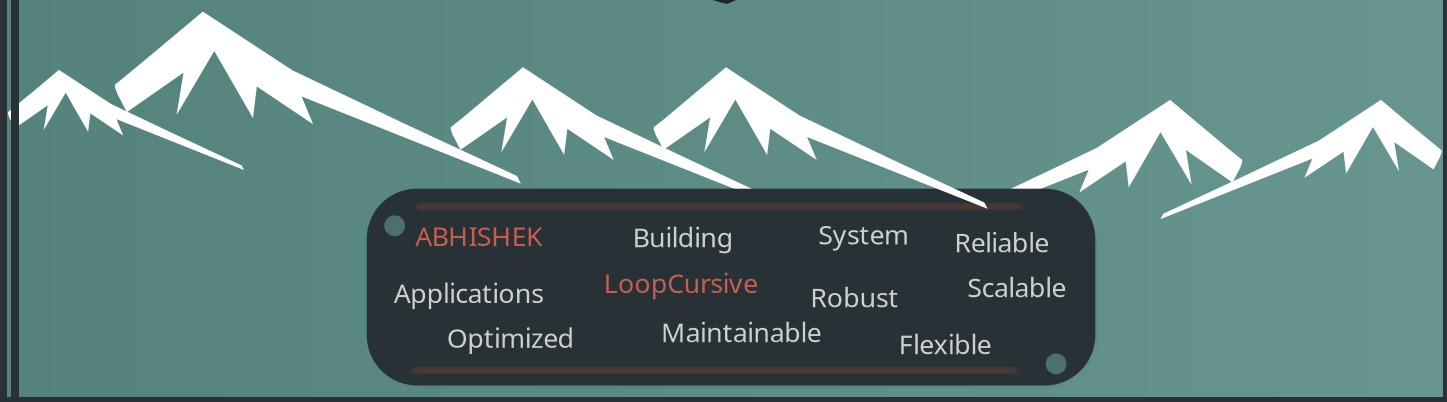
MASTERING THE RUST LANGUAGE

A LoopCursive Book



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
```

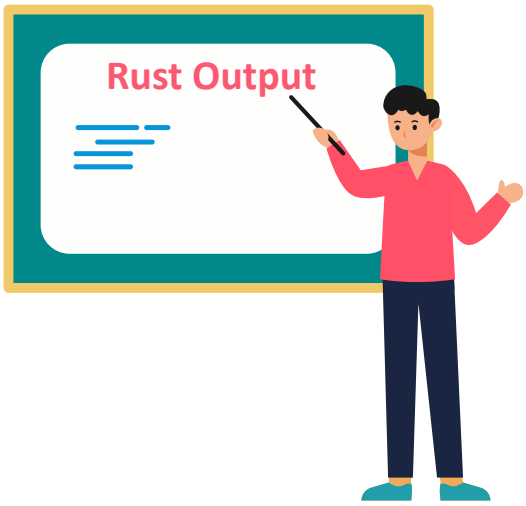
LINUX/MAC

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

NOTE

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

RUST OUTPUT



HELLO WORLD	CODE

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.	`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	CODE

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

The `print!` macro prints text inside double quotes.	`fn main() {`
	`  print!("Hello, World!");`
	`}`

RUST PRINTLN! MACRO	CODE

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

