Dharmsinh Desai University
MCA Department

Semester -II

Seminar

Report on
The Rust Programming Language

By

HARSORA RUSHIT MA019

Index

1. Introduction

1.1 Introduction Of Rust Programming Language

Welcome To The Rust Programming Language, An Introductory Book About Rust. The Rust Programming Language Helps You Write Faster, More Reliable Software. High-Level Ergonomics And Low-Level Control Are Often At Odds In Programming Language Design; Rust Challenges That Conflict.

Through Balancing Powerful Technical Capacity And A Great Developer Experience, Rust Gives You The Option To Control Low-Level Details (Such As Memory Usage) Without All The Hassle Traditionally Associated With Such Control.

The Rust Programming Language Helps You Write Faster, More Reliable Software. High-Level Ergonomics And Low-Level Control Are Often At Odds In Programming Language Design; Rust Challenges That Conflict.

Through Balancing Powerful Technical Capacity And A Great Developer Experience. Rust Gives You The Option To Control Low-Level Details (Such As Memory Usage) Without All The Hassle Traditionally Associated With Such Control.

1.1.1 Open Source and Open Governance

Open Source refers to software that has its source code made available to the public, allowing anyone to view, modify, and distribute it. Open Source software is often free to use and can be contributed to and improved by a community of developers and users.

Open Governance refers to the process by which an Open Source project is managed and governed. Open Governance involves transparency, inclusivity, and community participation in decision-making. It allows for the community to have a say in the direction and development of the project.

Open Source and Open Governance often go hand in hand. When a project is Open Source, anyone can contribute to it, but Open Governance ensures that the community has a say in how the project is run and what changes are made to it. This helps to create a more collaborative and inclusive environment, and can lead to better decision-making and faster innovation.

Some examples of Open Source and Open Governance projects include Linux, the Apache web server, and the Python programming language. These projects are managed by a community of developers and users who work together to improve and maintain them.

1.1.2 Top-tier performance

Top-tier performance refers to achieving the highest levels of performance in a given domain or task. This often involves optimizing performance metrics such as speed, efficiency, accuracy, and scalability.

Achieving top-tier performance can be a critical factor in many fields, such as finance, gaming, data analytics, scientific research, and machine learning. For example, a financial trading system needs to be able to process millions of transactions per second with low latency, while a machine learning system needs to train models on large datasets quickly and accurately.

To achieve top-tier performance, developers and engineers need to employ a variety of techniques such as algorithmic optimization, hardware acceleration, parallel processing, caching, and load balancing. They also need to have a deep understanding of the domain and the specific use case, as well as access to the latest tools and technologies.

Some examples of top-tier performance achievements include the fastest supercomputers in the world, the highest-performing gaming systems, and the most efficient data centers. These achievements are often the result of years of research and development, as well as collaboration among experts in multiple fields.

1.1.3 Like C/C++ or better

Rust is a programming language that is often compared to C and C++ because of its emphasis on performance, memory safety, and low-level system programming. However, Rust has some advantages over C and C++ that make it a better choice for certain use cases.

First, Rust has a modern and expressive syntax that is more readable and maintainable than C and C++. It also has advanced features such as pattern matching, closures, and iterators that make it easier to write high-level code.

Second, Rust has a unique ownership and borrowing system that enforces memory safety at compile-time, eliminating common bugs such as null pointers, buffer overflows, and data races. This makes Rust more secure and reliable than C and C++.

Third, Rust has a powerful package manager called Cargo, which simplifies dependency management and makes it easy to share and reuse code. This is in contrast to C and C++, which often require manual compilation and linking of libraries.

Finally, Rust has a growing community of developers and a vibrant ecosystem of libraries and frameworks that make it easier to get started and build complex systems.

Overall, Rust can be a better choice than C and C++ for projects that require high performance, memory safety, and modern programming features.

1.2 Motivation

Motivation is a driving force that inspires and directs behavior towards a goal or objective. Motivation can come from both internal and external factors, and can be influenced by a variety of factors such as personal values, beliefs, emotions, and social environment.

Motivation plays a critical role in many aspects of life, such as education, work, sports, and personal development. Motivation can help individuals overcome obstacles, achieve goals, and reach their full potential.

There are many different theories of motivation, but some of the key factors that can influence motivation include:

Intrinsic motivation: the desire to pursue a goal for its own sake, rather than for external rewards or pressure. This can be driven by factors such as personal interest, enjoyment, and curiosity.

