 to 170,141,183,460,469,231,731,687,303,715,884,105,727 |

## VALUE RANGE FOR UNSIGNED INTEGER

| Type | Range |
|---|---|
| u8 | 0 to 255 |
| u16 | 0 to 65,535 |
| u32 | 0 to 4,294,967,295 |
| u64 | 0 to 18,446,744,073,709,551,615 |
| u128 | 0 to 340,282,366,920,938,463,463,374,607,431,768,211,455 |

## `isize` IN RUST

- Type Signed integer type.
- Size On a 64-bit system isize is 64 bits wide, and on a 32-bit system, it is 32 bits wide.
- Usage `isize` is a number type in Rust that can be positive or negative, used for things like indexing and pointer calculations.

## `usize` IN RUST

- Type Unsigned integer type.
- Size On a 64-bit system isize is 64 bits wide, and on a 32-bit system, it is 32 bits wide.
- Usage `usize` is an unsigned integer type in Rust used for indexing and sizes, where only positive values are needed.

## FLOATING-POINT TYPE

In Rust, we use floating-point type to store fractional number.

## CATEGORIES OF FLOATING-POINT DATA TYPES

Floating-point numbers in Rust are classified into f32 and f64, each with different precision and sizes

### FLOATING-POINT: f32

`f32` is a 32-bit floating-point type in Rust with about 7 decimal digits of precision.

### CODE

```
fn main() {
    let x: f32 = 3.14;
    println!("Value: {}", x);
}
```

### FLOATING-POINT: f64

`f64` is a 64-bit floating-point type in Rust with about 15 decimal digits of precision.Rust uses f64 by default.

### CODE

```
fn main() {
    let x: f64 = 3.141592653589793;
    println!("Value: {}", x);
}
```

## BOOLEAN TYPE

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

## CODE

```
fn main() {
    let is_active: bool = true;
    let is_completed: bool = false;

    println!("{}", is_active);
    println!("{}", is_completed);
}
```

## CHARACTER TYPE

The character data type in Rust is used to store a character.
We use sinle quotes to represent a chanracter.

## CODE

```
fn main() {
    let letter: char = 'A';
    let symbol: char = '✓';
    let emoji: char = '😊';

    println!("Letter: {}", letter);
    println!("Symbol: {}", symbol);
    println!("Emoji: {}", emoji);
}
```

## TYPE INFERENCE IN RUST

Type inference is Rust's feature to automatically determine the type of a variable based on its value without type annotations.

## CODE

```
fn main() {
    let number = 42;
    let pi = 3.14;
    let letter = 'A';

    println!("Number: {}", number);
    println!("Pi: {}", pi);
    println!("Letter: {}", letter);
}
```

## RUST TYPE CASTING

- Type casting in Rust means converting a value from one type to another using the `as` keyword.

## IMPORTANT POINTS

1) Use the as keyword to change one type of data into another, like turning an i32 into u8.
2) Rust doesn't automatically convert types.
3) When converting, especially from a larger type to a smaller one, you might lose data (e.g., turning 3.14 into 3).

### INTEGER TO INTEGER

Converts a larger integer type (i32) to a smaller integer type (u8). Note that values that don’t fit in the smaller type will wrap around.

### CODE

```
fn main() {
    let large_number: i32 = 500;
    let small_number: u8 = large_number as u8;
    println!("Integer to Integer: {}",small_number);
}
```

### INTEGER TO FLOATING-POINT

Converts an integer (i32) to a floating-point number(f64). The integer is transformed into a number with decimal points.

### CODE

```
fn main() {
    let integer: i32 = 42;
    let floating: f64 = integer as f64;
    println!("Integer to Floating-Point: {}",
floating);
}
```

### FLOATING-POINT TO INTEGER

Converts a floating-point number(f64) to an integer(i32). The decimal part of the floating-point number is discarded.

### CODE

```
fn main() {
    let float: f64 = 3.99;
    let integer: i32 = float as i32;
    println!("Floating-Point to Integer: {}",
integer);
}
```

### FLOATING-POINT TO FLOATING-POINT

This example converts a high-precision f64 number to a lower-precision f32 number, demonstrating how the precision is reduced.

### CODE

```
fn main() {

    let double_precision: f64 = 3.141592653589793;
    let single_precision: f32 = double_precision as
f32;
    println!("Double precision (f64): {}",
double_precision);
    println!("Single precision (f32): {}",
single_precision);
}
```

### CHARACTER TO INTEGER

Convert a character to its Unicode code point (integer value).

### CODE

```
fn main() {
    let character: char = 'A';
    let code_point: u32 = character as u32;
    println!("Character: {}", character);
    println!("Unicode code point (int): {}",
code_point);
}
```

### INTEGER TO CHARACTER

Convert an integer representing a Unicode code point to a character.

### CODE

```
fn main() {
    let code_point: u32 = 66;
    let character: char = code_point as char;
    println!("Unicode code point (int): {}",
code_point);
    println!("Character: {}", character);
}
```

## UNICODE RANGE

1) A – z -> 65 to 90
2) a – z -> 97 to 122
3) 0 – 9 -> 48 to 57

## UNICODE

Unicode gives every letter, number, and emoji a special number so that text appears the same everywhere.

## NOTE

In Rust, you can only cast a u8 integer to a character, using as char. Using any other integer type will result in a compile-time error.

## BOOLEAN TO INTEGER

- true becomes 1 when cast to an integer
- false becomes 0 when cast to an integer

## CODE

```
fn main() {
    let my_bool: bool = true;
    let my_int: u8 = my_bool as u8;

    println!("my_bool: {}, my_int: {}", my_bool,
my_int);

    let my_bool2: bool = false;
    let my_int2: u8 = my_bool2 as u8;

    println!("my_bool2: {}, my_int2: {}", my_bool2,
my_int2);
}
```

## LIMITATIONS OF TYPE CASTING

There are some limitations while performing type casting in Rust.

1) **Loss of Precision:** Converting from a floating-point number to an integer results in losing the fractional part. For example, 3.7 becomes 3.
2) **Overflow and Underflow:** Casting a value that exceeds the range of the target type (e.g., a large integer to a smaller integer type) can lead to overflow, causing unexpected wraparound or errors
3) **Incompatibility Between Types:** Some types cannot be directly cast to one another. For example, you cannot cast a floating-point number directly to a character type in Rust.
4) **Rust-Specific Constraints:** Rust has strict type safety rules. Not all types are convertible, and attempting invalid conversions will result in compiler errors.

# RUST OPERATORS

## RUST OPERATORS

In Rust, operators are symbols used to perform operations on values.

## RUST OPERATOR TYPES

1) Arithmetic Operators
2) Compound Assignment Operators
3) Logical Operators
4) Comparison Operators

| OPERAND | OPERATOR | OPERAND |
|---|---|---|
| 5 | + | 5 |

## ARITHMETIC OPERATORS

We use arithmetic operators to perform addition, subtraction, multiplication, and division.

| ARITHMETIC OPERATORS TYPES | EXAMPLE |
|---|---|
| 1) + | a+b |
| 2) - | a-b |
| 3) * | a*b |
| 4) / | a/b |
| 5) % | a%b |

## ASSIGNMENT OPERATORS

We use an assignment operator to assign a value to a variable. (e.g. let x = 5;)

## COMPOUND ASSIGNMENT OPERATORS

Compound assignment operators in Rust combine an arithmetic operation with assignment.

| COMPOUND ASSIGNMENT OPERATORS TYPES | EXAMPLE |
|---|---|
| 1) += | a += 5 |
| 2) -= | a  -= 5 |
| 3) *= | a *= 5 |
| 4) /= | a /= 5 |
| 5) %= | a %= 5 |

## COMPARISON OPERATORS

We use comparison operators to compare two values or variables. A relational operator returns:
-true if the relation between two values is correct
-false if the relation is incorrect

| COMPARISON OPERATORS TYPES | EXAMPLE |
|---|---|
| 1) > (Greater than) | a > b |
| 2) >= (Greater than and equal to) | a >= b |
| 3) < (Smaller than) | a < b |
| 4) <= (Smaller than and equal to) | a <= b |
| 5) == (Equal to) | a == b |
| 6) != (Not equal to) | a!= b |

## LOGICAL OPERATOR

We use logical operators to make decisions in our code. A logical operation gives us either true or false based on the conditions.

| LOGICAL OPERATOR TYPES | EXAMPLE |
|---|---|
| 1) **&&** (Logical AND) – Returns true if both conditions are true. | (true and true) -> true |
| 2) **\| \|** (Logical OR) – Returns true if at least one condition is true. | (true and false) -> true |
| 3) **!** (Logical NOT) - Reverses the value; if it's true, it becomes false, and if it's false, it becomes true. | (true) -> false and (false) -> true |

## PRECEDENCE AND ASSOCIATIVITY

1) **Operator precedence** : controls which operators are evaluated first in an expression; higher precedence operators are evaluated before lower precedence ones.

2) **Associativity** : determines the order in which operators of the same precedence level are evaluated, typically left-to-right or right-to-left.

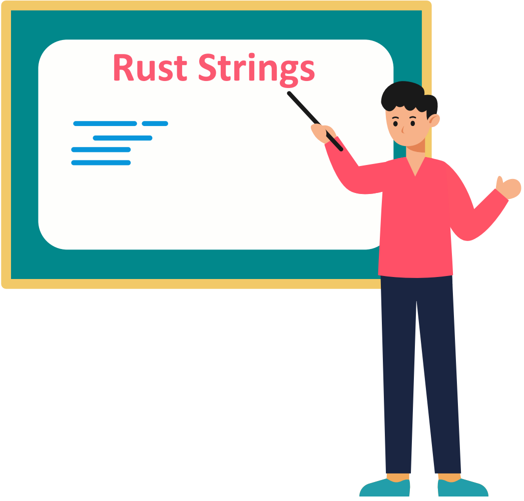
## RUST OPERATOR PRECEDENCE AND ASSOCIATIVITY TABLE

| PRECEDENCE | OPERATOR | ASSOCIATIVITY |
|---|---|---|
| 1 | ? | Right-to-left |
| 2 | as | Left-to-right |
| 3 | * , / , % | Left-to-right |
| 4 | + , - | Left-to-right |
| 5 | << , >> | Left-to-right |
| 6 | & | Left-to-right |
| 7 | ^ | Left-to-right |
| 8 | ==, !=, <, >, <=, >= | Left-to-right |
| 9 | && | Left-to-right |
| 10 | \|\| | Left-to-right |
| 11 | ! | Right-to-left |
| 12 | = (assignment), +=, -=, *=, /=, %= | Right-to-left |

## PRACTISE PROBLEMS

1) What is the difference between i32 and u32 in Rust?
2) What is the default integer and floating-point type in Rust if you don’t specify one?
3) What will be the output? -> (50 / (5 + 5)) * (8 - 3) + 2 * (4 - 1);
4) How do you declare a boolean variable in Rust?
5) How do you cast a f64 to an i32 in Rust?
6) What happens when you cast a larger integer type to a smaller one in Rust?
7) What is the purpose of the ! operator in Rust?
8) What is the difference between the = and == operators?
9) How do you define a constant in Rust? Write an example of a constant named MAX_USERS with a value of 1000.
10) What are the arithmetic operators in Rust? Give an example of using the +, -, and % operators.

# RUST STRINGS



## RUST STRINGS

In Rust strings are sequences of characters.

## RUST STRING TYPES

1) String literal(&str)
2) String object(String)

## STRING LITERAL(&str)

String literals are immutable, hardcoded strings that are known at compile time

## CODE

```
fn main() {
    let name: &str = "Abhishek";
    println!("{}", name);
}
```

## STRING OBJECT (String)

String objects (`String`) are mutable, heap-allocated strings that are created at runtime.

## CODE

```
fn main() {
    let mut greeting =
String::from("Hello");
    println!("{}", greeting);
}
```

## DECLARING STRING OBJECT

1) String::new()
2) String::from()

### String::new()

Creates a new, empty `String` object in Rust that can be modified later.

### CODE

```
fn main() {
    let mut my_string = String::new();
    my_string.push_str("Hello, Rust!");
    println!("{}", my_string);
}
```

### String::from()

This creates a string object with some default value.

### CODE

```
fn main() {
    let my_string = String::from("Hello,
Rust!");
    println!("{}", my_string);
}
```

### STRING LITERAL

String literals are immutable, fixed sequences of characters stored on the stack memory.

### VS

### STRING OBJECT

String objects are mutable, growable sequences of characters stored on the heap memory.

## STRING CONCATENATION

In Rust, you can concatenate (combine) strings using different methods:

1. Using + Operator (Moves Ownership)
2. Using format!() (Recommended)
3. Using push() and push_str() (For Mutable Strings)
4. Using Iterators and collect()

### Using + Operator

- The + operator takes ownership of the first String, so it cannot be used afterward.
- The second string must be a reference (&str), not a String.

### CODE

```
fn main() {
    let s1 = String::from("Hello");
    let s2 = String::from(", Rust!");
    let s3 = s1 + &s2; // s1 is moved,
cannot be used again
    println!("{}", s3); // Output: Hello,
Rust!
```

```
    // println!("{}", s1); ❌ Error: s1 is
moved
}
```

### Using format!()

The format!() macro does not take ownership and allows flexible string concatenation.

### CODE

```
fn main() {
    let s1 = String::from("Hello");
    let s2 = String::from(", Rust!");

    let s3 = format!("{}{}", s1, s2); //
s1 & s2 are NOT moved
    println!("{}", s3); // Output: Hello,
Rust!

    // s1 and s2 are still usable
    println!("{}", s1); // ✅ Works fine
}
```

### Using push() and push_str()

- push() → Appends a single character (char).
- push_str() → Appends a string slice (&str).

### CODE

```
fn main() {
    let mut s = String::from("Hello");

    s.push('!');        // Appends a single
character
    s.push_str(" Rust"); // Appends a
string slice

    println!("{}", s); // Output: Hello!
Rust
}
```

### Using Iterators and collect()

Used for dynamically joining multiple strings.