```
}
```

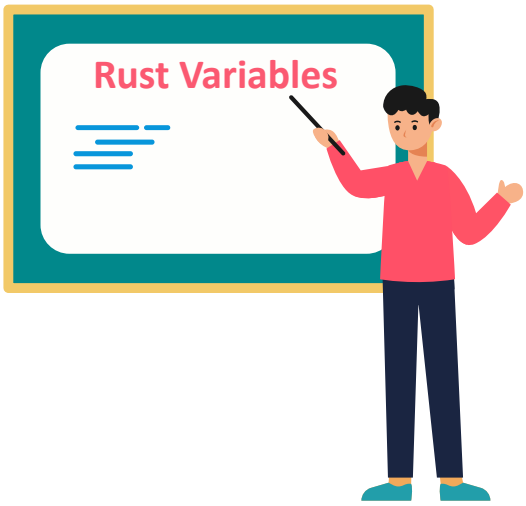
Note

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	CODE

Use the let keyword to declare a variable in Rust	`fn main() {`
	`  let x = 5;`
	`}`

PRINT VARIABLES	CODE

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

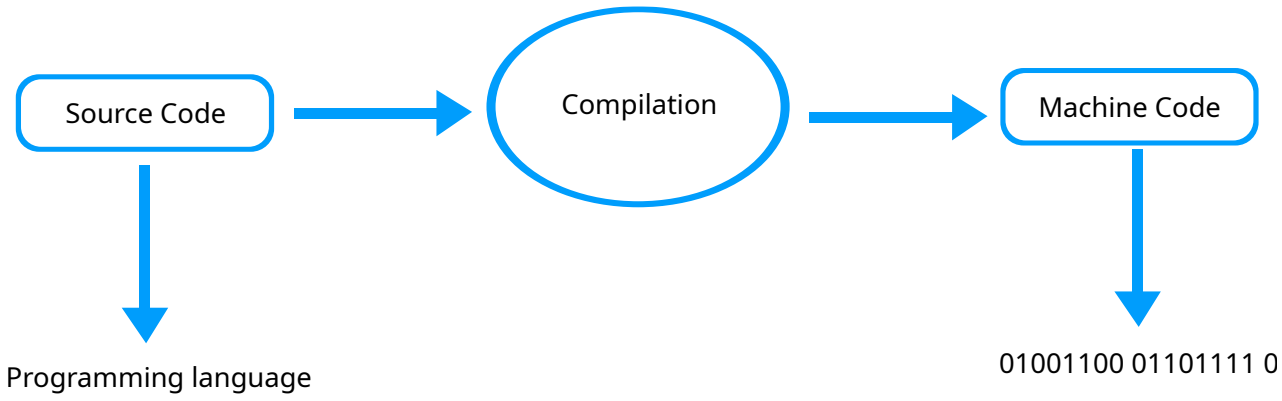
it is required to specify the data type when defining constants

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.

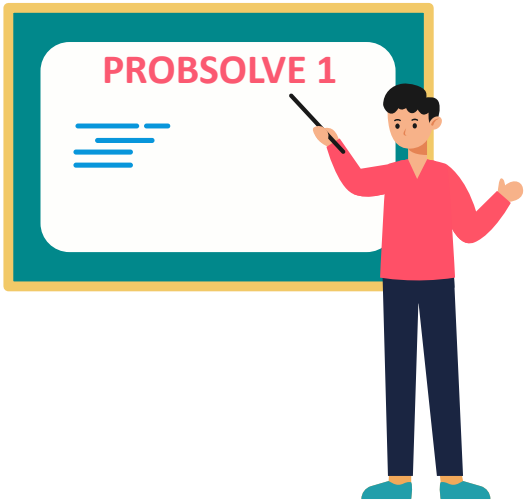


CONSTANTS	V/S	IMMUTABLE VARIABLE
1) A fixed value that never changes		1) A value that can't be changed after being set
2) `let` (e.g., `let x = 5;`)		2) `let` (e.g., `let x = 5;`)
3) Must be explicitly declared		3) Can be inferred or explicitly declared
4) Globally accessible based on visibility		4) 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



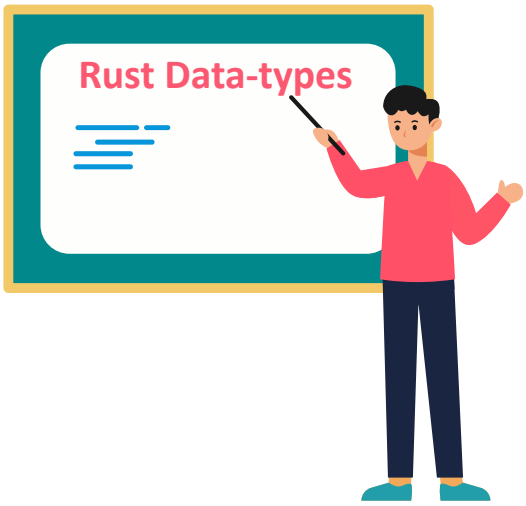
PROBE SOLVE 1 PRACTISE PROBLEMS

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 - Minimum Value : 0 - Maximum Value : 2^n - 1	For an `u8` (8 bit unsigned integer) - Minimum Value: 0 - 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

```
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	CODE

`f32` is a 32-bit floating-point type in Rust with about 7 decimal digits of precision.	`fn main() {`
	`  let x: f32 = 3.14;`
	`  println!("Value: {}", x);`
	`}`