Extrinsic motivation: the desire to pursue a goal for external rewards or pressure, such as money, recognition, or social approval.

Self-efficacy: the belief in one's ability to succeed at a task or goal.

Goal setting: the process of setting specific, challenging, and achievable goals that provide a sense of direction and purpose.

Social support: the encouragement, feedback, and assistance provided by others, which can help individuals stay motivated and overcome challenges.

Overall, motivation is a complex and multifaceted concept that can have a significant impact on individual and collective behavior and achievement. Understanding and harnessing motivation can be a key factor in personal and professional success.

2. About Rust

2.1 About Rust

Rust is a system programming language developed by a Mozilla employee "Graydon Hoare" in 2006. He described this language as a "safe, concurrent and practical language" that supports the functional and imperative paradigm.

The syntax of rust is similar to the C++ language.

Rust is free and open source software, i.e., anyone can use the software freely, and the source code is openly shared so that the people can also improve the design of the software.

Rust is declared as one of the "most loved programming language" in the stack overflow developer survey in 2016, 2017 and 2018.

There is no direct memory management like calloc or malloc. It means, the memory is managed internally by Rust.

2. Features Of Rust

2.2 .1 Zero cost abstraction

In Rust, we can add abstractions without affecting the runtime performance of the code. It improves the code quality and readability of the code without any runtime performance cost.

2.2.2 Error messages

In C++ programming, there is an excellent improvement in error messages as compared to GCC. Rust goes one step further in case of clarity. Error messages are displayed with (formatting, colors) and also suggest misspellings in our program.

2.2.3 Move semantics

Rust provides this feature that allows a copy operation to be replaced by the move operation when a source object is a temporary object

2.2.4 Threads without data races

A data race is a condition when two or more threads are accessing the same memory location. Rust provides the feature of threads without data races because of the ownership system. Ownership system transmits only the owners of different objects to different threads, and two threads can never own the same variable with write access.

2.2.5 Pattern matching

Rust provides the feature of pattern matching. In pattern matching, patterns in Rust are used in conjunction with the 'match' expressions to give more control over the program's control flow. Following are the combinations of some patterns:

Literals
Arrays, enums, structs, or tuples
Variables
Wildcards
Placeholders

2.2.6 Guaranteed memory safety

Rust guaranteed the memory safety by using the concept of ownership. Ownership is a middle ground between the memory control of C and the garbage collection of java. In Rust programs, memory space is owned by the variables and temporarily borrowed by the other variables. This allows Rust to provide the memory safety at the compile time without relying on the garbage collector.

2.2.7 Efficient C bindings

Rust provides the feature of 'Efficient C bindings' means that the Rust language can be able to interoperate with the C language as it talks to itself. Rust provides a 'foreign function interface' to communicate with C API's and leverage its ownership system to guarantee the memory safety at the same time.

2.2.8 Safe memory space allocation

In Rust, memory management is manual, i.e., the programmer has explicit control over where and when memory is allocated and deallocated. In C language, we allocate the memory using malloc function and then initialize it but Rust refuses these two operations by a single '~' operator. This operator returns the smart pointer to int. A smart pointer is a special kind of value that controls when the object is freed. Smart pointers are "smart" because they not only track where the object is but also know how to clean it up.

3. Environment Setup

3.1 Installation on Windows

The first step is to install Rust. We'll download Rust through rustup, a command line tool for managing Rust versions and associated tools. You'll need an internet connection for the download.

Note: If you prefer not to use rustup for some reason, please see the Other Rust Installation Methods page for more options.

The following steps install the latest stable version of the Rust compiler. Rust's stability guarantees ensure that all the examples in the book that compile will continue to compile with newer Rust versions. The output might differ slightly between versions because Rust often improves error messages and warnings. In other words, any newer, stable version of Rust you install using these steps should work as expected with the content of this book.

3.2 Command Line Notation

In this chapter and throughout the book, we'll show some commands used in the terminal. Lines that you should enter in a terminal all start with $. You don't need to type the $ character; it's the command line prompt shown to indicate the start of each command. Lines that don't start with $ typically show the output of the previous command. Additionally, PowerShell-specific examples will use > rather than $.