### CODE

```
fn main() {
    let words = vec!["Hello", "Rust",
"World"];
    let sentence = words.join(" "); //
Joins with a space

    println!("{}", sentence); // Output:
Hello Rust World
}
```

### STRING TYPE CASTING

Convert integer into string using .to_string()

### CODE

```
fn main() {
    let number = 2020; // Integer
    let number_as_string =
number.to_string(); // Convert number to
String
```

```
    println!("{}", number_as_string); //
Output: "2020" (String)
    println!("{}", number_as_string ==
"2020"); // Output: true
}
```

## STRING SLICING

In Rust, string slicing allows you to extract a part of a string without copying the data.

## CREATING STRING SLICE

A string slice is a reference (&str) to a portion of a String. You use range indexing [start..end], where:

- start is the beginning index (inclusive).
- end is the ending index (exclusive).

### BASIC STRING SLICING

String slicing allows you to extract a part of a string.

### CODE

```
fn main() {
    let s = String::from("Hello, Rust!");

    let slice = &s[0..5]; // Get "Hello"
(indexes 0 to 4)
    println!("{}", slice);
}
```

### OMITTING START OR END INDEX

- &s[..end] → Slice from the beginning to end – 1.
- &s[start..] → Slice from start to the end of the string.

### CODE

```
fn main() {
    let s = String::from("Hello, Rust!");

    let slice1 = &s[..5];  // "Hello"
(from start)
    let slice2 = &s[7..];  // "Rust!"
(until end)

    println!("{}", slice1);
    println!("{}", slice2);
}
```

## STRING METHODS

Rust provides various methods to manipulate strings. Here are some commonly used ones:

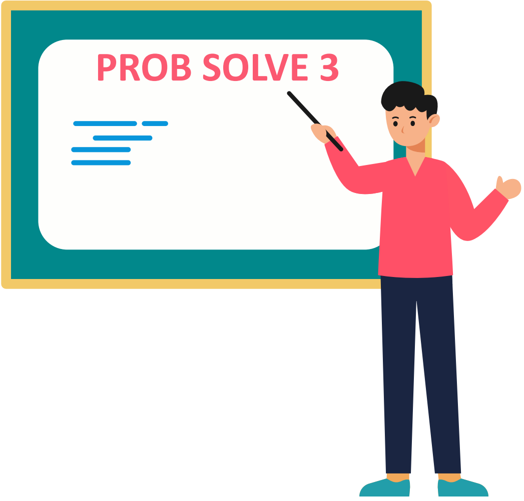
| Method | Description | Example |
|---|---|---|
| .len() | Returns the length | `"Hello".len() → 5` |
| .is_empty() | Checks if the string is empty | `"".is_empty() → true` |
| .push(char) | Appends a single character | `s.push('R')` |
| .push_str(&str) | Appends a string slice | `s.push_str("ust")` |
| .insert(index, char) | Inserts a character at a position | `s.insert(5, '!')` |
| .insert_str(index, &str) | Inserts a string slice at a position | `s.insert_str(5, " Rust")` |
| .replace(&str, &str) | Replaces all occurrences of a substring | `"hello".replace("l", "x") → "hexxo"` |
| .trim() | Removes leading/trailing whitespace | `" Rust ".trim() → "Rust"` |
| .split(&str) | Splits a string into an iterator | `"a,b,c".split(",") → ["a", "b", "c"]` |
| .chars() | Returns an iterator over characters | `for c in "Rust".chars()` |
| .contains(&str) | Checks if substring exists | `"Rust".contains("us") → true` |
| .starts_with(&str) | Checks if string starts with a substring | `"Rust".starts_with("Ru")` |
| .ends_with(&str) | Checks if string ends with a substring | `"Rust".ends_with("st")` |
| .to_lowercase() | Converts to lowercase | `"RUST".to_lowercase() → "rust"` |
| .to_uppercase() | Converts to uppercase | `"rust".to_uppercase() → "RUST"` |
| .to_string() | Converts to String | `5.to_string() → "5"` |

## ESCAPE SEQUENCE

Escape sequences are special character combinations used to represent characters that cannot be typed directly in a string. They start with a backslash (\).

| ESCAPE SEQUENCE | USE CASE | Meaning |
|---|---|---|
| `\n` | New line | Newline Moves text to a new line |
| `\t` | Tab | Adds a tab space |
| `\\` | Back Slash | Prints a backslash |
| `\’` | Single Quote | Prints a single quote |
| `\”` | Double Quote | Prints a double quote |
| `\r` | Carriage Return | Moves cursor to the beginning of the line |

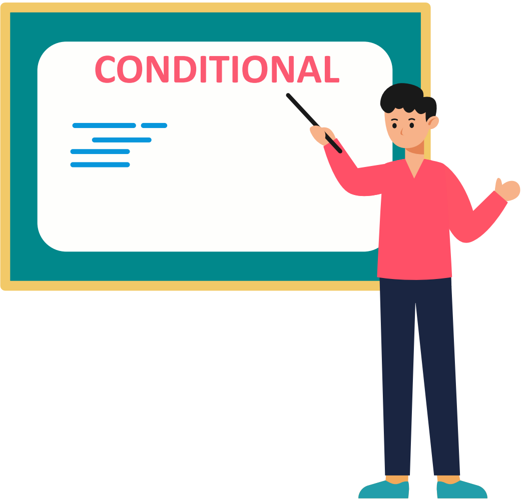
# PROB-SOLVE 3



## PROBE SOLVE 3 PRACTISE PROBLEMS

1. What is the difference between String and &str in Rust?
2. How do you create an empty String in Rust?
3. How can you convert a &str into a String?
4. How do you concatenate two String values using the + operator?
5. What is the difference between using + and format!() for concatenation?
6. How do you append a character to a String in Rust?
7. How do you extract a substring from a String in Rust?
8. What is the difference between &s[..], &s[0..5], and &s[5..] in Rust?
9. What does \n, \t, and \\ do in Rust strings?
10. How do you include a double quote (") inside a string in Rust?
11. How do you convert a number into a String?
12. How do you convert a String to uppercase?
13. How do you check the length of a String in Rust?
14. What method is used to check if a String is empty?
15. What is the difference between .push() and .push_str() in Rust?

# RUST CONDITIONALS



## CONDITIONAL STATEMENTS

Conditional statements allow programs to make decisions based on conditions. Rust provides the following:

1) if-else Statement
2) else-if Statement
3) match Expression

## IF-ELSE

The if statement checks a condition.

- If true, it executes the first block.
- If false, it executes the else block.

## CODE

```
fn main() {
    let age = 18;

    if age >= 18 {
        println!("You can vote!");
    } else {
        println!("You are too young to
vote.");
    }
}
```

### ELSE-IF

Use else if to check multiple conditions in sequence.

### CODE

```
fn main() {
    let number = 0;

    if number > 0 {
        println!("Positive number");
    } else if number < 0 {
        println!("Negative number");
    } else {
        println!("Zero");
    }
}
```

### MATCH STATEMENT

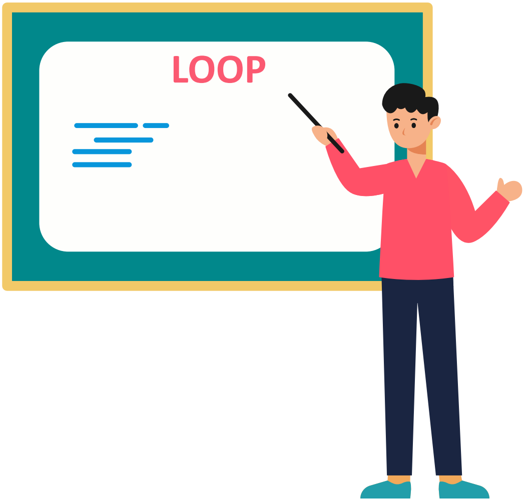
The match expression compares a value against multiple patterns and executes the matching case.

### CODE

```
fn main() {
    let day = 3;
    let result = match day {
        1 => "Monday",
        2 => "Tuesday",
        3 => "Wednesday",
        4 => "Thursday",
        5 => "Friday",
        6 | 7 => "Weekend!",
        _ => "Invalid day",
    };
    println!("Day: {}", result);
}
```

| ELSE-IF | V/S | MATCH STATEMENT |
|---|---|---|
| Checks conditions one by one | | Matches a value directly |
| Comparisons (>, <, ==, etc.) | | Exact values (numbers, enums, etc.) |
| Slower (checks each condition) | | Faster (direct match) |
| Gets long with many conditions | | Cleaner for multiple cases |

# RUST LOOPS



## RUST LOOP

A loop in Rust is used to repeat a block of code multiple times until a condition is met or it is stopped manually. Rust provides three types of loops:

1) loop → Infinite loop
2) while → Runs while a condition is true
3) for → Best for iterating over ranges and collections

## LOOP (INFINITE LOOP)

- Runs forever unless stopped manually.
- Used for continuous tasks like servers or game loops.
- Use `break` to stop a loop manually

## CODE

```
fn main() {
    let mut count = 0;

    loop {
        println!("Count: {}", count);
        count += 1;
    }
}
```

## WHILE LOOP

- Runs while a condition is true.
- Best when the number of iterations is unknown.

## CODE

```
fn main() {
    let mut number = 5;

    while number > 0 {
        println!("Number: {}", number);
        number -= 1;
    }  }
```

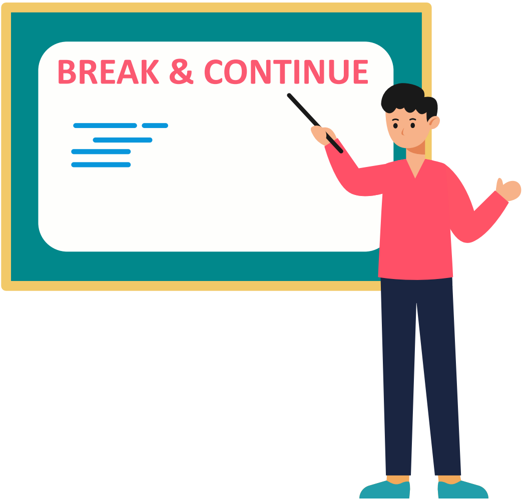
## FOR LOOP

- Used for iterating over a range or collection.
- More concise and safer than while.

## CODE

```
fn main() {
    for i in 1..=5 { // 1 to 5 (inclusive)
        println!("Value: {}", i);
    }
}
```

# BREAK & CONTINUE



## BREAK AND CONTINUE

Rust provides break and continue to control loop execution.

- break → Stops the loop immediately.
- continue → Skips the current iteration and moves to the next one.

### BREAK (STOPS THE LOOP)

- Exits the loop immediately when a condition is met.
- Used when you want to stop looping early.

### CODE

```
fn main() {
    for i in 1..=10 {
        if i == 5 {
            break; // Stops when i = 5
        }
        println!("{}", i);
    }
}
```

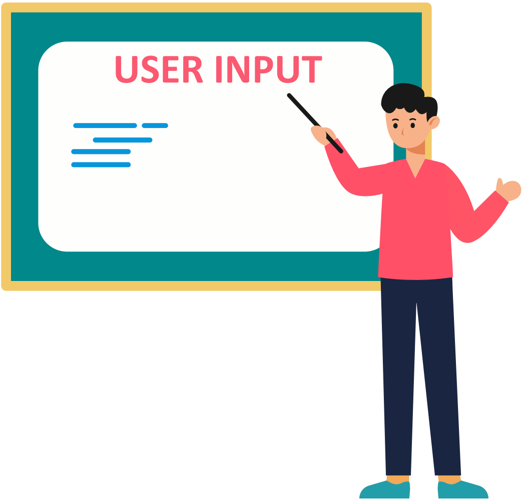
### CONTINUE (SKIPS ITERATION)

- Skips the current iteration and moves to the next one.
- Used to avoid certain values inside a loop.

### CODE

```
fn main() {
    for i in 1..=5 {
        if i == 3 {
            continue; // Skips when i = 3
        }
        println!("{}", i);
    }
}
```

# USER INPUT



## USER INPUT

User input: when a person gives data to a program through the keyboard or other devices to help the program know what to do next.

## READ A STRING INPUT

- In Rust, we can take user input using the std::io module. The most common way to read user input is through the stdin() function.

## CODE

```
use std::io;

fn main() {
    let mut input = String::new();

    println!("Enter your name:");

    io::stdin()
        .read_line(&mut input)
        .expect("Failed to read input");

    println!("Hello, {}!", input.trim());
}
```

## EXPLANATION

- **std::io :** provides functions to take user input and handle input/output operations in Rust.
- **String::new() : C**reates an empty String to store user input.
- **io::stdin().read_line(&mut input) :** Reads user input from the standard input and stores it in input.
- **.expect("Failed to read input")** : Handles errors if input reading fails.
- **input.trim()** : Removes any trailing newline or spaces

## io::stdin()

This gets the standard input, which means it allows the program to take input from the keyboard.

- It tells Rust: "I want to read something the user types."
- It creates a usable object (Stdin) that can read input.

**Simplified Meaning**

- Standard input (stdin) → A way to take user input.
- Usable object (Stdin instance) → Something that helps read what the user types.

## .read_line(&mut input)

- This takes what the user types and stores it in input.
- It uses &mut input because the input will be changed (modified).
- It waits for the user to press Enter before accepting the input.
- It saves the input as text (a String), including the Enter key (\n) at the end.

## READ DIFFERENT TYPE INPUT

- In Rust, we can take user input using the std::io module. The most common way to read user input is through the stdin() function.