FLOATING-POINT: f64	CODE

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

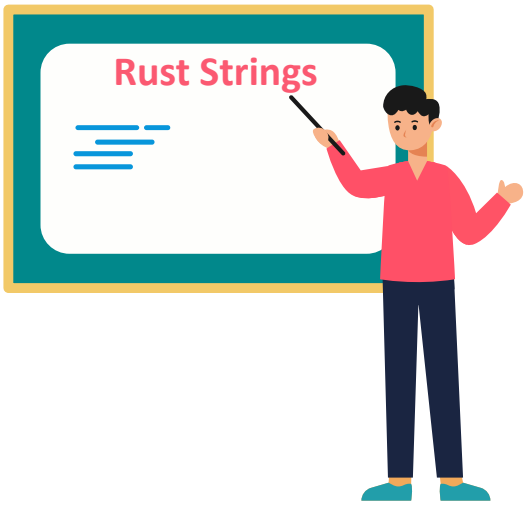
RUST OPERATOR PRECEDENCE AND ASSOCIATIVITY TABLE

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()	CODE

Creates a new, empty `String` object in Rust that can be modified later.	`fn main() {`
	`  let mut my_string = String::new();`
	`  my_string.push_str("Hello, Rust!");`
	`  println!("{}", my_string);`
	`}`

String::from()	CODE

This creates a string object with some default value.	`fn main() {`
	`  let my_string = String::from("Hello, Rust!");`
	`  println!("{}", my_string);`
	`}`

STRING LITERAL	VS	STRING OBJECT
String literals are immutable, fixed sequences of characters stored on the stack memory.		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	CODE

• 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.
	`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!()	CODE

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

• push() → Appends a single character (char).
• push_str() → Appends a string slice (&str).
	`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()	CODE

Used for dynamically joining multiple strings.	`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	CODE

Convert integer into string using .to_string()	`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:

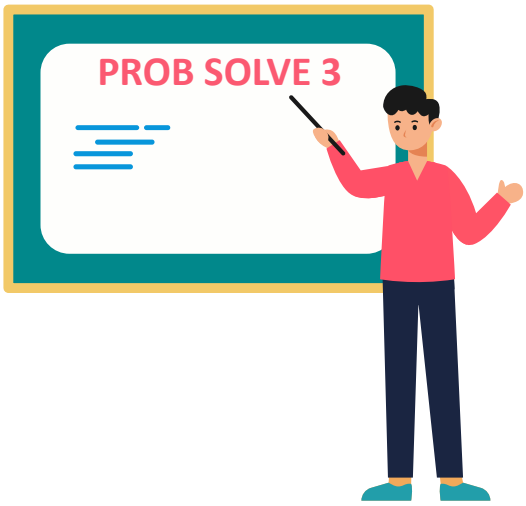
Method	Description	Example
.len()	Returns the length	`"Hello".len() → 5`
.is_empty()	Checks if the string is empty	`"".is_empty() → true`
.push(char)	Appends a single character	`s.push('R')`
.push_str(&str)	Appends a string slice	`s.push_str("ust")`
.insert(index, char)	Inserts a character at a position	`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

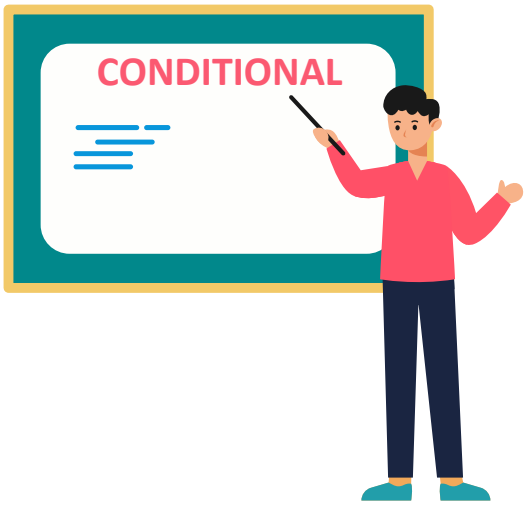
PROB-SOLVE 3



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	CODE
Use else if to check multiple conditions in sequence.	