3.3 Installing rustup on Linux or macOS

If you're using Linux or macOS, open a terminal and enter the following command:

```
$ curl --proto '=https' --tlsv1.2 https://sh.rustup.rs -sSf | sh
```

The command downloads a script and starts the installation of the rustup tool, which installs the latest stable version of Rust. You might be prompted for your password. If the install is successful, the following line will appear:

Rust is installed now. Great!
You will also need a linker, which is a program that Rust uses to join its compiled outputs into one file. It is likely you already have one. If you get linker errors, you should install a C compiler, which will typically include a linker. A C compiler is also useful because some common Rust packages depend on C code and will need a C compiler.

On macOS, you can get a C compiler by running:
$ xcode-select --install
Linux users should generally install GCC or Clang, according to their distribution's documentation. For example, if you use Ubuntu, you can install the build-essential package.

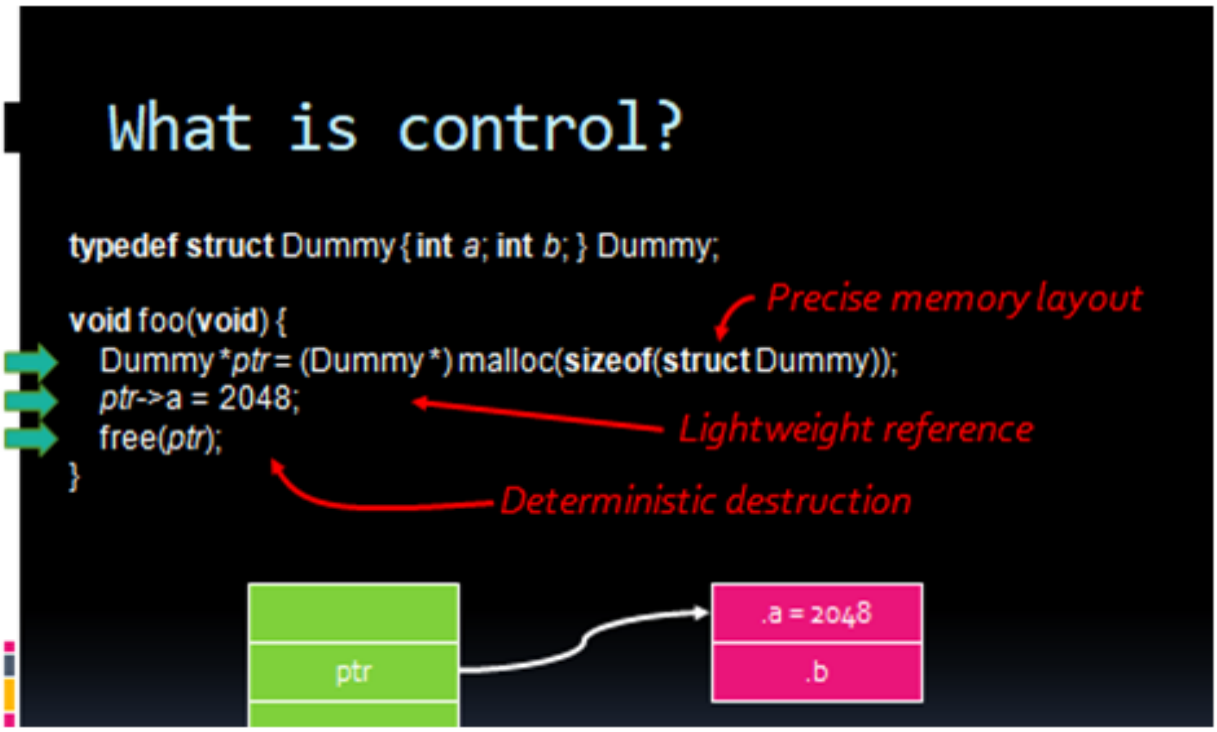
3.4 Basic Syntax of rust

```
fn
main(){
  println!("HELLO WORLD !");
}
```