## CODE

```
use std::io;

fn main() {
    let mut input = String::new();

    println!("Enter a number:");

    io::stdin()
        .read_line(&mut input)
        .expect("Failed to read input");

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

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

## EXPLANATION

- **.trim() :** Removes whitespace and the newline character.
- **.parse() :** Converts the string to a number.
- **.expect("Please enter a valid number") :** Ensures the input is valid.

## MODULES

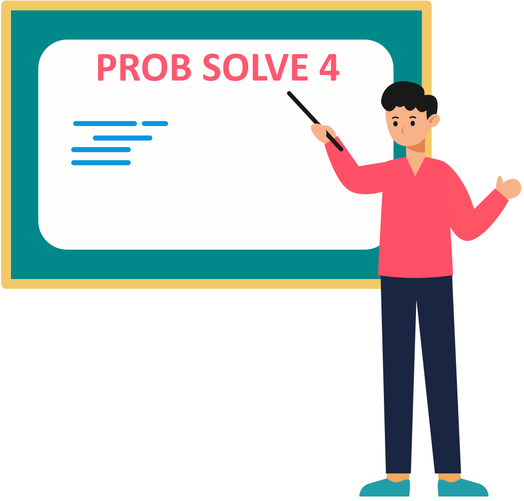
Modules in Rust group related functionalities together. They are part of the language, but you must import them to use their features.

## QUICK ANALOGY

Think of modules like tools in a toolbox. 🛠️

- The Rust language is the toolbox.
- Modules (std::io, std::fs, etc.) are different tools inside.
- You import (use std::io;) to take the tool you need and use it.

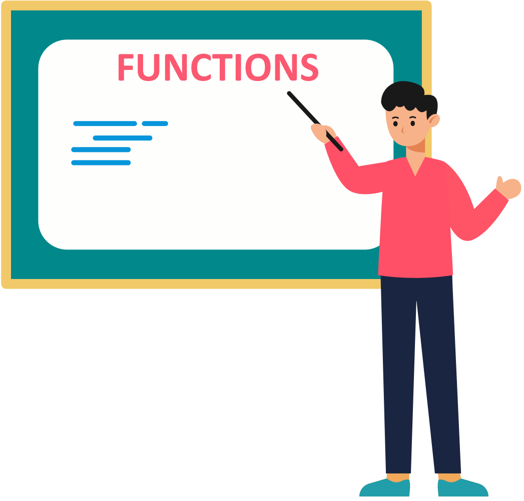
# PROB-SOLVE 4



## PROBE SOLVE 4 PRACTISE PROBLEMS

1) Write a Rust program to check if a number is positive, negative, or zero using if, else if, and else.
2) How to store the result of an if condition in a variable in Rust?
3) What is the difference between loop, while, and for in Rust?
4) Write a Rust program that prints numbers from 1 to 10 using a for loop.
5) Create an infinite loop using loop and break it after 3 iterations.
6) How does break work in a Rust loop? Write a small code snippet showing its usage.

# FUNCTIONS



## WHAT IS A FUNCTION

A function in Rust is a reusable block of code that performs a specific task.

- Helps in organizing code.
- Avoids repetition.
- Improves readability and maintainability.

## DEFINING & CALLING FUNCTION

In Rust, functions are defined using the **fn** keyword.

- fn greet() → Function definition.
- Function is called using greet(); inside main().

## CODE

```
fn greet() {
    println!("Hello, Rust!");
}

fn main() {
    greet(); // Calling the function
}
```

## PARAMETERS & ARGUMENTS

A function can take input values (parameters).
a: i32, b: i32 → Two parameters of type i32.

The values we pass while calling the function (add(5, 3);) are called arguments

## CODE

```
fn add(a: i32, b: i32) {  //  parameters
    println!("Sum: {}", a + b);
}

fn main() {
    add(5, 3); // Output: Sum: 8 & arguments
}
```

## FUNCTIONS WITH RETURN VALUES

A function can return a value using -> (return type).

-> i32 → Function returns an integer.
The last expression (without ;) is automatically returned.

## CODE

```
fn square(num: i32) -> i32 {
    num * num // No `;`, last expression is
returned
}

fn main() {
    let result = square(4);
    println!("Square: {}", result); // Output:
Square: 16
}
```

## FUNCTION MULTIPLE RETURN VALUES

A function can return multiple values using a tuple.

Returns both sum and product.
Use tuple destructuring to extract values.

## CODE

```
fn calculate(a: i32, b: i32) -> (i32, i32) {
    (a + b, a * b) // Returns a tuple (sum,
product)
}

fn main() {
    let (sum, product) = calculate(4, 5);
    println!("Sum: {}, Product: {}", sum,
product);
}
```

## INLINE FUNCTION

An inline function is marked with #[inline] to suggest the compiler to place the function's code directly at the call site, instead of calling it.

Use it for small and frequently used functions to improve performance.

## CODE

```
#[inline]
fn square(x: i32) -> i32 {
    x * x
}

fn main() {
    let result = square(4); // The compiler may
replace this call with 4 * 4
    println!("Result: {}", result);
}
```

## WHEN TO USE INLINE FUNCTION

Use #[inline] when:

- The function is small and called frequently.
- It's in a different crate or module, and you want the compiler to consider inlining it at the call site during cross-crate optimization.

## NOTES

- It's just a hint. The compiler might ignore it.
- Too much inlining = larger binary size (code bloat).
- Don't use it blindly — benchmark if you care about performance.

## INLINE ATTRIBUTES VARIENTS

#[inline] --> Suggest compiler to inline (if it thinks it's worth it)
#[inline(always)] --> Stronger hint — compiler should inline it if possible
#[inline(never)] --> Tell compiler to never inline this function

## RECURSION

When a function calls itself to solve a smaller piece of the problem until it reaches a base case.

## CODE

```
fn countdown(n: u64) {
    // Base case: stop the recursion when n
reaches 0
    if n == 0 {
        println!("{}", n); // Print 0
    } else {
        println!("{}", n); // Print the current
number
        countdown(n - 1); // Recursive call
with n - 1
    }
}

fn main() {
    let start = 5;
    println!("Starting countdown from {}:",
start);
    countdown(start);
}
```

## BASIC STRUCTURE OF RECURSION

- **Base Case**: Every recursive function must have a base case (or termination condition) to stop the recursionCountdown.

- **Recursive Case**: The function calls itself, generally with a smaller or simpler version of the original problem.

## VARIABLE SCOPE

Scope refers to the region of the code where a variable is valid and can be used.

### BASIC SCOPE

Variables defined in a block {} are only accessible inside that block.

### CODE

```
fn main() {
    let x = 5; // x comes into scope
    {
        let y = 10;
        println!("x: {}, y: {}", x, y);
    }
    println!("x: {}", x);
    // println!("y: {}", y); // ERROR: y is not
in scope here
}
```

### SHADOWING

Rust allows you to shadow a variable by re-declaring it with the same name in the same scope.

### CODE

```
fn main() {
    let x = 5;
    let x = x + 1; // shadows the previous x
    println!("x: {}", x); // prints 6
}
```

### FUNCTION SCOPE

Function scope means that variables declared inside a function are only accessible within that function and are dropped when the function ends.

### CODE

```
fn print_value() {
    let val = 100;
    println!("val: {}", val);
}

fn main() {
    // println!("{}", val); // ERROR: val is
not in scope here
    print_value();
}
```

## NOTE

- Scope defines where a variable is valid and accessible.
- Block scope: Variables are valid from the point they're declared until the end of the block {}.
- Shadowing allows reuse of variable names within the same scope.
- Function-local variables are not visible outside their function.

## RUST CLOSURES

In Rust, closures are functions without names. They are also known as anonymous functions or lambdas.

### DEFINE CLOSUERS

A closure in Rust is similar to a regular function, but it is defined inline and can capture variables from its environment.

### CODE

```
fn main() {
    let add = |a, b| a + b;
}
```

### CALL CLOSUERS

To call a closure, we use the variable name to which the closure is assigned.

### CODE

```
fn main() {
    let result = add(5, 3);
    println!("Result: {}", result);
}
```

### CLOSUERS WITHOUT PARAMETER

A closure without parameters in Rust is defined using | | syntax.

### CODE

```
fn main() {
   let greet = || println!("Hello, world!");
   greet();
}
```

### CLOSUERS AND TYPES

Rust can infer the type of the closure, but if necessary, you can specify the type explicitly.

### CODE

```
fn main() {
    let multiply = |a: i32, b: i32| a * b;
}
```

### MULTI-LINE CLOSUERS

In closures we can have multiple statements.When we want to define a multiline closure, we can use block-style syntax and explicitly return a value.

### CODE

```
fn main() {
    let process_numbers = |a, b| {
        let sum = a + b;
        let product = a * b;
        return sum + product
    };
}
```

### CLOSURE ENVIORNMENT CAPTURING

If a variable and a closure are in the same scope (like inside the main() function or any other function), the closure can capture the variable from that scope and use it.

### CODE

```
fn main() {
    let x = 10;
    let add_x = |y| x + y;
    println!("Result: {}", add_x(5));
}
```

### CLOSURE BY MOVE

The move keyword ensures the closure takes ownership of x from main() and uses it.

### CODE

```
fn main() {
    let s = String::from("Hello");

    let take_ownership = move || {
        println!("Owned string: {}", s);
    };

    take_ownership(); // Works, because `s` is
moved into the closure
    // println!("{}", s); // ERROR: `s` has
been moved and is no longer available
}
```

### EXPLICIT VS IMPLICIT RETURN TYPES

Rust allows functions to return values in two main ways: explicit and implicit return types.

### IMPLICIT RETURN

An implicit return occurs when the last expression in a function is returned without a return keyword and without a semicolon.

### CODE

```
fn add(a: i32, b: i32) -> i32 {
    a + b // 👈 implicit return
}
```

### EXPLICIT RETURN

An explicit return uses the return keyword to return a value anywhere in the function.

### CODE

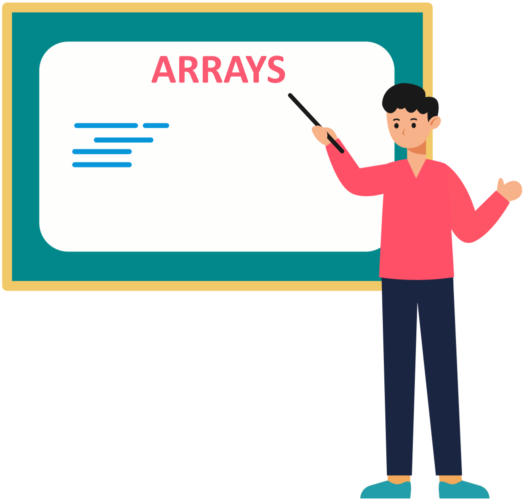
```
fn test_return() {
    println!("This will be printed
first");
    return;
    println!("This will NOT be printed");
}
```

**NOTE**

When using return keyword, the function stops immediately, and any code after the return statement will not be executed. This is useful for early exits from a function.

| IMPLICIT RETURN | VS | EXPLICIT RETURN |
|---|---|---|
| Last expression without return or semicolon | | Use of the return keyword and semicolon |
| No semicolon | | Requires a semicolon |
| Use most functions, simple and concise | | Early exits, conditional returns, |

# ARRAYS



## RUST ARRAY

An array is a list of elements of the same type.

## CREATING ARRAY IN RUST

In Rust, we can create an array in three different ways:

1. Array without data type
2. Array with data type
3. Array with default values

### ARRAY WITHOUT DATA TYPE

In Rust, we use the square brackets [] to create an array.

- numbers - name of the array
- [1, 2, 3, 4, 5] - element inside the array

### CODE

```
fn main() {
    let numbers = [1, 2, 3, 4, 5];
    println!("Arr: {:?}", numbers);
}
```

### ARRAY WITH DATA TYPE

In Rust, we can define an array with an explicit data type and length using the syntax [type; size].

- [i32; 4] – declares that the array will store 4 elements of type i32 (32-bit signed integers).
- [1, 2, 3, 4] – the actual elements stored in the array.

This ensures that:

- The array contains exactly 4 elements.
- Each element is of type i32.

### CODE

```
fn main() {
    let numbers: [i32; 4] = [1, 2, 3, 4];
    println!("array = {:?}", numbers);
}
```

### ARRAY WITH DEFAULT VALUES

In Rust, we can initialize an array with the same value repeated using the syntax [type; size] = [value; count].

- [3; 5] – means the value 3 will be repeated 5 times in the array.

### CODE

```
fn main() {
    let arr: [i32; 5] = [3; 5];
    println!("array = {:?}", arr);
}
```

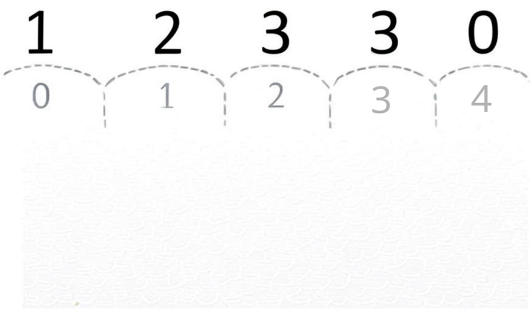
### INDEX

An index in an array is like a numbered label for each item, starting from 0. It helps you quickly find or access a specific item in the array.

Example:
In ["cat", "dog", "bird"]:

- "cat" is at index 0,
- "dog" at 1,
- "bird" at 2.

Array Indexing (Zero-Based)



### NOTE

Arrays in Rust can only store elements of the same data type, but that type can be any valid type like i32, f64, or char

### ACCESS ELEMENTS OF ARRAY

In Rust, we can access specific elements in an array using indexing with square brackets [].

- numbers[0] – accesses the first element of the array (10). Indexing starts from 0.
- numbers[2] – accesses the third element (30).

### CODE

```
fn main() {
    let numbers = [10, 20, 30, 40, 50];
    println!("First = {}", numbers[0]);
    println!("Third = {}", numbers[2]);
}
```

### MUTABLE ARRAY

In Rust, arrays are immutable by default, meaning their values cannot be changed after creation. To modify elements, we must declare the array as mutable using the mut keyword.

- mut – makes the array numbers mutable, allowing changes to its elements.
- numbers[0] = 10; – updates the first element from 1 to 10.

### CODE

```
fn main() {
    let mut numbers = [1, 2, 3, 4, 5];
    numbers[0] = 10;
    println!("Updated = {:?}", numbers);
}
```

### ITERATING AN ARRAY

In Rust, we can use a for loop to go through each element in an array one by one.

### CODE

