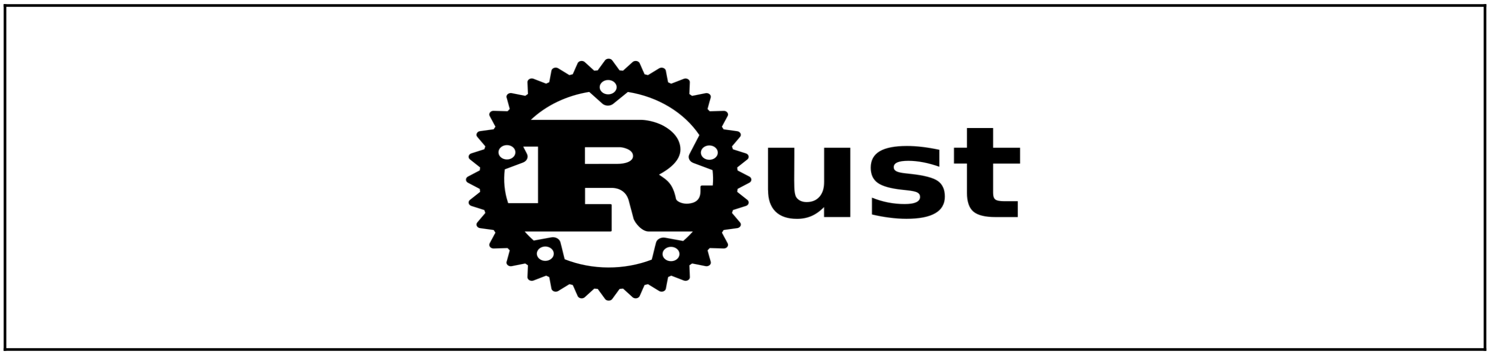
Full Lab to Master Rust Programming Language

Presentation · March 2025

Youcef Benabderrezak
University of Boumerdes

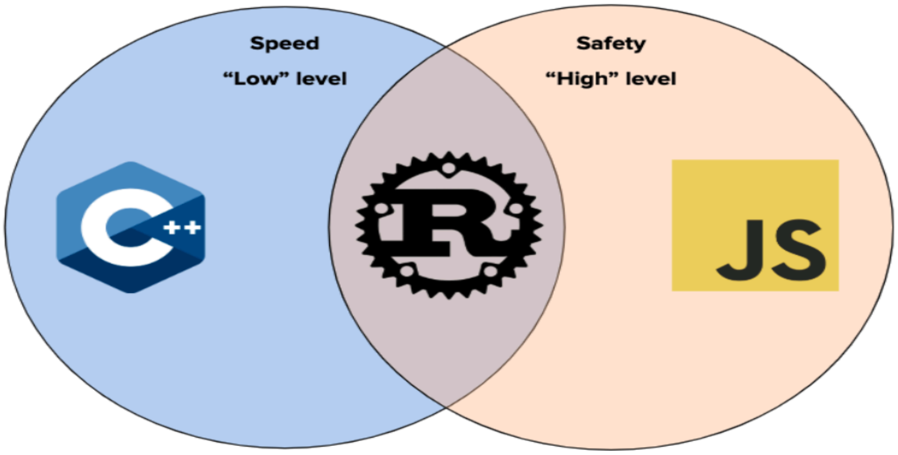
Introduction

- Rust is a modern systems programming language that focuses on performance, safety, and concurrency.
- It was designed to help developers build reliable and efficient software while preventing common bugs, such as memory errors, through its unique ownership model.



Here are some key points about Rust :

1. **Memory Safety Without Garbage Collection** :Rust's ownership, borrowing, and lifetime system ensures memory is managed safely at compile time without the need for a garbage collector.
2. **Concurrency** : Rust's design makes it easier to write concurrent code that is free from data races, which is crucial for modern, multi-threaded applications.
3. **Performance** : Rust offers performance comparable to C or C++, making it ideal for system-level programming, embedded systems, and high-performance applications.
4. **Modern Tooling and Ecosystem** : The language is backed by Cargo (its package manager and build system) and has a rich ecosystem of libraries (crates) available on crates.io.
5. **Expressive Syntax** : Rust supports modern language features such as pattern matching, algebraic data types (enums), and powerful macros for metaprogramming



1. Getting Started : Installing Rust on Ubuntu

1.1. Installing Rust via rustup

- Rust's recommended installation tool is rustup.
- Open your terminal and run : **curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh**
- Follow the on-screen instructions to complete the installation.
- This sets up the latest stable Rust compiler and Cargo—the Rust package manager.
- Cargo's bin directory to your PATH : **export PATH="$HOME/.cargo/bin:$PATH"**
- verify your installation with :

```
rustc --version
cargo --version
```

1.2. Creating Your First Rust Project

- Use Cargo to create a new project :

```
cargo new hello_rust
cd hello_rust
```

- Inside the src/main.rs file, you'll find a simple "Hello, world!" program. Run it with :

```
cargo run
```

2. Rust Basics

2.1. Hello World & Basic Syntax

Code	Explanation

fn main() {
    println!("Hello, world!");
}
	● fn main() defines the entry point.
	● println! is a macro that prints text to the console.