4.Control And Safety

4.1 what is control

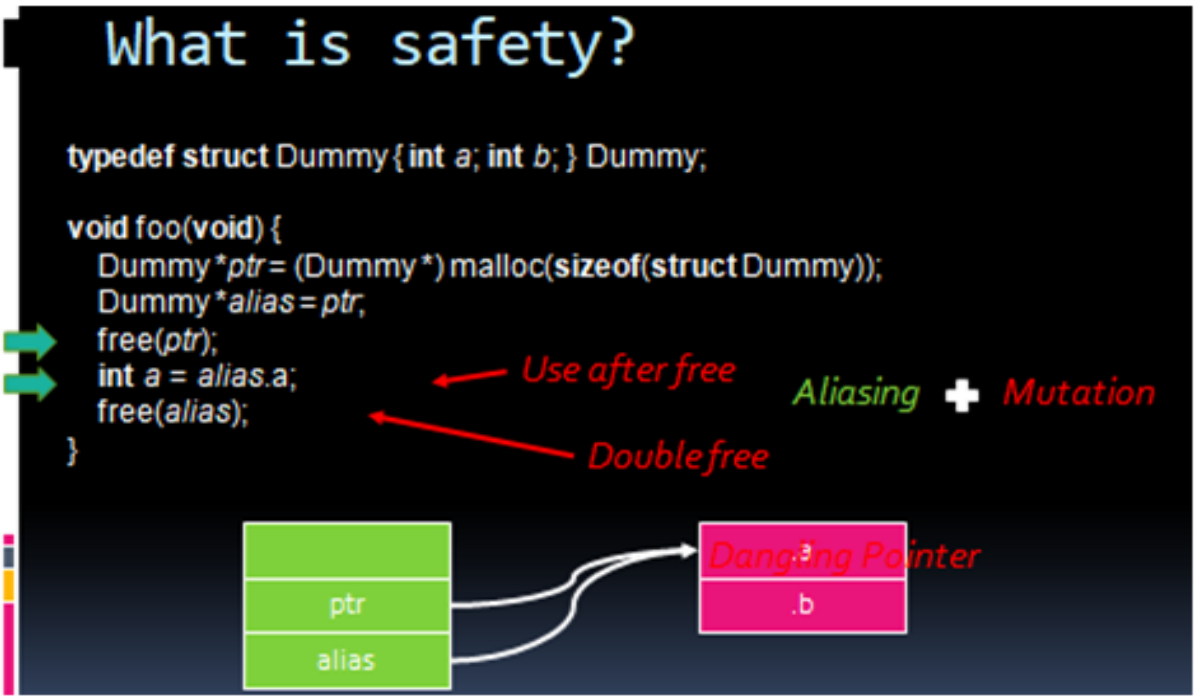
The ability to run some code depending on whether a condition is true and to run some code repeatedly while a condition is true are basic building blocks in most programming languages. The most common constructs that let you control the flow of execution of Rust code are if expressions and loops.



(4.1 what is control)

4.2 what is safety

In Rust, safety is a key feature of the language. Rust's ownership and borrowing system ensures that memory is managed safely at compile-time, eliminating common errors such as null pointers, buffer overflows, and data races. This system also prevents undefined behavior by enforcing strict rules for mutable and immutable references to data, and it helps prevent memory leaks by ensuring that all resources are freed when they are no longer needed

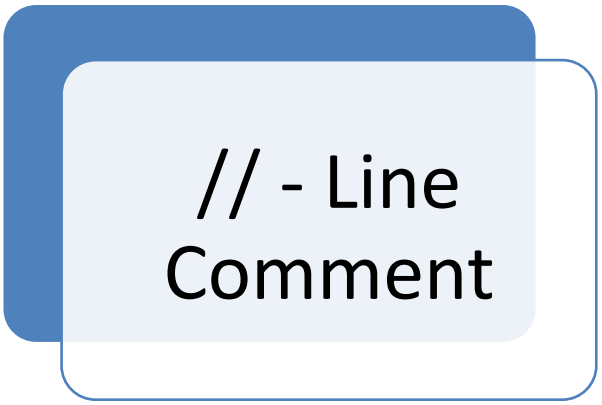


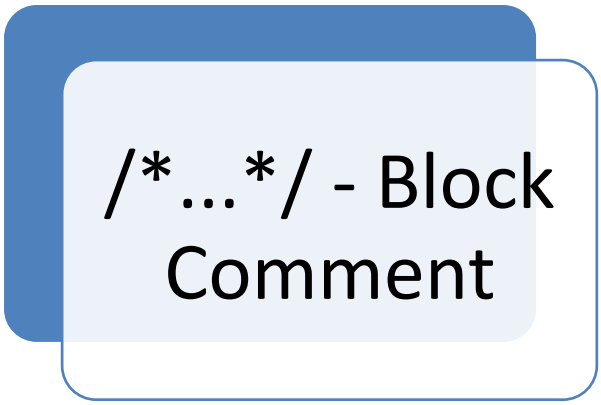
(4.2 what is safety)

4.3 In The Real World

Rust's use in web development, network programming, game development, systems programming, and embedded systems.