```
fn main() {
    let numbers = [10, 20, 30, 40, 50];
    for num in numbers {
        println!("Value = {}", num);
    }
}
```

## PASSING ARRAY TO FUNCTION

In Rust, we can pass an array to a function by specifying the type and size of the array in the function parameter.

- fn print_array(arr: [i32; 5]) – function that takes an array of 5 i32 values as a parameter.

## CODE

```
fn main() {
    let numbers = [1, 2, 3, 4, 5];
    print_array(numbers);
}

fn print_array(arr: [i32; 5]) {
    println!("Array = {:?}", arr);
}
```

## RETURNING ARRAY FROM FUNCTION

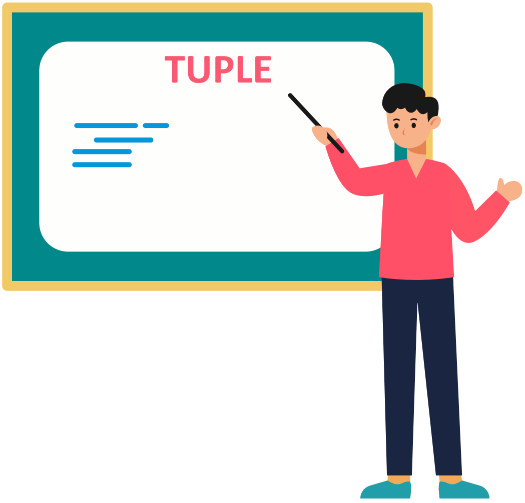
In Rust, a function can return an array by specifying the return type as an array with a fixed size.

- -> [i32; 3] – return type of the function (an array with 3 i32 elements).

## CODE

```
fn main() {
    let result = create_array();
    println!("Returned = {:?}", result);
}

fn create_array() -> [i32; 3] {
    [10, 20, 30]
}
```

# TUPLE



## RUST TUPLE

A tuple allows to store multiple values of different data types.

## CREATING TUPLE IN RUST

In Rust, we can create an tuple in two different ways:

1. Tuple without data type
2. Tuple with data type

### TUPLE WITHOUT DATA TYPE

We can create a tuple without explicitly writing the data types — Rust will automatically infer the types based on the values.

### CODE

```
fn main() {
    let n = ("Alice", 25, 5.7);
    println!("{},{},{}", n.0, n.1, n.2);
}
```

### TUPLE WITH DATA TYPE

Tuple with data types explicitly declares the type of each element in the tuple. This helps ensure type safety and makes the code more readable and predictable.

### CODE

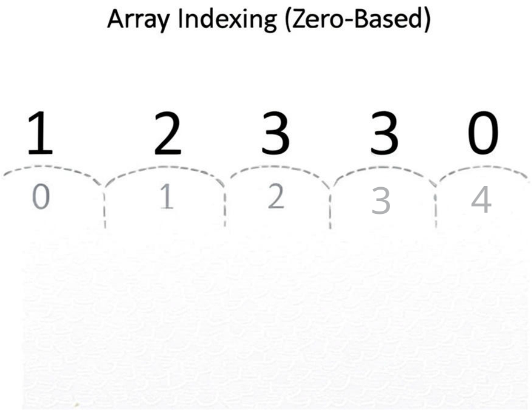
```
fn main() {
    let s: (&str, i32, f32) = ("Alice",
20, 85.5);
    println!("{}, {},{}", s.0, s.1, s.2);}
```

## INDEX

An index in a tuple is like a numbered label for each item, starting from 0. It helps you quickly find or access a specific item in the tuple

Example:
In the tuple ("cat", 2, 3.5):

- "cat" is at index 0,
- 2 is at index 1,
- 3.5 is at index 2.



## ACCESS ELEMENTS OF TUPLE

In Rust, we can access specific elements in a tuple using indexing with dot notation (.) followed by the index number.

- person.0 – accesses the first element of the tuple ("Alice").
- person.2 – accesses the third element (5.7).

## CODE

```
fn main() {
    let person = ("Alice", 25, 5.7);
    println!("First = {}", person.0);
    println!("Third = {}", person.2);
}
```

## PRINT ENTIRE TUPLE

In Rust, we can print the entire tuple using the {:?} format specifier inside the println! macro. This allows us to display all elements of the tuple at once.

- {:?} – debug format specifier Prints the whole tuple in a debug-friendly format.

## CODE

```
fn main() {
    let person = ("Alice", 25, 5.7);
    println!("Whole tuple: {:?}", person);
}
```

## MUTABLE TUPLE

In Rust, tuples are immutable by default. To modify a tuple, we use the mut keyword to make it mutable.

- person.0 = "Bob";   - Change first element
- person.1 = 30;   - Change second element

## CODE

```
fn main() {
    let mut person = ("Alice", 25, 5.7);
    person.0 = "Bob";
    person.1 = 30;

    println!("{:?}", person);
}
```

## NOTE

You can only change the element to the same type as when it was created. Changing data types is not allowed.

## DESTRUCTURING A TUPLE

We can break down tuples into smaller variables in Rust, known as destructuring.

(name, age, height) – destructures the tuple into three variables name, age, and height.

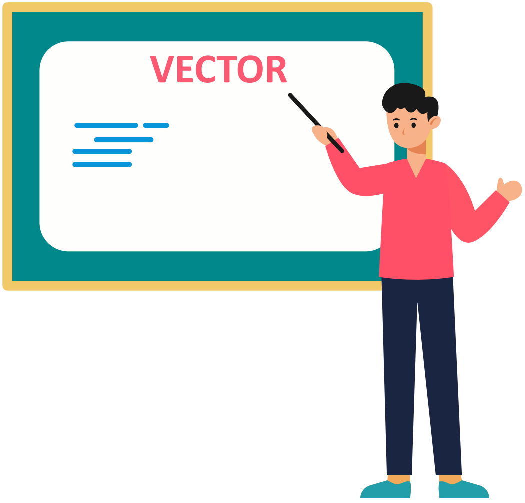
## CODE

```
fn main() {
    let person = ("Alice", 25, 5.7);
    let (name, age, height) = person;

    println!("Name: {}, Age: {}, Height:
{}", name, age, height);
}
```

## NOTE

Destructuring a tuple is also known as tuple unpacking.

# VECTOR



## RUST VECTOR

A vector is a resizable array. Unlike regular arrays, vectors can grow or shrink in size.

### CREATING VECTOR

In Rust, we can create a vector using the `vec!`

- let v - the name of the variable
- vec![1, 2, 3] - initialize a vector with integer values 1, 2, 3

### CODE

```
fn main() {
    let v = vec![1, 2, 3];
}
```

### ACCESS VECTOR

`{:?}` - debug format specifier Prints the whole vector in a debug-friendly format.

### CODE

```
fn main() {
    println!("{:?}", v);
}
```

## ACCESS ELEMENTS OF VECTOR

We can access elements of a vector using the number index.

## CODE

```
fn main() {
    let num = vec!["One", "two"];
    println!("{}", num[0]);
}
```

## ADDING ELEMENTS TO VECTOR

To add values to a vector in Rust, you can create a mutable vector by using `mut` and use the `push()` method.

## CODE

```
fn main() {
    let mut num = vec![2, 4, 6, 8, 10];
    println!("original = {:?}", num);

    num.push(12);
    num.push(14);

    println!("changed = {:?}", num);
}
```

## REMOVE LAST ELEMENT OF VECTOR

To remove the last element from a vector you can create a mutable vector by using `mut` and use the `pop`() method.

## CODE

```
fn main() {
    let mut num = vec![2, 4, 6, 8, 10];
    println!("original = {:?}", num);

    num.pop();

    println!("changed = {:?}", num);
}
```

## REMOVE ELEMENTS AT SPECIFIED INDEX

We can remove elements from a vector by making it mutable and with the `remove()` method

## CODE

```
fn main() {
    let mut num = vec![2, 4, 6, 8, 10];
    println!("original = {:?}", num);

    num.remove(2);

    println!("changed = {:?}", num);
}
```

## NOTE

Removing an element will shift all other values beyond that element by one (-1 index).

### ADD ELEMENTS AT SPECIFIED INDEX

We can add elements from a vector by making it mutable and with the `insert()` method

### CODE

```
fn main() {
    let mut num = vec![2, 4, 6, 8, 10];
    println!("original = {:?}", num);

    num.insert(0,100);

    println!("changed = {:?}", num);
}
```

### NOTE

Adding an element will shift all subsequent values one position forward (+1 index)

### LOOP THROUGH VECTOR

We can use the for loop to iterate through a vector.

### CODE

```
fn main() {
    let num = vec![2, 4, 6, 8, 10];
    for i in num {
        println!("{}", i);
    }
}
```

### EXPLICITLY DECLARE TYPE OF VECTOR

We can define vector type explicitly using the `Vec<Type>` macro.

### CODE

```
fn main() {
    let v: Vec<u8> = vec![1, 2, 3];
    println!("{:?}", v);
}
```

### EMPTY VECTOR

we can create an empty vector using the `Vec::new()` method.

Declare the variable with mut

### CODE

```
fn main() {
    let mut v: Vec<i32> = Vec::new();
    v.push(20);
    println!("{:?}", v);
}
```

## PASSING VECTOR TO FUNCTION

In Rust, we can pass a vector to a function by borrowing it using a reference (&Vec<T>) so that ownership is not moved.

- fn print_vector(v: &Vec<i32>) – function that takes a reference to a vector of i32 values as a parameter.

### CODE

```
fn main() {
    let numbers = vec![1, 2, 3, 4, 5];
    print_vector(&numbers);
}

fn print_vector(v: &Vec<i32>) {
    println!("Vector = {:?}", v);
}
```

## RETURNING VECTOR FROM FUNCTION

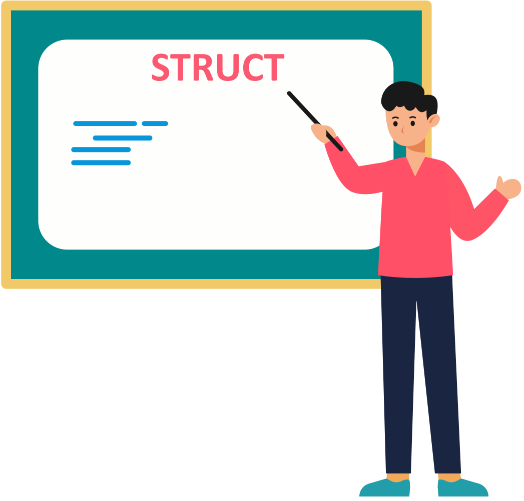
In Rust, we can return a vector from a function by specifying the return type as Vec<T> in the function signature.

- fn create_vector() -> Vec<i32> – function that returns a vector of i32 values.

### CODE

```
fn main() {
    let num = create_vector();
    println!("Returned = {:?}", num);
}

fn create_vector() -> Vec<i32> {
    vec![10, 20, 30]
}
```

# STRUCT



## RUST STRUCT

Rust structs are used to store different types of data together.

### CREATING STRUCT

In Rust, we use the `struct` keyword to define a structure.

- struct - keyword to define a structure
- Person is the name of the struct.
- It has two fields: name and age.
- String and u8 are the data types.

### CODE

```
struct Person {
    name: String,
    age: u8,
}
```

### STRUCT OBJECT

We create an object of struct so we can access the fields of the struct using dot syntax (.)

### CODE

```
fn main(){
let user = Person {
  name: String::from("John"),
  age: 35,
};

println!("Name: {}", user.name);
println!("Age: {}", user.age);
}
```

## MODIFY FIELD

To change value of struct we have to make object mutable.

## CODE

```
fn main(){
let mut user = Person {
  name: String::from("John"),
  age: 35,
};

user.age = 36; // Change value of age
println!("Name: {}", user.name);
println!("Updated age: {}", user.age);
}
```

## DESTRUCTURING FIELD STRUCTURE

Destructuring is the process of breaking down fields into smaller variables.

## CODE