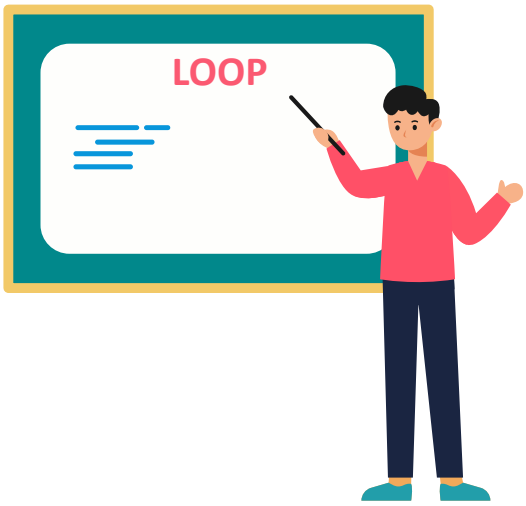
```
fn main() {
    let number = 0;

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

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

```
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.

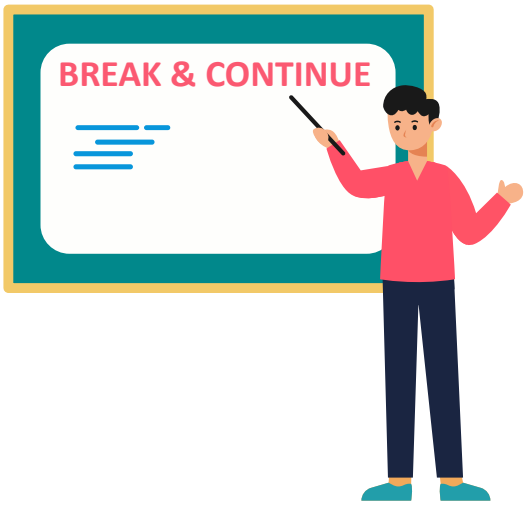
CODE

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

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

FOR LOOP	CODE

• Used for iterating over a range or collection.
• More concise and safer than while.
	`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);
    }
}
```

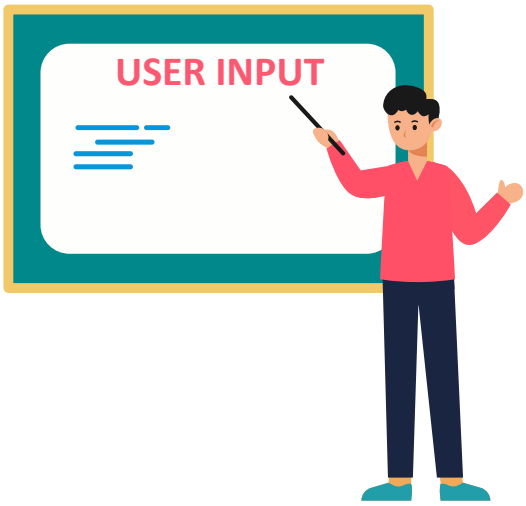
CONTINUE (SKIPS ITERATION)

- 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

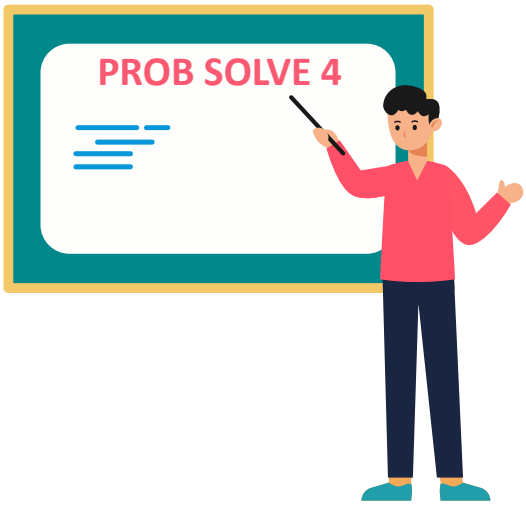
Modules in Rust group related functionalities together. They are part of the language, but you must import them to use their features.

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.

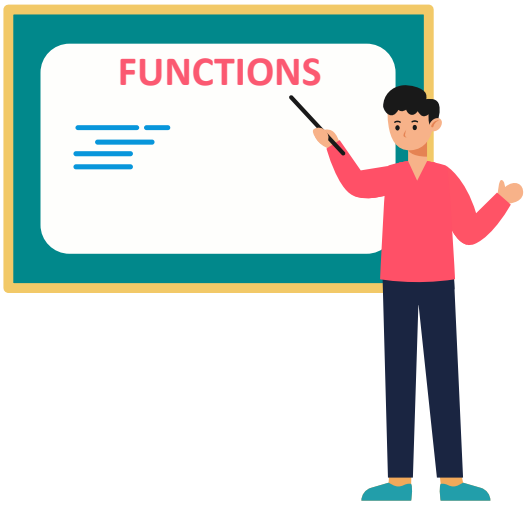