4.4 Comments





4.5 Basic Rust Program

```rust
use std::io;

fn main() {
    println!("Enter the first number:");
    let mut num1 = String::new();
    io::stdin().read_line(&mut num1).unwrap();
    let num1: i32 = num1.trim().parse().unwrap();

    println!("Enter the second number:");
    let mut num2 = String::new();
    io::stdin().read_line(&mut num2).unwrap();
    let num2: i32 = num2.trim().parse().unwrap();

    let sum = num1 + num2;
    println!("The sum of : {}",sum);
}
```

4.6 Why Use Rust ?

1.High performance while ensuring memory safety

2.Support for concurrent programming

3.The growing number of Rust packages at crates.io repository

4.A vibrant community driving the development of the language

5.Backwards compatibility and stability ensured

5. Data Type in rust

5.1 Integer

5.1.1 sign integer

```
fn main() {
        // Signed integer type
        let x: i32 = -200;
        let y: i32 = 200;
            println!("x = {}", x);
            println!("y = {}", y);
        }
```

Output:

```
x = -200
y = 200
```

It Declares A Function Called Main With No Parameters And No Return Value.

The let keyword is used to introduce a new variable binding.

i - specifies signed integer type (can store both positive or negative value)

32 - size of the data type (takes 32 bits of space in memory)

The {} is a placeholder for the value of the variables passed as arguments after the string. In this case, the variables x and y are printed using {} with their respective values.

5.1.2 unsign integer

```
fn main() {
        // Unsigned integer type
            let x: u32 = 300;
            println!("x = {}", x);
    }
```

Output:

```
x = 300
```

We Can Also Create Variables That Can Only Store Positive Integer Values
U32 specifies That The x variable Can Only Store Positive Values.
U specifies Unsigned Integer Type.
If We Try To Store Negative Numbers To u32 type Variables, We Will Get An Error

5.2 Floating-Point

```
fn main() {
 // f32 floating point type
        let x: f32 = 3.1;
// f64 floating point type
        let y: f64 = 45.0000031;
        println!("x = {}", x);
        println!("y = {}", y);
}
```

```
Output:
x = 3.1
y = 45.0000031
```

Rust Also Has two Primitive Types for Floating-point Numbers.
Rust's Floating-point Types Are F32 And F64 , Which Are 32 Bits And 64 Bits In Size, Respectively.
F32 is A Type Declaration For The Floating Point Value. In This Case, x is Assigned To A Floating Point Value Of 3.1.

5.3 Boolean

```
fn main() {
// boolean type
    let flag1: bool = true;
    let flag2: bool = false;
    println!("flag1 = {}", flag1);
    println!("flag2 = {}", flag2);
}
```

```
Output:
flag1 = true
flag2 = false
```

In Rust, A Boolean Data Type Can Have Two Possible Values: true or false
We Have Used The bool keyword To Represent The Boolean Type In Rust.
Note: booleans Are Frequently Used In Conditional Statements Like If/Else Expressions.

5.4 Character

```
fn main() {
// char type
    let character: char = 'z';
    println!("character = {}", character);
}
```

```
Output:
character = z
```

The character data type in Rust is used to store a character.
char represents the character type variable and we use single quotes to represent a character.
We can also store special characters like $, @, &, etc. using the character type.
We can also store numbers as characters using single quotes

6. Type Of Interface

6.1 Trait Interface

Traits are used to define a set of methods that can be implemented by different types. Traits are similar to interfaces in other programming languages

Example :-

```
trait Greeting {
        fn greet(&self);
 }
struct Person {
        name: String,
}
impl Greeting for Person {
        fn greet(&self) {
                println!("Hello, my name is {}!",self.name);
                }
 }
fn main() {
        let person = Person { name: String::from("Harsora")
        };
   person.greet();
}
```

Output:-

Hello, my name is Harsora!

Defined Within A Trait And Can Be Implemented By Any Type That Implements The Trait. They Are Similar To Instance Methods But Are Not Defined On A Specific Struct Or Enum.