```
  struct Person {
        name: String,
        age: u8,
    }

fn main() {
    let person = Person {
        name: String::from("Abhishek"),
        age: 18,
    };

    let Person { name, age } = person;

    println!("Person name = {}", name);
    println!("Person age = {}", age);
}
```

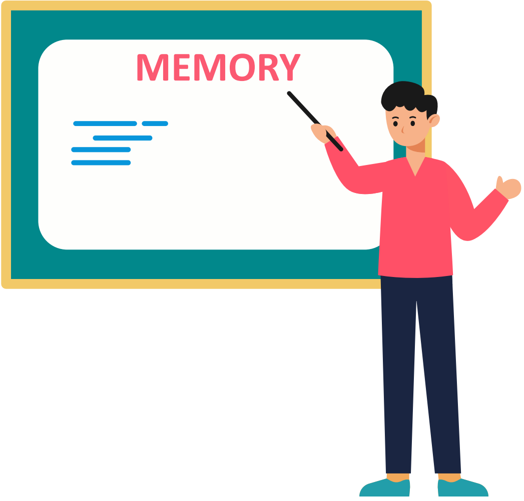
| FEATURES | TUPLE | STRUCT |
|---|---|---|
| has named fields? | no — fields use index | yes — fields have names |
| easy to understand? | less readable | more readable |
| use case | Small, quick, temporary data | Complex, meaningful, reusable data |
| access | `tuple.0`, `tuple.1` | `struct.field_name` |
| used for | `Storing simple data` | `Storing organised data` |

**NOTE**

✓ Use **tuple** for quick data storage when names aren't needed.

✓ Use **struct** when you want your data to be **clear**, **structured**, and **reusable**.

# STACK & HEAP



## MEMORY

Memory is where a computer stores data temporarily (RAM) or permanently (disk). In programming, memory is mainly used to store variables, data, and code while the program runs.

## RUST CODE, RAM, AND DISK – SIMPLE EXPLANATION

Your Rust code (like main.rs) is saved on disk — this is permanent.
When you compile, it creates a program file (like main.exe), also saved on disk.

When you run the program:

- The code is loaded into RAM so the computer can use it.
- All your variables and data also go into RAM.
- The program runs using only RAM.

After the program finishes:

- RAM is cleared.
- Your files are still saved on the disk.

## MEMORY TYPES

Memory is mainly of 2 types:

1. Stack
2. Heap

## STACK

The stack is a area of RAM used to store:

- Fixed-size data (like integers, booleans, floats)
- Function call info (local variables, return points)

It works like a stack of plates — Last-In, First-Out (LIFO):

- The last item added is the first one removed
- When a function ends, its stack data is automatically removed

## KEY POINTS:

- Fast to use and access
- Memory is freed automatically when not needed
- Has a small fixed size
- Best for simple and short-lived data

## HEAP

The heap is a area of RAM used to store:

- Dynamic or large data (e.g., String, Vec, custom types)
- Data whose size is not known at compile time

It works like a flexible storage space:

- You request memory from the heap (Rust handles this)
- Memory is not freed automatically — Rust uses ownership to manage it safely

## KEY POINTS:

- Used for data that can grow or change size
- Slower to access than the stack
- Has a larger and flexible size
- Memory is managed by Rust using ownership rules
- Best for complex or long-lived data

## DYNAMIC DATA

Dynamic data = Data that can grow or change size during program execution.
Stored in the heap because its size isn't fixed.

Examples: String, Vec, Box<T>

## STACK VS HEAP

- Stack is used for small, fixed-size data, heap is used for large or dynamic data.
- Stack is faster, heap is slower but more flexible.
- Stack memory is automatically freed, heap is manually managed (Rust uses ownership to do this safely).
- Stack has a limited size, heap can grow as needed (until RAM is full).
- Data in the stack is directly stored, in the heap it's accessed through pointers.

## STACK AND HEAP MEMORY LIMITS

Stack has a limited size:
When you run (launch) your program, the operating system sets a fixed amount of stack memory (usually 1–8 MB).
This size does not change while the program is running (during execution).
If your program uses more stack memory than this, it causes a stack overflow.

Heap can grow as needed:
The heap can use the available free RAM on your system.
For example, if your system has 4 GB RAM, and 1 GB is already used by other programs, your program’s heap can use up to about 3 GB (depending on system conditions).
Heap memory grows and shrinks dynamically as your program requests or frees memory during execution.

## STACK : STORING AND ACCESSING DATA

### STACK

- Storing: Used for fixed-size, known-at-compile-time data (e.g., integers, booleans).

- Accessing: Accessed directly by variable name because their location is known.

### CODE

```
fn main() {
    let x = 10;       // Stored on stack
    println!("{}", x);   // Direct access
}
```

## HEAP : STORING AND ACCESSING DATA

### HEAP

- Storing: Used for dynamic or large data (e.g., String, Vec) whose size can change or isn't known at compile time. Memory is allocated dynamically.

- Accessing: Accessed indirectly through pointers or smart pointers. Variables hold pointers to heap data.

### CODE

```
fn main() {
    let s = String::from("hello");
    println!("{}", s);
}
```

## IN SHORT

- You use the variable name directly (like s in println!("{}", s)), and Rust accesses the heap data for you.

- Under the hood, it's following the pointer to get the actual data.

## DATA TYPES IN STACK VS HEAP

| STACK | HEAP |
|---|---|
| `i32`, `u32`, `f32`, `f64` (numbers) | `String` → actual text stored on heap |
| `bool`, `char` | `Vec<T>` → elements stored on heap |
| `Fixed-size arrays → [i32; 3]` | `Box<T>` → value stored on heap |
| Tuples with only simple types → `(i32, bool)` | `HashMap<K, V>` → keys & values on heap |
| Simple structs (with only stack data) | Complex structs with `String`, `Vec`, etc. |
| Function call data (e.g., arguments, return info) | Data that grows or size not known at compile time |

## NOTE

Some types are split:

- Example: `String` → pointer on stack, text data on heap.

## HOW STRING IS STORED IN MEMORY (RUST)

The variable (pointer, length, capacity) is stored on the stack

- The actual string data (e.g., `"hello"`) is stored in the heap

So:

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

→ s is on the stack, `"hello"` is on the heap.

## SPLIT TUPES IN RUST

| Type | Heap Stores | Stack Stores |
|---|---|---|
| String | Text data (`"hello"`, etc.) | Pointer, length, capacity |
| `Vec<T>` | Vector elements (`T`) | Pointer, length, capacity |
| `Box<T>` | The actual value (`T`) | Pointer |
| `HashMap<K,V>` | Keys and values | Pointer to internal table |
| `Rc<T>` | Shared value | Pointer + ref count (via smart pointer) |
| `Arc<T>` | Shared + thread-safe value | Pointer + atomic ref count |

## SUMMARY

These types live partly on stack (control info) and partly on heap (actual data).
Rust handles this automatically with ownership and safety.

# OWNERSHIP

## OWNERSHIP

Ownership means each value in Rust has one owner. When the owner goes out of scope, the value is dropped.

## VARIABLE SCOPE

Scope is the part of code where a variable is valid and owns its value.

## OWNERSHIP RULES

Rust has some ownership rules.

1. Each value has one owner.
2. Only one owner at a time.
3. Value is dropped when the owner goes out of scope.

## TRANSFER OWNERSHIP

When a value is assigned to a new variable, ownership moves to it, and the old variable can't be used.

- Ownership moves from s1 to s2

## CODE

```
fn main() {
    let s1 = String::from("hello");
    let s2 = s1;

    // println!("{}", s1); // ❌
    println!("{}", s2);  // ✅ }
```

### CLONE

clone makes a deep copy of the data, so both variables own separate copies and can be used independently.

### CODE

```
fn main() {
    let s1 = String::from("hello");
    let s2 = s1.clone();

    println!("{}", s1);
    println!("{}", s2);
}
```

### COPY

Copy duplicates simple data (like integers), so both variables can be used independently without moving ownership.

### CODE

```
fn main() {
    let x = 5;
    let y = x; // x is copied, not moved

    println!("x = {}", x);
    println!("y = {}", y);
}
```

### NOTE

In ownership, the data types that use Copy are:

1. Integers (i32, u8, etc.)
2. Floating-point numbers (f32, f64)
3. Booleans (bool)
4. Characters (char)

Data types that use ownership transfer are:

1. String
2. Vec<T>
3. Most heap-allocated types and custom structs without Copy

### NOTE : ARRAY COPY VS OWNERSHIP TRANSFER

Arrays can be copied or moved depending on what they hold.

- Arrays with primitive types like [i32; N] are copied on assignment.
- Arrays with complex types like [String; N] transfer ownership (move) on assignment.

# BORROWING

## BORROWING

A way to access a value without taking ownership of it, created using the `&` symbol.

## REFERENCE

A reference is a pointer to a variable, that stores its memory address.

## POINTER

A pointer stores the memory address of another variable.

## BORROWING

- This code creates a string variable `a` and a reference `b`.
- `b` is a reference to the string value "Hello".

## CODE

```
fn main() {
    let a = String::from("Hello");
    let b = &a;
    println!("a = {}", a);
    println!("b = {}", b);
}
```

## NOTE

- b` is a reference to `a`, which means it stores the memory address of `a`.
- This way, `b` can access the value of `a` without creating a copy of it.
- There is only one actual value, which is stored in `a`, and `b` is just using the memory address of `a` to access it.

## MUTABLE REFERENCES

To change a value through a reference, you need to make the reference mut.

## CODE

```
fn main() {
    let mut name = String::from("Abhi");
    let name_ref = &mut name;
    name_ref.push_str(" loop");
    println!("{}", name_ref);
}
```

## BENEFITS OF BORROWING

- It lets you use values without taking ownership
- Using a reference is Faster because it avoids cloning
- Make programs safer and faster.

## KEY POINTS

- Borrowing doesn't take ownership
- Use &mut for mutable references

# HASH MAP

## HASH MAP

- A HashMap is a way to store data in the form of key-value pairs.

- To use a HashMap, you must import it from Rust's standard library :

```
use std::collections::HashMap;
```

## NOTE

A HashMap stores data in key-value pairs. The key is used to uniquely identify the value, which is the actual data. We use the key to efficiently access or modify the value.

## WHY USE HASH MAP

- To store data by key
- To quickly look up values
- To group related data (like names and scores)

### CREATING HASH MAP

Use `HashMap::new()` to create an empty HashMap

### CODE

```
use std::collections::HashMap;

fn main() {
    let mut scores = HashMap::new();
}
```

### ADD ELEMENTS TO THE HASH MAP

We add elements using the `.insert()` method.

- "Abhi" is the key, 10 is the value.
- You can add as many as you want
- using {:?} or {:#?} is the correct and easy way to see the whole HashMap

### CODE

```
use std::collections::HashMap;

fn main() {
    let mut scores = HashMap::new();

    scores.insert("Abhi", 10);
    scores.insert("Loop", 20);

    println!("{:?}", scores);
}
```

### IMPORTANT

.insert() will overwrite the value if the key already exists.

In .insert(key, value):

- First element is the key
- Second element is the value

Type Consistency Required:

- All keys must be the same type
- All values must be the same type

### ACCESS ELEMENTS HASH MAP

**`.get("Abhi")`** looks for the value with key **`"Abhi"`.**

It returns an Option:

- **`Some(&10)`** if found
- **`None`** if not found

**`.copied()`** converts the reference **`&10`** to the actual value 10.

**`.unwrap_or(0)`** means:

- If the key exists, get the value
- If not, use 0 as a default

So, **`score_1`** stores the value for **"Abhi"**, or **0** if **"Abhi"** is missing.

### CODE

```
use std::collections::HashMap;

fn main() {
    let mut scores = HashMap::new();

    scores.insert("Abhi", 10);
    scores.insert("Loop", 20);

    let score_1 =
scores.get("Abhi").copied().unwrap_or(0);
    println!("{}",score_1);

}
```

### FOR STRING VALUES

for string use `.cloned().unwrap_or(String::from("Unknown"));` instead of `copied().unwrap_or(default)`.

- **.cloned()** is used to turn &String into a new String
- **.unwrap_or(...)** gives "Unknown" if the key doesn’t exist

| Type | METHOD | DEFAULT VALUE |
|---|---|---|
| i32 | .copied().unwrap_or(0) | 0 |
| f64 | .copied().unwrap_or(0.0) | 0.0 |
| bool | .copied().unwrap_or(false) | false |
| char | .copied().unwrap_or(‘ ’) | ‘ ’ |
| String | .cloned().unwrap_or(String::from("Unknown")) | “unknown” |

### UPDATE VALUES

Use `.insert(key, value)` to update the value of an existing key in a HashMap. If the key exists, the old value is replaced with the new one.

### CODE

```
use std::collections::HashMap;

fn main() {
    let mut scores = HashMap::new();

    scores.insert("Abhi", 10);
    println!("{:?}",scores);

    scores.insert("Abhi", 20);
    println!("{:?}",scores);
}
```

## REMOVE VALUES

To remove a key and its value from a HashMap, use the `.remove(key)` method.

## CODE

```
use std::collections::HashMap;

fn main() {
    let mut scores = HashMap::new();

    scores.insert("Abhi", 10);
    scores.insert("Loop", 20);
    println!("{:#?}",scores);
    scores.remove("Abhi");
    println!("{:#?}",scores);
}
```

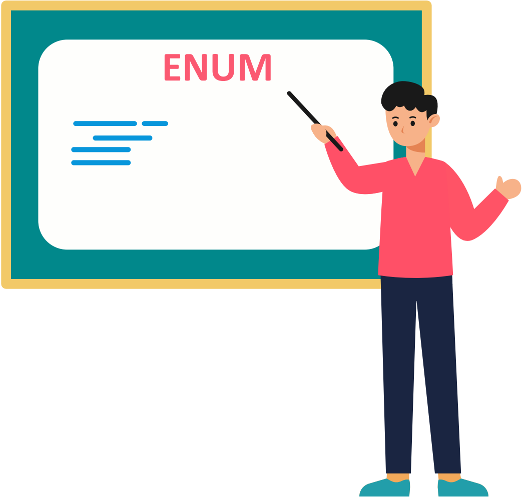
## LOOP THROUGH HASHMAP

We use a for loop to go through all key–value pairs in the HashMap.

## CODE

```
use std::collections::HashMap;

fn main() {
    let mut scores = HashMap::new();

    scores.insert("Abhi", 10);
    scores.insert("Loop", 20);

    println!("{:#?}",scores);

    for(i , x) in &scores{
        println!("For key {} value is
{}",i,x);
    }

}
```

# ENUM



## ENUM

An enum (short for enumeration) is a type that can be one of a few different values. Each value is called a variant.

### DECLARING AN ENUM

Use the **enum** keyword and list the variants (possible values) separated by commas.

- Direction is the name of the enum.
- North, South, East, and West are variants of the Direction enum.

### CODE

```
enum Direction {
    North,
    South,
    East,
    West,
}
```

### USE AN ENUM

- Create a variable and assign it one of the enum's variants.
- Use Direction::North to access the variant.

### CODE

```
fn main() {
    let go = Direction::North;
}
```

## MATCHING ON ENUM VALUES

- Use the match statement to run different code based on which variant is used.
- Each Direction::Variant is a case in the match.

### CODE

```
fn main() {
    let go = Direction::North;

    match go {
        Direction::North => println!
("Going North"),
        Direction::South => println!
("Going South"),
        Direction::East => println!("Going
East"),
        Direction::West => println!("Going
West"),
    }
}
```

## ENUMS WITH DATA

Enum variants can hold data (like strings, numbers, etc.).

### CODE

```
enum LoginStatus {
    Success(String),
    Error(String),
}
```

## USE ENUMS WITH DATA

You can assign values and match on them:

### CODE