PROB-SOLVE 4



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

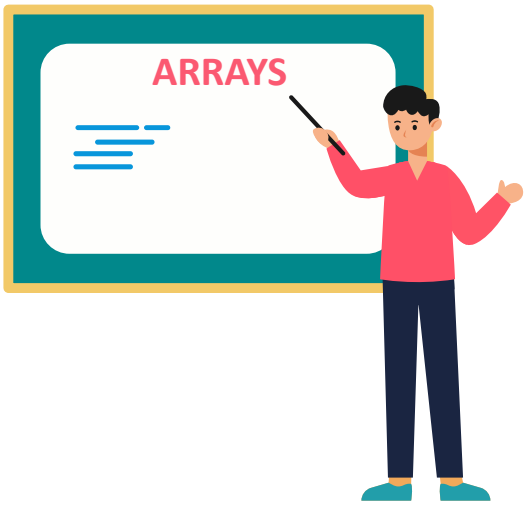
```
fn test_return() {
    println!("This will be printed
first");
    return;
    println!("This will NOT be printed");
}
```

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

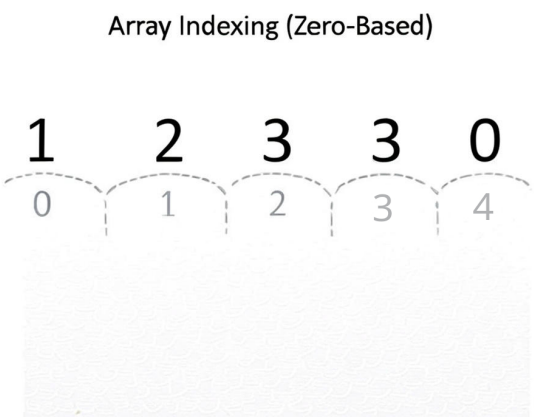
```
fn main() {
    let arr: [i32; 5] = [3; 5];
    println!("array = {:?}", arr);
}
```

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.



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

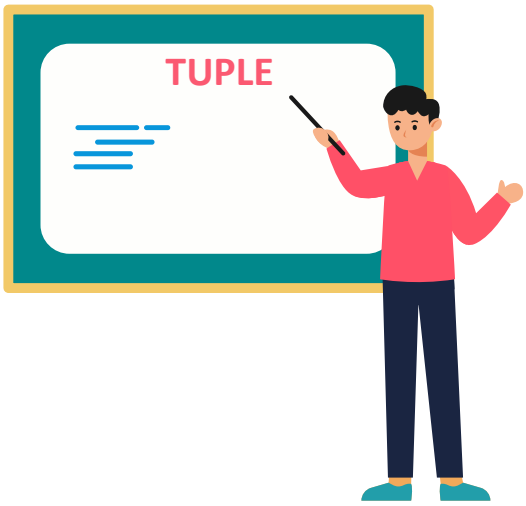
In Rust, a function can return an array by specifying the return type as an array with a fixed size.

- -> [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

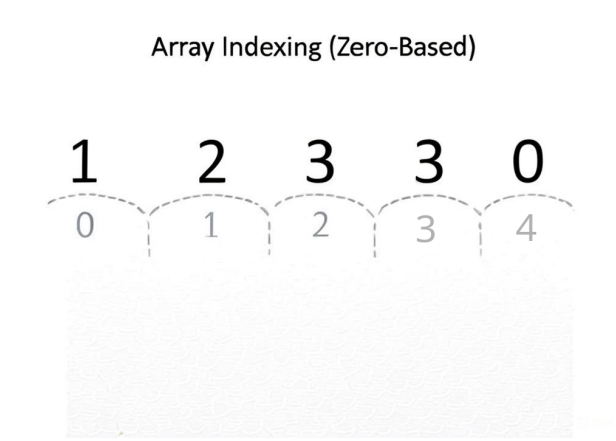
```
fn main() {
    let s: (&str, i32, f32) = ("Alice",
20, 85.5);
    println!("{}, {},{}", s.0, s.1, s.2);}
```

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.