2.2. Variables and Mutability

- Rust variables are **immutable (غير قابل للتغيير)** by default.
- Use mut to allow changes.

Code	**Explanation**

fn main() {
    let x = 5;
    println!("The value of x is: {}", x);
 	
    let mut y = 10;
    println!("Initial value of y: {}", y);
    y = 15;
    println!("Updated value of y: {}", y);
}
	● let x = 5; creates an immutable binding.
	● let mut y = 10; creates a mutable binding.

2.3. Data Types

Rust has scalar types (integers, floats, booleans, characters) and compound types (tuples, arrays).

Code	**Output**

fn main() {
    // Scalar types
    let integer: i32 = 100;
    let float: f64 = 3.14;
    let boolean: bool = true;
    let character: char = 'R';
 	
    // Compound types: Tuple and Array
    let tup: (i32, f64, char) = (500, 6.4, 'x');
    let (a, b, c) = tup; // destructuring tuple
    let arr: [i32; 3] = [1, 2, 3];	

println!("Integer: {}, Float: {}, Boolean: {}, Character: {}", integer, float, boolean, character); println!("Tuple values: {} {} {}", a, b, c); println!("Array element at index 0: {}", arr[0]); }	Integer: 100, Float: 3.14, Boolean: true, Character: R Tuple values: 500 6.4 x Array element at index 0: 1

2.4. Control Flow

Rust supports standard control structures: if/else, loops (loop, while, for).

Example: If/Else and Looping

Code	Output

fn main() {
    let number = 6;
 	
    if number % 2 == 0 {
        println!("{} is even", number);
    } else {
        println!("{} is odd", number);
    }
 	
    // For loop
    for n in 1..=5 {
        println!("Number: {}", n);
    }
}
	6 is even
	
	Number: 1
	Number: 2
	Number: 3
	Number: 4
	Number: 5

3. Functions and Comments

3.1. Functions

Functions are declared using the fn keyword.

Code	**Output**

fn main() {
    let result = add(5, 3);
    println!("Sum is: {}", result);
}
 	
fn add(a: i32, b: i32) -> i32 {
    a + b // **implicit return; no semicolon**
}
	Sum is : 8

3.2. Comments

- Use // for single-line comments and /* ... */ for block comments.

```
fn main() {
    // This is a single-line comment.
    println!("Comments are ignored by the compiler.");

    /*
      This is a block comment.
      You can comment out multiple lines.
    */
}
```

4. Ownership, Borrowing, and Lifetimes

Rust’s ownership model ensures memory safety without a garbage collector.

4.1. Ownership Basics

Example: Ownership and Moves

Code	Output

fn main() {
    let s1 = String::from("hello");
    let s2 = s1; // **s1 is moved to s2**
    println!("{}", s1);
}
	error because s1 is no longer valid

4.2. Borrowing and References

Borrowing allows you to reference data without taking ownership.

Example: Borrowing

Code	**Output**

fn main() {
    let s = String::from("hello");
    print_string(&s); // **Passing a reference**
    println!("s is still valid: {}", s);
}
 	
fn print_string(s: &String) {
    println!("{}", s);
}
	hello
	s is still valid: hello

4.3. Lifetimes (Basic Intro)

Lifetimes prevent dangling references. For example:

Code	**Output**

fn main() {
    let r;
    {
        let s = String::from("hello");
        r = &s;
  // s is dropped at the end of this block, so r would be invalid if used outside
    }
println!("{}", r);
}
	s does not live long enough

Tip : Lifetime annotations are often required in function signatures when multiple references are involved

5. Data Structures: Structs, Enums, and Pattern Matching

5.1. Structs : Structs are custom data types that group related data.

Code	**Output**

struct User {
    username: String,
    email: String,
    sign_in_count: u64,
    active: bool,
}
fn main() {
    let user1 = User {
        username: String::from("ahmed"),
        email: String::from("alice@example.com"),	

sign_in_count: 1,
        active: true,
    };
    println!("Username : {}", user1.username);
}
	Username : ahmed

5.2. Enums and Pattern Matching

Enums allow you to define a type by enumerating its possible variants.

Code	Output

enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(i32, i32, i32),
}
fn main() {
    let msg = Message::Write(String::from("Hello"));
 	
    match msg {
Message::Quit => println!("Quit message"),
Message::Move { x, y } => println!("Move to ({}, {})", x, y),
Message::Write(text) => println!("Text message: {}", text),
Message::ChangeColor(r, g, b) => println!("Change color to ({}, {}, {})", r, g, b),
    }
}
	Text message: Hello

6. Collections and Iterators

Rust offers powerful built-in collections like vectors, strings, and hash maps.