```
fn main() {
    let result1 =
LoginStatus::Success(String::from("Welcome
, John!"));
    let result2 =
LoginStatus::Error(String::from("Incorrect
password"));

    match result1 {
        LoginStatus::Success(message) =>
println!("Success: {}", message),
        LoginStatus::Error(message) =>
println!("Error: {}", message),
    }
    match result2 {
        LoginStatus::Success(message) =>
println!("Success: {}", message),
        LoginStatus::Error(message) =>
println!("Error: {}", message),
    }
}
```

# FILE HANDLING

## FILE HANDLING

File handling means working with files, like reading, writing, creating, or deleting them using code.

### CREATE A FILE

Creating a file means making a new empty file on your computer using Rust code.

- use std::fs::File; → It lets you use the File tool to work with files
- File::create("data.txt") → Creates a file.
- .expect("creation failed") → Shows this message if file creation fails.

### CODE

```
use std::fs::File;

fn main() {
    let mut _data_file =
File::create("data.txt").expect("creation
failed");
    println!("Successfully created
data.txt");
}
```

## WRITE TO A FILE

Writing to a file means saving some text or data into a file using code.

- Use write_all(b"...") to write data into a file.
- Add use std::io::Write; to use write methods.
- File::create creates a new file or replaces the old file before writing.

## CODE

```
use std::fs::File;
use std::io::Write;

fn main() {
    let mut file =
File::create("data.txt").expect("File
creation failed");

    file.write_all(b"Hello,
Rust!").expect("Write failed");

    println!("Data written to file.");
}
```

## READ FROM A FILE

Reading from a file means getting the text or data stored inside a file using code.

- Use File::open("filename") to open a file.
- let mut file_content = String::new(); makes an empty string to store file data.
- read_to_string reads the file and converts it into a string.
- .unwrap() stops the program if an error happens.

## CODE

```
use std::fs::File;
use std::io::Read;

fn main() {
    let mut data_file =
File::open("data.txt").unwrap();

    let mut file_content = String::new();

    data_file.read_to_string(&mut
file_content).unwrap();

    println!("File content: {:?}",
file_content);
}
```

## APPEND TO A FILE

Appending to a file means adding new data to the end of an existing file without deleting the old content.

- OpenOptions::new().append(true).open("file") to open a file for appending.
- write_all(b"...") to add data at the file's end.

## CODE

```
use std::fs::OpenOptions;
use std::io::Write;

fn main() {
    let mut file = OpenOptions::new()
        .append(true)
        .open("data.txt")
        .expect("Cannot open file");

    file.write_all(b"\nThis text is
appended.")
        .expect("Write failed");

    println!("Data appended to file.");
}
```

## DELETE A FILE

Deleting a file means removing a file completely from your computer using Rust code.

- fs::remove_file("data.txt") deletes the specified file.
- .expect("...") shows an error message if it fails.

## CODE

```
use std::fs;

fn main() {

fs::remove_file("data.txt").expect("Failed
to delete file");
    println!("File deleted
successfully.");
}
```

## NOTE

- **Create**: Make a new file (empty or overwrite if it exists).
- **Write**: Save data to a file, replacing its current content.
- **Append**: Add data to the end of an existing file without deleting old content.
- **Read**: Get the content from a file into your program.
- **Delete**: Remove the file from your computer.

# ERROR HANDLING

## ERROR HANDLING

Error handling is how a program deals with problems when something goes wrong during execution.

## ERROR TYPES

- **Recoverable Errors** – things you can handle, like a missing file.
- **Unrecoverable Errors** – bugs in your program, like indexing out of bounds.

## UNRECOVERABLE ERRORS

- Unrecoverable errors stop the program immediately. They occur when something goes wrong and continuing would be unsafe or meaningless.
- Rust handles these using a macro called panic!.

## CODE

```
fn main() {
    let numbers = [10, 20, 30];
    println!("Number: {}", numbers[5]);
}
```

## PANIC

panic! does three main things:

- Shows the error message explaining what went wrong.
- Frees up resources by cleaning up (dropping variables, closing files, etc.).
- Stops the program immediately so it doesn't continue in a broken state.

## EXPLICITLY CALLING PANIC!

Code after a panic! is unreachable because panic! stops the program immediately.

## CODE

```
fn main() {
    println!("1");
    println!("2");
    panic!("Something went wrong");
    println!("3");
}
```

## QUICK NOTE

- Don't try to handle panics directly—write code that avoids panics.
- Focus on handling recoverable errors where your program can continue or recover.

## RECOVERABLE ERRORS

- Recoverable errors are errors that can be handled, and when they occur, the program does not stop.
- For example, if you try to open a file that doesn't exist, you can create the file instead of stopping the execution of the program or exiting the program with a panic.

## HANDLE RECOVERABLE ERRORS

Rust handles recoverable errors using the Result type.
You usually handle them using:

- match
- if let
- .unwrap() / .expect() (not recommended unless you're sure)

### USING MATCH

- You handle both success and error cases.
- Program does not crash even if the file doesn't exist.
- Ok(file) = success
- Err(error) = something went wrong

### CODE

```
use std::fs;

fn main() {
    let result =
fs::read_to_string("index.html");

    match result {
        Ok(content) => println!("File
content:\n{}", content),
        Err(error) => println!("Failed to
read file: {}", error),
    }
    println!("Hello, world!");

}
```

### USING IF LET

- Cleaner if you only care about success.
- You lose direct access to the error info unless you include an else if let Err(e) block.

### CODE

```
use std::fs;

fn main() {
    if let Ok(file) =
fs::read_to_string("index.html") {
        println!("File opened!{}", file);
    } else {
        println!("Could not open file.");
    }
    println!("Hello, world!");
}
```

### .UNWRAP()

This will panic if the file doesn't exist. Use only if you're 100% sure there won't be an error.

### CODE

```
use std::fs;

fn main() {
    let file =
fs::read_to_string("index.html").unwrap();
    println!("File opened!{}", file);

    println!("Hello, world!");
}
```

## .EXPECT()

- Like unwrap, but with a custom panic message
- Still panics if it fails, but helps with better debugging.

## CODE

```
use std::fs;

fn main() {
    let file =
fs::read_to_string("index.html").expect("Failed to open file");
    println!("File opened!{}", file);
    println!("Hello, world!");
}
```

## NOTE

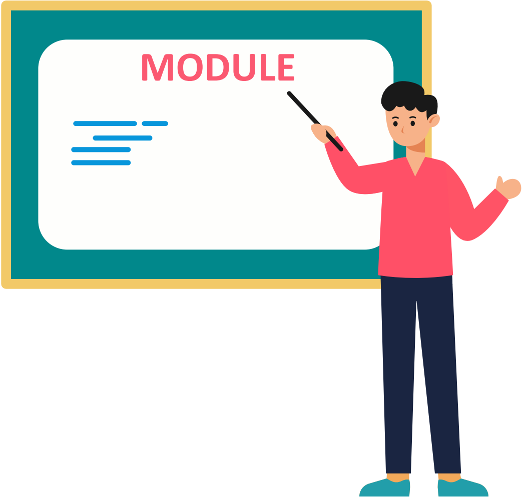
- Using match — Full Control
- Using if let — Shorter for Ok only
- .unwrap() — Only if You're Sure!
- .expect("Message") — Like unwrap, but with a custom panic message

## BEST PRACTICE

- Use match or if let for real-world apps.
- Use unwrap() / expect() only in small test programs or when you're certain nothing can go wrong.

| METHOD | SAFE | CRASHES ON ERROR? | BEST FOR |
|---|---|---|---|
| match | ✅ | ❌ | Full error handling |
| `if let` | ✅ | ❌ | Simple success check |
| `.unwrap()` | ❌ | ✅ | Quick tests (not production) |
| `.expect()` | ❌ | ✅ | Debugging with clear messages |

# MODULE



## RUST MODULE

A module in Rust is used to organize related code such as functions, structs, and enums.
It works like a folder that groups similar code together, making programs clean and manageable.

### DEFINING A MODULE

- The **mod** keyword is used to define a module
- greetings is the name of the module.
- **pub** = makes function public (accessible outside)

### CODE

```
mod greetings {
    pub fn say_hello() {
        println!("Hello from the greetings
module!");
    }
}
```

### ACCESSING MODULE

- In main(), we call it using: greetings::say_hello();
- :: = used to access module contents

### CODE

```
fn main() {
    greetings::say_hello();
}
```

### NOTE

If you don’t use **pub**, the function is private by default and can’t be used outside the module.

### SPLITTING MODULES INTO FILES

To keep your code clean and organized, you can move module code into a separate file.
This is a very common practice in real Rust projects.

### FILE STRUCTURE

```
project/
├── src/
│   ├── main.rs
│   └── greetings.rs
```

### IN MAIN.RS

```rust
mod greetings; // Load the module from greetings.rs

fn main() {
    greetings::say_hello();
}
```

### IN GREETINGS.RS

```rust
pub fn say_hello() {
    println!("Hello from another file!");
}
```

### NOTE

- The name after mod must match the filename (without .rs).
- So mod greetings; loads greetings.rs.

### IMPORTANT

- By default, everything in a module is private — you can’t access it from outside.
- Use the **pub** keyword to make a function, struct, or variable public, so it can be used outside the module.

### NESTED MODULES

- A nested module is a module inside another module.
- We access a function inside a nested module using the path: **outer::inner::function_name().**

### IN GREETINGS.RS

```
mod outer {
    pub mod inner {
        pub fn greet() {
            println!("Hello from inner
module!");
        }
    }
}

fn main() {
    outer::inner::greet();
}
```

## NESTED MODULES USING MULTIPLE FILES

### FOLDER STRUCTURE

```
src/
├── main.rs
└── library/
    ├── mod.rs      ← parent module file (required mod.rs name!)
    ├── math.rs     ← nested module
    └── utils.rs    ← nested module
```

### FILE BASED

**main.rs**

```
mod library;

fn main() {
    library::math::add(10,
5);
```

**library/mod.rs**

```
pub mod math;
```

**library/math.rs**

```
pub fn add(a: i32, b: i32) {
    println!("Sum: {}", a +
b);
}
```

### IMPORTANT

1. If a module is a folder with submodules, it must have a **mod.rs** file.
2. You can use any name for the inner module files like math.rs, utils.rs.
3. **main.rs** must be outside the folder where **mod.rs** and inner files live.
4. Use mod modname; in main.rs to bring the module into scope.
5. Use pub mod xyz; inside mod.rs or parent file to expose nested modules.

## FLAT FILE MODULE STYLE

You can use modname.rs (like library.rs) instead of mod.rs.
Cleaner structure, recommended in modern Rust projects.
Just creat same file and folder names.

## FOLDER STRUCTURE

```
src/
├── main.rs
├── library.rs     ← replaces mod.rs
└── library/
    ├── math.rs
    └── utils.rs
```

## library.rs

```
pub mod math;
pub mod utils;
```

## USE KEYWORD

The use keyword is used to bring modules, functions, structs, or enums into scope, so you don’t have to write their full path every time.

## CODE

```
mod outer {
    pub mod inner {
        pub fn greet() {
            println!("Hello!");
        }
    }
}

use outer::inner::greet;

fn main() {
    greet();
}
```

## AS KEYWORD

- Use **as** with **use** to rename an imported item and avoid name conflicts.
- Syntax: **use path::to::item as new_name;**

## CODE

```
mod database {
    pub fn connect() {
        println!("Connected to the
database!");
    }
}

mod network {
    pub fn connect() {
        println!("Connected to the
network!");
    }
}

// Use aliasing to avoid name conflict
use database::connect as db_connect;
use network::connect as net_connect;
```

```
fn main() {
    db_connect();
    net_connect();
}
```

# CRATE & PACKAGE

## CRATE

A crate is a collection of Rust code that is compiled by the Rust compiler as a single unit. Every Rust program is a crate.

## CRATE TYPES

Rust has 2 main types of crates:

1. Binary crate
2. Library crate

### BINARY CRATE

- A binary crate is a crate that can run as a program.
- It must have a main() function — the starting point.
- It compiles into an executable binary and runs it.

### CODE

```
fn main() {
    println!("I am a binary crate!");
}
```

## LIBRARY CRATE

- A library crate is used to define functions, types, logic, etc.
- It does NOT have a main() function — so it can't be run directly.
- It's meant to be used by other crates.

## CODE

```
pub fn greet() {
    println!("Hello from the library 
crate!");
}
```

## PACKAGE

A package is a bundle that contains one or more crates.
Usually, a package contains:

- Exactly one library crate (optional), and
- Zero or more binary crates.
- So, a package can contain one or multiple crates, but never zero crates.

## CREATE A NEW PACKAGE

- This creates a new directory called my_package with:
- A Cargo.toml file (the package manifest)
- A src folder with a main.rs file (for a binary crate)

## COMMAND

```
cargo new my_package
```

## EXPLORE THE CREATED PACKAGE

- Cargo.toml contains package metadata.
- src/main.rs is the main source file for your binary crate.

## FILE STRUCTURE

```
my_package/
├── Cargo.toml
└── src/
    └── main.rs
```

## BUILD AND RUN THE PACKAGE

In the `my_package` directory, run: `cargo build`

- This compiles your package.

To run the executable: `cargo run`

| CREATING A LIBRARY PACKAGE | COMMAND |
|---|---|
| • This creates a new directory called my_lib_package | `cargo new my_lib_package --lib` |

# TESTING

## TESTING

Testing in Rust means writing special macro to automatically check if your code works correctly.

### WRITE TEST

- `#[cfg(test)]` - only compiles when testing
- `use super::*;` - brings function into scope
- `#[test]` - an attribute that tells Rust "this function is a test".

### CODE

```
pub fn add(a: i32, b: i32) -> i32 {
    a + b
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_add() {
      // Write assert macros here
    }
}
```

## RUN TEST

In terminal (inside project folder), run:

```
cargo test
```

## NOTE

Write functions and test macros inside lib.rs, and then run all tests using cargo test.

## ASSERT MACROS

Assert macros are built-in tools in Rust used to check that your program behaves the way you expect during testing.

## MOST USED ASSERT MACRO

| MACRO | USE FOR |
|---|---|
| assert!() | Check a condition is true. |
| assert_eq!() | Check two values are equal. |
| assert_ne!() | Check two values are not equal. |
| #[should_panic] | This is not a macro but an attribute. Used when the test is supposed to panic. |

# CARGO

## CARGO

Cargo is Rust's package manager and build system. It:

- Manages dependencies.
- Builds package.
- Runs tests.
- Generates docs.

## CREATE A NEW PACKAGE

Click here to go to that page

## BUILDING AND RUNNING THE PACKAGE

Click here to go to that page

## TESTING CODE

Click here to go to that page

### DEPENDENCY MANAGEMENT

Dependencies are external libraries (crates) that a Rust project relies on, listed in the Cargo.toml file. You can find and download them from ***crates.io*** and Cargo manages them for you.

### ADD DEPENDENCY

You can add dependencies in two ways:

### OPTION 1: MANUALLY EDIT Cargo.toml

```
[dependencies]
rand = "0.8"
```

### OPTION 2: USE THE TERMINAL (RECOMMENDED)

```
cargo add rand
```

This automatically edits Cargo.toml for you and adds the latest compatible version.

### BUILD THE PROJECT

- Build the Project to Download Dependencies.
- Once you've added a dependency, let Cargo fetch it.
- This compiles your code and downloads the crate from crates.io

### COMMAND

```
cargo build
```

### USE THE DEPENDENCY IN YOUR CODE

To use in your code you need to add rand to src/main.rs

### CODE

```
use rand::Rng;

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

    let mut rng = rand::rng();
    let i: i32 = rng.random();
    println!("{}", i);
println!("{}", rng.random_range(1..=5));
}
```

# MACRO

## MACRO

A macro in Rust is a way to write code that writes other code (metaprogramming)

## MACRO TYPES

Rust has two types of macros:

1) Declarative macros (macro_rules!) – easier, most common.
2) Procedural macros – more powerful, for advanced use cases (we'll ignore these for now).

## DECLARE MACRO

- **`macro_rules!`** say_hello declares a macro named say_hello.
- **`()`** means it takes no arguments.
- The **=>** part defines what code will be generated when the macro is used.
- **`say_hello!();`** is how we invoke it.

## CODE

```
macro_rules! say_hello {
    () => {
        println!("Hello from macro!");
    };
}

fn main() {
    say_hello!(); // Calls the macro
}
```

### MACRO WITH ARGUMENTS

- **`$name:expr`** means "match an expression and call it $name".
- The macro uses that **`$name`** inside the println!.

### CODE

```
macro_rules! greet {
    ($name:expr) => {
        println!("Hello, {}!", $name);
    };
}

fn main() {
    greet!("Alice");
    greet!("Bob");
}
```

### MULTIPLE PATTERNS

Now the macro works both with and without arguments!

### CODE

```
macro_rules! test {
    () => {
        println!("No input!");
    };
    ($x:expr) => {
        println!("You said: {}", $x);
    };
}

fn main() {
    test!();
    test!("something");
}
```

### REPEATING INPUTS

- macros accept any number of inputs using **`$(...)*`**
- **`$( $x:expr ),*`** matches 0 or more comma-separated expressions.
- **`$( ... )*`** means: repeat this for each match.

### CODE

```
macro_rules! print_numbers {
    ( $( $num:expr ),* ) => {
        $(
            println!("{}", $num);
        )*
    };
}

fn main() {
    print_numbers!(10, 20, 30);
}
```

### HOW MACROS WORK

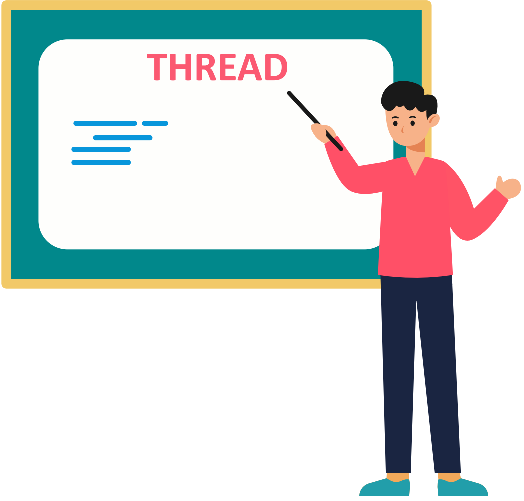
- Macros don't run at runtime — they work at compile time.
- When you call a macro, Rust does the following at compile time:
- "Hey Rust, when you see **`say_hi!()`**, replace it with **`println!("Hi!")`**."
- The macro call is replaced with the macro's logic before the program is compiled.

### CODE

```
macro_rules! say_hi {
    () => {
        println!("Hi!");
    };
}

fn main() {
    say_hi!();  //replaced by println! ("Hi!") at compile time

}
```

| Point | Macros | Functions |
|---|---|---|
| When it works | At compile time | At runtime |
| Syntax | Uses ! (e.g., `println!()`) | Normal call (e.g., `add(2, 3)`) |
| Purpose | For writing repetitive code automatically | For writing reusable logic |
| Type checking | Happens after macro expands (can be confusing) | Happens during definition/use (easier) |
| When to use | Use when you want to generate flexible or repeated code | Use for clear, reusable, type-safe logic |

# THREAD



## THREAD

A thread allows a program to run different parts of code at the same time.

### CREATING A THREAD

The **`thread::spawn`** function is used to create a new thread. The spawn function takes a closure as parameter. The closure defines code that should be executed by the thread.

- **`thread::spawn`** → starts a new thread.
- closure **`|| { ... }`** → the code that runs inside the new thread.

### CODE

```
use std::thread;
use std::time::Duration;

fn main() {
    //create a new thread
    thread::spawn(|| {
        for i in 1..10 {
            println!("hi number {} from the
spawned thread!", i);

thread::sleep(Duration::from_millis(1));
        }
    });

    //code executed by the main thread
    for i in 1..5 {
        println!("hi number {} from the main
thread!", i);

thread::sleep(Duration::from_millis(1));
    }
}
```

### NOTE

- The new thread will be stopped when the main thread ends. The output from this program might be a little different every time.
- Both threads run at the same time, so the output will be mixed.

### JOIN THREAD

**`handle.join().unwrap()`** → makes the main program wait until the thread finishes.

### CODE

```
use std::thread;
use std::time::Duration;

fn main() {
   let handle = thread::spawn(|| {
      for i in 1..10 {
         println!("hi number {} from the
spawned thread!", i);

thread::sleep(Duration::from_millis(1));
      }
   });
   for i in 1..5 {
      println!("hi number {} from the main
thread!", i);

thread::sleep(Duration::from_millis(1));
   }
   handle.join().unwrap();
}
```

### CREATING MULTIPLE THREAD

We can create more than one thread by calling **`thread::spawn()`** multiple times.

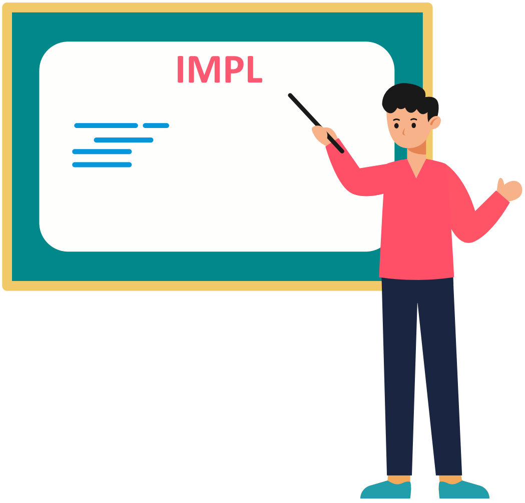
### CODE

```
use std::thread;
use std::time::Duration;

fn main() {
    // first new thread
    let t1 = thread::spawn(|| {
        for i in 1..=3 {
            println!("Thread 1: {i}");

thread::sleep(Duration::from_millis(200));
        }
    });

    // second new thread
    let t2 = thread::spawn(|| {
        for i in 1..=3 {
            println!("Thread 2: {i}");

thread::sleep(Duration::from_millis(200));
        }
    });

    // main thread
    for i in 1..=3 {
        println!("Main thread: {i}");

thread::sleep(Duration::from_millis(200));
    }

    // wait for threads to finish
    t1.join().unwrap();
    t2.join().unwrap();
}
```

**NOTE**

- Threads start together, but order is unpredictable → output is mixed because the OS decides which one runs first(including main).
- If you don't use **`join()`**, the main thread may finish first, killing other threads early.
- Always use **`join()`** if you want all threads to complete.
- Even if you don't create any extra threads, your program is already running inside one thread, called the main thread. The main function always runs in the main thread by default.

# IMPL



## IMPL

impl stands for "implementation". It is used to define functions for a struct, enum, or trait.

### &SELF – IMMUTABLE BORROW

- We use the **impl** keyword to define functions for a struct.
- We can read data, but not change it.

### CODE

```
struct Rectangle {
    width: u32,
    height: u32,
}

impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}

fn main() {
    let rect = Rectangle {
        width: 10,
        height: 5,
    };

    println!("Area: {}", rect.area());
}
```

### &MUT SELF – MUTABLE BORROW

- We can change the struct's data.

### CODE

```
struct Rectangle {
    width: u32,
    height: u32,
}

impl Rectangle {
    fn double_size(&mut self) {
        self.width *= 2;
        self.height *= 2;
    }
}

fn main() {
    let mut rect = Rectangle {
        width: 4,
        height: 3,
    };

    rect.double_size();
    println!("Doubled size: {} x {}",
rect.width, rect.height);
}
```

### SELF – TAKE OWNERSHIP

- We take ownership of the struct. It gets moved.
- We use self to destroy data

### CODE

```
struct Rectangle {
    width: u32,
    height: u32,
}

impl Rectangle {
    fn destroy(self) {
        println!("Destroying rectangle: {}
x {}", self.width, self.height);
    }
}

fn main() {
    let rect = Rectangle {
        width: 6,
        height: 2,
    };

    rect.destroy();

    // rect can’t be used here anymore
    // println!("{}", rect.width); //
error: value moved
}
```

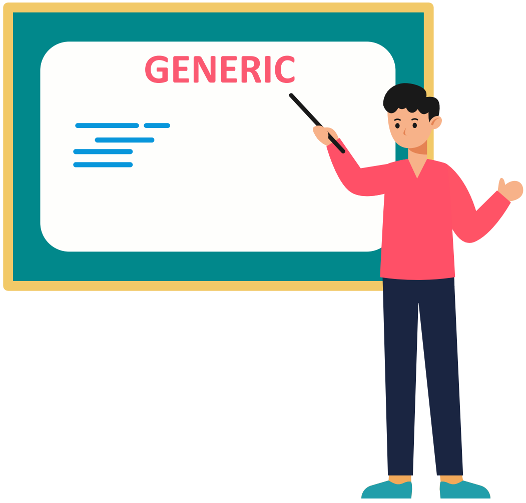
## ASSOCIATED FUNCTION

- An associated function is a function that is defined inside an impl block, but does not take self, &self, or &mut self.
- Used to create new instances of a struct.
- To Add Custom Logic When Creating
- Keeps code clean and reusable.
- You call it on the type itself, not on an instance.
- fn new(width: u32, height: u32) -> Self
- This says: "Take two numbers, and return a new Rectangle."
- Inside: Self { width, height }
- Same as writing Rectangle { width, height }

## CODE

```
struct Rectangle {
    width: u32,
    height: u32,
}

impl Rectangle {
    fn new(width: u32, height: u32) ->
Self {
        Self { width, height }
    }
}

impl Rectangle {
    fn new_strict(width: u32, height: u32)
-> Self {
        if width == 0 || height == 0 {
            panic!("Size must be greater
than 0");
        }
        Self { width, height }
    }
}

fn main() {
    let rect = Rectangle::new(7, 3);
    println!("Created rectangle: {} x {}",
rect.width, rect.height);
}
```

# GENERIC



## GENERIC

Generics allow you to write code that works with any data type.

### BASIC GENERIC FUNCTION

- We use <T> before a function to define a generic type.
- T is a placeholder for any type that will be decided when the function is called.

### CODE

```
fn identity<T>(value: T) -> T {
    value
}

fn main() {
    let a = identity(5);
    let b = identity("hello");
    let c = identity(3.14);

    println!("{}, {}, {}", a, b, c);
}
```

### MULTIPLE GENERIC FUNCTION

- We write <T, U> to use two generic types.
- T: Display means T should be printable.
- use std::fmt::Display; is needed to use the Display trait.
- We can use any name for generic types (T, U, X, A, etc.).
- This function prints both values using {}.

### CODE

```
use std::fmt::Display;

fn combine<T: Display, U: Display>(a: T,
b: U) {
    println!("a: {}, b: {}", a, b);
}

fn main() {
    combine(42, "hello");
    combine(true, 3.14);
}
```

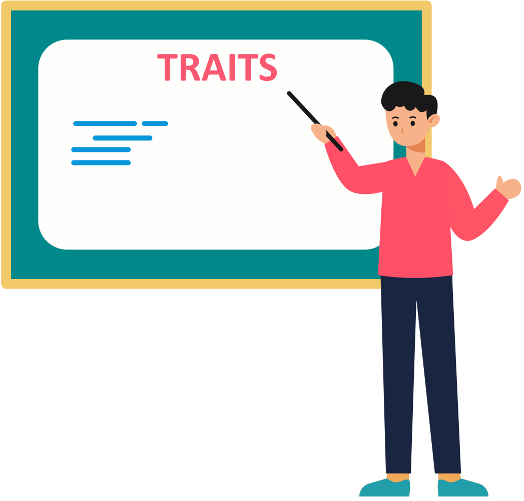
### BASIC GENERIC STRUCT

- struct Point<T> means the struct works with any type T
- One generic parameter <T>: All fields using T must be the same type when you create an instance.
- Multiple generic parameters <T, U>: Each field can be a different type.

### CODE

```
struct Point<T , U> {
    x: T,
    y: U,
}

fn main() {
    let int_point = Point { x: 1, y: 2 };
    let float_point = Point { x: 1.1, y:
"Abhishek" };

    println!("int_point: {}",
int_point.x);
    println!("float_point: {}",
float_point.x);
    println!("int_point: {}",
int_point.y);
    println!("float_point: {}",
float_point.y);
}
```

# TRAITS



## TRAITS

A trait in Rust is like a set of rules. If a type implements a trait, it must follow its rules.

### DEFINE A TRAIT

- If you implement this, you must have a **speak()** function.
- Use **trait** keyword to define a trait.

### CODE

