MASTERING THE RUST LANGUAGE

A LoopCursive Book

ABHISHEK

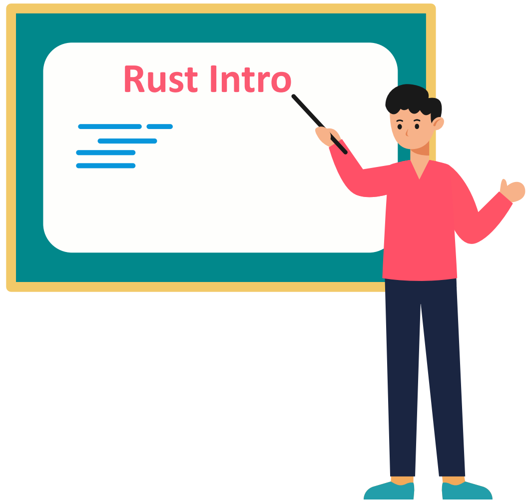
Building System Reliable

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
```

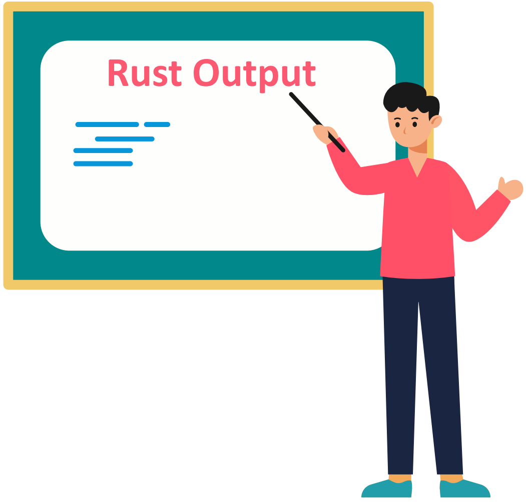
LINUX/MAC

```
$ 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);
```

```
}
```

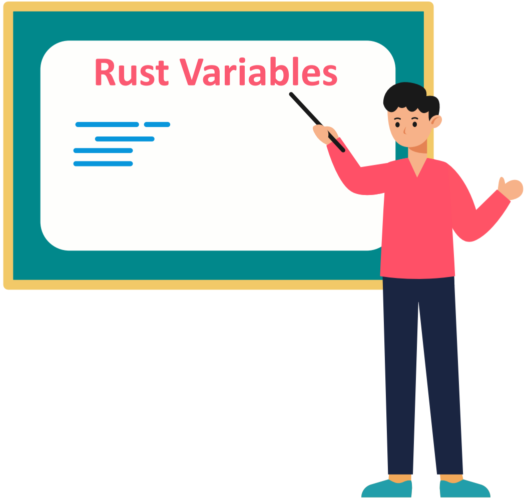
Note

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	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	CODE
We use the \`mut\` keyword to make a variable mutable.	see below