6.1. Vectors

- Vectors in Rust are a growable, heap-allocated collection type defined by the **Vec<T>** struct.
- They allow you to store a sequence of values that all have the same type and can be dynamically resized

Code	Output

fn main() {
    let mut v = Vec::new();
    v.push(5);
    v.push(6);
    v.push(7);
 	
    for i in &v {
        println!("{}", i);
    }
}
	5
	6
	7

6.2. Strings : Rust’s String type is a growable, mutable UTF-8 encoded string.

Code	Output

fn main() {
    let mut s = String::from("Hello");
    s.push_str(", world!");
    println!("{}", s);
}
	Hello, world!

6.3. HashMap

- Hash maps in Rust, provided by the **std::collections::HashMap** type, are collections that store key-value pairs.
- They are useful when you need to associate data together and retrieve values quickly based on their keys

Code	Output

use std::collections::HashMap;
fn main() {
    let mut scores = HashMap::new();
    scores.insert(String::from("Blue"), 10);
    scores.insert(String::from("Red"), 50);
 	
    for (team, score) in &scores {
        println!("Team {} : {}", team, score);
    }
}
	Team Red : 50
	Team Blue : 10

6.4. Iterators and Closures : Iterators provide a powerful way to process sequences.

Example: Iterator with a Closure

Code	Output

fn main() {
    let numbers = vec![1, 2, 3, 4, 5];
    let doubled: Vec<i32> = numbers.iter().map(|x| x * 2).collect();
    println!("Doubled numbers: {:?}", doubled);
}
	Doubled numbers: [2, 4, 6, 8, 10]

7. Modules, Crates, and Package Management

7.1. Modules : Modules help you organize your code into logical units.

Example: Creating a Module

Create a file structure like this :

```
src/
├── main.rs
└── greeting.rs
```

File	Content

greeting.rs	pub fn say_hello() {
	    println!("Hello from the greeting module!");
	}

main.rs	mod greeting;
	
	fn main() {
	    greeting::say_hello();
	}

7.2. Using External Crates

- Use Cargo to add dependencies. For example, to use the **rand** crate, add it to your **Cargo.toml** :

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

- Then use it in your code :

```rust
use rand::Rng;

fn main() {
    let mut rng = rand::thread_rng();
    let n: u8 = rng.gen_range(0..100);
    println!("Random number: {}", n);
}
```

8. Error Handling

Rust **distinguishes (يميز)** between **recoverable (القابلة للاسترداد)** and unrecoverable errors.

8.1. The Option and Result Types

Example : Using Option

Code	**Output**

fn main() {
    let some_number = Some(5);
    let absent_number: Option<i32> = None;
    match some_number {
        Some(n) => println!("Number is : {}", n),
        None => println!("No number found"),
    }
}
	Number is : 5

Example : Using Result

Code

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


fn main() {
    let file = File::open("hello.txt");
    let _file = match file {
        Ok(f) => f,
        Err(e) => {
            println!("Error opening file: {:?}", e);
            return;
        }
    };
}
```

Tip *:* Use the **?** operator to propagate errors more succinctly

9. Generics, Traits, and Lifetimes

9.1. Generics

- Generics in Rust allow you to write flexible, reusable code that can work with many different data types without sacrificing type safety.
- Instead of writing the same code for each data type, you define the behavior once with a generic placeholder (or type parameter) that can be substituted with any concrete type when the code is used

Example: A Generic Function

Code	Output

fn largest<T: PartialOrd>(list: &[T]) -> &T {
    let mut largest = &list[0];
 	
    for item in list.iter() {
        if item > largest {
            largest = item;
        }
    }
    largest
}
 	
fn main() {
    let numbers = vec![34, 50, 25, 100, 65];
println!("The largest number is {}", largest(&numbers));
}
	The largest number is 100

9.2. Traits : Traits define shared behavior. They're similar to **interfaces** in other languages.

Code	Output

trait Summary {
    fn summarize(&self) -> String;
}
 	
struct Article {
    headline: String,
    content: String,
}
 	
impl Summary for Article {	

fn summarize(&self) -> String {
format!("{}: {}", self.headline, &self.content[0..20])
}
}
 	
fn main() {
let article = Article {
headline: String::from("Rust is awesome"),
content: String::from("Rust provides memory safety without a garbage collector..."),
};
println!("Summary: {}", article.summarize());
}
	Summary: Rust is awesome: Rust provides memory

9.3. Lifetimes (Revisited)

When multiple references are involved, lifetime annotations ensure references are valid.

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() { x } else { y }
}

fn main() {
    let str1 = String::from("long string is long");
    let str2 = "short";
    let result = longest(str1.as_str(), str2);
    println!("Longest string: {}", result);
}
```

10. Advanced Topics

- Closures
- Concurrency
- Asynchronous Programming
- Macros
- Unsafe Rust

11. Additional Tips & Resources

- **Documentation:**
  - The Rust Programming Language Book (often called "The Book") is the best starting point.
  - Rust by Example offers hands-on examples.
- **Community:**
  - Join Rust forums, Discord channels, and local meetups for further learning.
- **Practice:**
  - Build small projects or contribute to open-source Rust projects to apply what you learn.