```
trait Speak {
    fn speak(&self);
}
```

## IMPLEMENT TRAIT FOR STRUCT

- Now we teach Dog and Cat how to speak(), using their name field.

## CODE

```
struct Dog {
    name: String,
}

struct Cat {
    name: String,
}

impl Speak for Dog {
    fn speak(&self) {
        println!("{} says: Woof!",
self.name);
    }
}

impl Speak for Cat {
    fn speak(&self) {
        println!("{} says: Meow!",
self.name);
    }
}
```

## USE IN MAIN

- Create your animals and call speak() on them!

## CODE

```
fn main() {
    let dog = Dog {
        name: String::from("Buddy"),
    };

    let cat = Cat {
        name: String::from("Whiskers"),
    };

    dog.speak();
    cat.speak();
}
```

## NOTE

You can also create a separate impl block for a struct, where you write other functions not related to any trait.

# RESULT

## RESULT

- Result is enum in rust which is used for error handling.
- It has two parts called "variants :   Result<T, E>
  - Ok(T) → everything went fine, and you get a value T.
  - Err(E) → something went wrong, and you get an error E.

## USE RESULT

- match - To check what kind of result
- result.unwrap(); - Get the value or crash.
- result.unwrap_err(); - Get the error or crash.
- result.is_ok(); - Returns true if it's Ok.
- result.is_err(); - Returns true if it's Err.

## CODE

```
fn divide(a: i32, b: i32) -> Result<i32,
String> {
    if b == 0 {
        Err("Can't divide by
zero!".to_string())
    } else {
        Ok(a / b)
    }
}
fn main() {
    let result = divide(10, 2);

    match result {
        Ok(answer) => println!("Answer is
{}", answer),
        Err(error) => println!("Error:
{}", error), }}
```

# Asynchronous

## SYNCHRONOUS PROGRAMMING

Synchronous programming means tasks are done one at a time. The next task starts only after the first one is done.

## ASYNCHRONOUS PROGRAMMING

Asynchronous programming means tasks can be done at the same time. The next task can start before the first one is done.

## ASYNCHRONOUS PROGRAMMING EXAMPLE

Asynchronous programming is like boiling water — instead of waiting the full 15 minutes for it to boil, you can use that time to chop vegetables. You don't have to wait for one task to finish before starting another.

### What is async

- **async** means "this function runs asynchronously — it doesn't block the program."
- It returns a Future. The function won't actually run until you **.await** it.

### CODE

```
async fn my_function() {
    // do something
}
```

### What is await

- **.await** is used to wait for an async function to finish.
- This tells Rust: “Wait for the result of this async task, then move on.”
- Without **.await**, the function won't actually run — it just builds the plan (the "Future").

### CODE

```
my_function().await;
```

### What is tokio

- tokio is an async runtime for Rust.
- It helps Rust run and schedule async code behind the scenes.
- Without a runtime like tokio, your async code won’t work — because Rust by default doesn’t know how to "run" async tasks.

### CODE

```
#[tokio::main]
async fn main() {
    // your async code runs here
}
```

### NOTE

- We'll use tokio because it's the most used runtime in Rust for async.
- In Cargo.toml, add:
  - `[dependencies]`
  - `tokio = { version = "1", features = ["full"] }`

### CREATE ASYNC FUNCTION

- We create async function using async keword
- In real async code, functions usually return something — and they often can fail.
- It returns a Result
- .await - We wait for fetch_data to finish

### CODE

```
#[tokio::main]
async fn main() {
    let result = fetch_data().await;

    match result {
        Ok(data) => println!("Got data:
{}", data),
        Err(e) => println!("Error: {}",
e),
    }
}

async fn fetch_data() -> Result<String,
String> {
    // Simulate success
    let success = true;

    if success {
        Ok("Data from server".to_string())
    } else {
        Err("Failed to fetch
data".to_string())
    }
}
```

### SIMULATE DELAY

- We Simulate Delay Using tokio::time::sleep
- In real async programs (e.g. network or file I/O), things don't return instantly — they wait, but in async we don't block the program.
- sleep(Duration::from_secs(2)) - Wait 2 seconds
- You'll use this when working with real APIs or databases

### CODE

```
use tokio::time::{sleep, Duration};

#[tokio::main]
async fn main() {
    println!("Fetching data...");

    let result = fetch_data().await;

    match result {
        Ok(data) => println!("Got data:
{}", data),
        Err(e) => println!("Error: {}",
e),
    }
}

async fn fetch_data() -> Result<String,
String> {
    // Wait 2 seconds without blocking
    sleep(Duration::from_secs(2)).await;

    Ok("Data from server".to_string())
}
```

## RUNNING MULTIPLE ASYNC TASKS

### Run Async Tasks Sequentially

- One task waits for the previous to finish.
- Even though each task takes 2 seconds, the total is 4 seconds because they run one after another.

### CODE

```
use tokio::time::{sleep, Duration};

#[tokio::main]
async fn main() {
    let start = std::time::Instant::now();

    fetch_data(1).await;
    fetch_data(2).await;

    let duration = start.elapsed();
    println!("Total time: {:?}",
duration);
}

async fn fetch_data(id: u32) {
    println!("Task {} started", id);
    sleep(Duration::from_secs(2)).await;
    println!("Task {} finished", id);
}
```

### Run Async Tasks In Parallel (with join!)

- We run both tasks at the same time.
- Both tasks start together and finish in 2 seconds — that's true concurrency.

### CODE

```
use tokio::{time::{sleep, Duration},
join};

#[tokio::main]
async fn main() {
    let start = std::time::Instant::now();

    join!(
        fetch_data(1),
        fetch_data(2)
    );

    let duration = start.elapsed();
    println!("Total time: {:?}",
duration);
}

async fn fetch_data(id: u32) {
    println!("Task {} started", id);
    sleep(Duration::from_secs(2)).await;
    println!("Task {} finished", id);
}
```

### NOTE

- **let start = std::time::Instant::now();** -> Captures the current time — a starting point to measure how long something takes. Think of it like pressing the start button on a stopwatch.
- **let duration = start.elapsed();** -> Calculates how much time has passed since the start.
- **sleep(Duration::from_secs(2)).await;** -> Pauses the async task for 2 seconds, but doesn't block the whole program.
- **Duration::from_secs(2)** -> creates a 2-second delay.
- **join!(fetch_data(1), fetch_data(2));** -> Runs both async tasks in parallel, and waits until both are done.

### Run Async Tasks in the Background

when we use .await or join!, our program waits for the tasks to finish.
But sometimes, you want to:
Start a task
Let it run in the background
Keep doing other things without waiting
That's what tokio::spawn is for.

tokio::spawn(async { ... }) -> Start a background task
sleep(3).await in main -> Let the background task have time to finish
Without that final sleep(3), the program might exit before the background task finishes.

### CODE

```
use tokio::time::{sleep, Duration};

#[tokio::main]
async fn main() {
    println!("Start");

    // Spawn a background task
    tokio::spawn(async {

sleep(Duration::from_secs(2)).await;
        println!("Background task done");
    });

    println!("Main function continues");

    // Give enough time for the background
task to finish
    sleep(Duration::from_secs(3)).await;

    println!("Main function done");
}
```

### IMPORTANT NOTE

- tokio::spawn(...) returns a JoinHandle, which you can .await if you want the result.
- If you don't .await it, the task runs independently.

## Use .await on spawned task

## CODE

```
use tokio::time::{sleep, Duration};

#[tokio::main]
async fn main() {
    let handle = tokio::spawn(async {

sleep(Duration::from_secs(2)).await;
        "Result from background task"
    });

    println!("Waiting for background
task...");

    let result = handle.await.unwrap(); //
Wait for it to finish
    println!("Got result: {}", result);
}
```

## Handling Errors in Async Functions

When working with async code (APIs, files, DB), things can go wrong:
Network might fail

File might not exist

Data might be invalid

async fn -> Result<...> -> Async function that may fail
match get_data(...).await -> Handle success or error

## CODE

```
use tokio::time::{sleep, Duration};

#[tokio::main]
async fn main() {
    match get_data(true).await {
        Ok(data) => println!("Success:
{}", data),
        Err(err) => println!("Error: {}",
err),
    }

    match get_data(false).await {
        Ok(data) => println!("Success:
{}", data),
        Err(err) => println!("Error: {}",
err),
    }
}

async fn get_data(success: bool) ->
Result<String, String> {
    sleep(Duration::from_secs(1)).await;

    if success {
        Ok("Here is your
data".to_string())
    } else {
        Err("Something went
wrong".to_string())
    }
}
```

### Async HTTP Request with reqwest

Add dependencies in Cargo.toml

```
[dependencies]
tokio = { version = "1", features = ["full"] }
reqwest = { version = "0.11", features = ["json"] }
```

reqwest::get(url).await -> Send an async GET request

.await? -> Wait and return error if failed

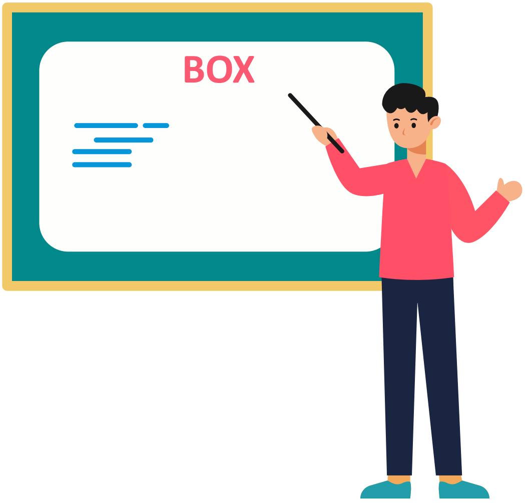
response.text().await? -> Convert response to string

Result<String, reqwest::Error> -> Function might fail (e.g. no internet)

### CODE

```
use reqwest;
use tokio;

#[tokio::main]
async fn main() {
    match fetch_data().await {
        Ok(data) => println!("Response:\
n{}", data),
        Err(e) => println!("Error: {}",
e),
    }
}

async fn fetch_data() -> Result<String,
reqwest::Error> {
    let url =
"https://jsonplaceholder.typicode.com/post
s/1";

    let response =
reqwest::get(url).await?;             //
Send GET request
    let body = response.text().await?;
// Get response body as text

    Ok(body)
}
```

# BOX



## BOX

Box is a smart pointer that lets you store data on the heap instead of the stack.

**Stack** = fast memory for small, fixed-size data.

**Heap** = memory for big or dynamically-sized data.

## WHEN TO USE BOX

- You have large data that you don't want to copy.
- You need a known size type (e.g. recursive types like linked lists).
- You just want to own data on the heap.

## CREATING AND USING A BOX

- x is a normal integer on the stack.
- y is a Box<i32>. The integer value (5) is on the heap.

## CODE

```
fn main() {
    let x = 5;
    let y = Box::new(5);
    println!("x = {}, y = {}", x, y);
}
```

## DEREFERENCING A BOX

- To get the value inside the Box, you can use the * operator.
- *b means "go to the heap and get the value."

## CODE

```
fn main() {
    let b = Box::new(10);
    println!("b = {}", *b); // prints 10
}
```

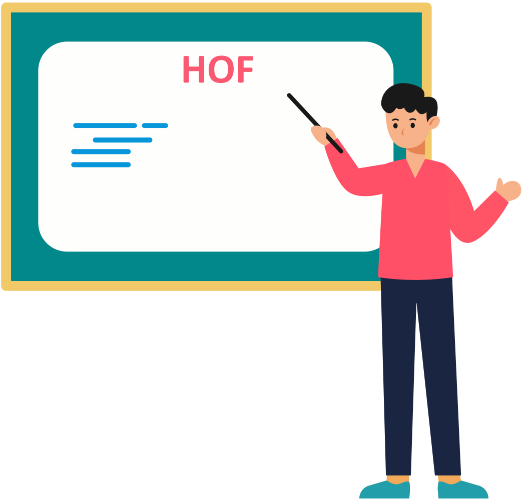
## BOX WITH STRUCTS

- You can put structs inside boxes too.

## CODE

```
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let p = Box::new(Point { x: 1, y:
2 });
    println!("Point: ({}, {})", p.x, p.y);
}
```

# HOF



## HIGH ORDER FUNCTION

A Higher-Order Function (HOF) is a function that does one (or both) of the following:

1) Takes another function as a parameter, or
2) Returns a function as its result.

### PASSING A FUNCTION TO ANOTHER FUNCTION

- add_one → a normal function.
- apply_function → takes a function (f) and a value, then calls f(value).

### CODE

```
fn add_one(x: i32) → i32 {
    x + 1
}

fn apply_function(f: fn(i32) → i32,
value: i32) → i32 {
    f(value)
}

fn main() {
    let result = apply_function(add_one,
5);
    println!("Result: {}", result);
}
```

## USING CLOSURES AS FUNCTIONS

- square → closure that squares a number.
- apply_function → calls the passed function with a given value.
- main → passes `square` and `4` to `apply_function` and prints the result.

## CODE

```
fn main() {
    let square = |x: i32| x * x;

    let result = apply_function(square,
4);
    println!("Result: {}", result);
}

fn apply_function(f: fn(i32) -> i32,
value: i32) -> i32 {
    f(value)
}
```

## RETURNING A FUNCTION

- add_one → a normal function that takes one i32 and returns an i32.
- get_adder → returns a function pointer of type fn(i32) -> i32.
- main → calls get_adder(), stores the returned function in f, and then calls f(5).

## CODE

```
fn add_one(x: i32) -> i32 {
    x + 1
}

fn get_adder() -> fn(i32) -> i32 {
    add_one
}

fn main() {
    let f = get_adder();
    let result = f(5);
    println!("Result: {}", result);
}
```