```
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

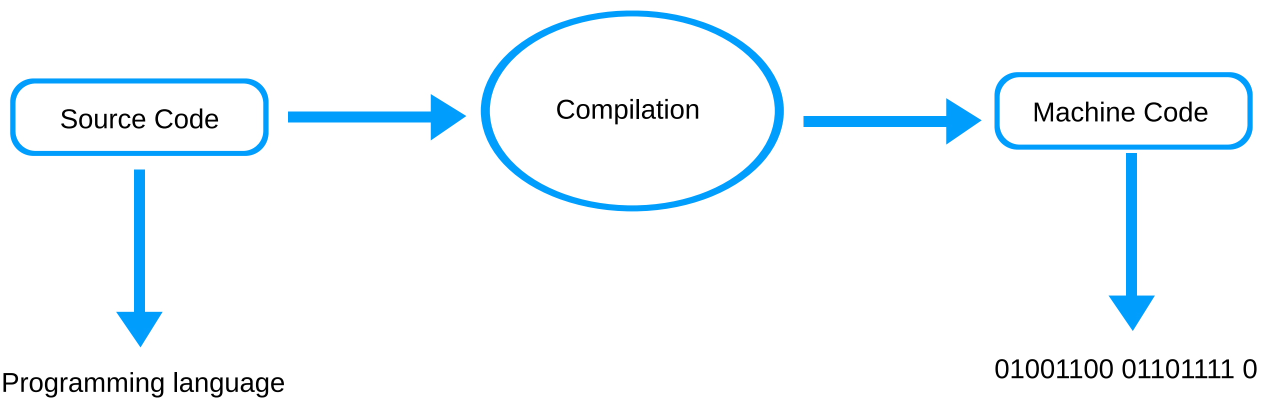
CODE

```
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.

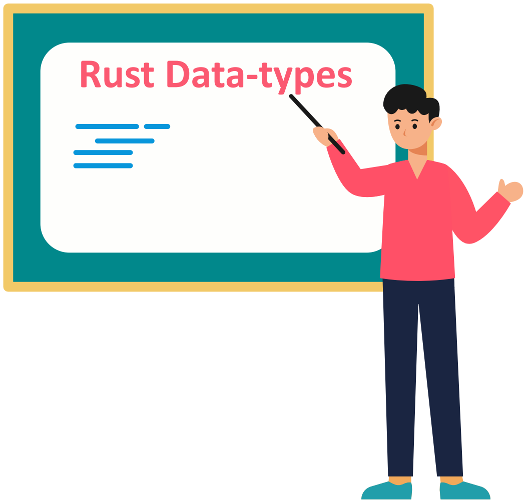


CONSTANTS	V/S	IMMUTABLE VARIABLE
1) A fixed value that never changes		1) A value that can't be changed after being set
2) `let` (e.g., `let x = 5;`)		2) `let` (e.g., `let x = 5;`)
3) Must be explicitly declared		3) Can be inferred or explicitly declared
4) Globally accessible based on visibility		4)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

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

`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.

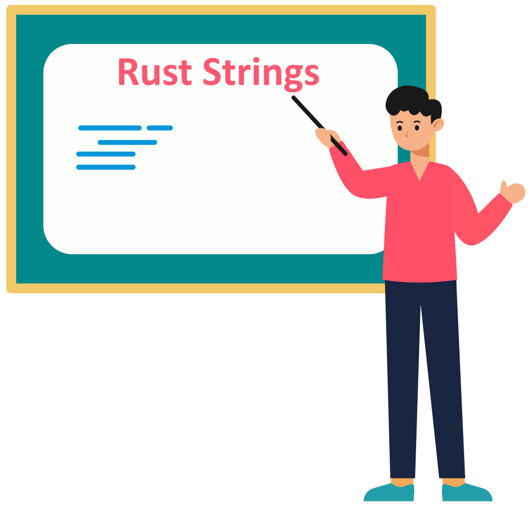
RUST OPERATOR PRECEDENCE AND ASSOCIATIVITY TABLE

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 is the purpose of the char type in Rust?
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?

RUST STRINGS



RUST STRINGS
In Rust strings are sequences of characters.

RUST STRING TYPES
1) String literal(&str) 2) String object(String)

<table>
<tr><th>STRING LITERAL(&str)</th><th>CODE</th></tr>
<tr><td>String literals (`&str`) are fixed, hardcoded strings known at compile time</td><td>

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

</td></tr>
</table>

<table>
<tr><th>STRING OBJECT</th><th></th><th>CODE</th></tr>
<tr><td>In Rust,String object is a mutable, growable sequence of characters stored in the heap, defined in the Standard Library, and capable of representing text values that are determined at runtime.</td><td></td><td>

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

</td></tr>
</table>

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

String::from()		CODE

Creates a new \`String\` object in Rust from a string literal or slice that can be modified later.		<pre>fn main() {
		    let my_string = String::from("Hello,
		Rust!");
		    println!("{}", my_string);
		}</pre>

STRING LITERAL	VS	STRING OBJECT
String literals are immutable, fixed sequences of characters stored on the stack		String objects are mutable, growable sequences of characters stored on the heap