The Greet Method Takes A Reference To Self As An Argument And Returns Nothing.

Person Struct With A Single Field Name Of Type String

The Person Struct. It Provides A Method Implementation For The Greet Method That Simply Prints A Greeting Message Using The Name Field Of The Person Struct.

Person Instance With The Name Field Set To " Harsora ", And Then Calls The Greet Method On That Instance, Which Prints The Greeting Message To The Console.

6.2 Struct Interface

Structs are used to define complex data structures that can be used as a data interface between different parts of the program.

Example :-

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

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

fn main() {
    let rect = Rectangle { width: 10, height: 20 };
    println!("Area: {}", rect.area());
}
```

Output:-

Area: 200

we define a struct Rectangle with two fields width and height, and a method area that calculates the area of the rectangle. We implement this method for the Rectangle struct using the impl keyword.

In the main function, we create an instance of the Rectangle struct and call its area method to calculate and print its area.

6.3 Function Interface

Functions in Rust can be used as an interface to perform a specific task or operation.

Example:-

```
fn apply<F>(f: F, x: i32) -> i32
where
    F: Fn(i32) -> i32,
{
    f(x)
}
fn main() {
    let result = apply(|x| x * 2, 5);
    println!("Result: {}", result);
}
```

Output:-
Result: 10

we define a function apply that takes a closure f and an i32 value x, and applies the closure to x to get a new i32 value. The closure f takes an i32 parameter and returns an i32 value. We use the where keyword to specify that F must be a closure that takes an i32 parameter and returns an i32 value.

In the main function, we call the apply function with a closure that multiplies its argument by 2 and an i32 value of 5. This returns 10, which we then print using the println macro.

6.4 Macro Interface

Macros are used to define code that can be reused in different parts of the program. They are similar to functions but operate on the AST (Abstract Syntax Tree) level.

Example :-

```
#[cfg(target_os = "linux")]
macro_rules! define_os_specific_function {
   () => {
      fn os_specific_function() {
         println!("This is a Linux-specific function.");
      }
   };
}

#[cfg(target_os = "macos")]
macro_rules! define_os_specific_function {
   () => {
      fn os_specific_function() {
         println!("This is a macOS-specific function.");
      }
   };
}

#[cfg(target_os = "windows")]
macro_rules! define_os_specific_function {
   () => {
      fn os_specific_function() {
         println!("This is a Windows-specific function.");
      }
   };
}

fn main() {
   define_os_specific_function!();
```

```
    os_specific_function();
}
```
Output:-

This is a Linux-specific function.

we define a macro define_os_specific_function that defines a platform-specific function depending on the operating system. We use the cfg attribute to conditionally compile the macro depending on the target operating system. The macro_rules! macro is used to define the macro, which defines a function that prints a platform-specific message.

In the main function, we call the define_os_specific_function macro to define the appropriate function for the target operating system. We then call the os_specific_function function, which is defined by the macro, to print the platform-specific message.

7. Concurrency

7.1 what is Concurrency

Rust aims to handle concurrent and parallel programming efficiently and safely. With the increasing use of multiple processors, concurrent and parallel programming have become essential.

However, historically they have been difficult and error-prone. Rust hopes to change this by providing a safe and efficient way to handle these types of programming.

7.2 types of Concurrency

7.2.1 Threads

Rust provides native support for threads, which allow you to execute multiple tasks concurrently. Threads can communicate with each other using channels, which provide a safe and efficient way to pass messages between threads.

Example :-

```
use std::thread;
    fn main() {
        let t1 = thread::spawn(|| { for i in 1..=5 {
        println!("Thread 1: {}", i);}  } );
        let t2 = thread::spawn(|| { for i in 1..=5 {
        println!("Thread 2: {}", i);  }  }  );
        t1.join().unwrap();
        t2.join().unwrap();
}
```

Output:-

```
Thread 1: 1
Thread 2: 1
Thread 1: 2
Thread 2: 2
Thread 1: 3
Thread 2: 3
```

Thread 1: 4
Thread 2: 4
Thread 1: 5
Thread 2: 5

7.2.2 Futures

Rust's **async**/**await** syntax allows you to write asynchronous code in a familiar synchronous style. Futures provide a way to represent an operation that may not complete immediately, allowing other code to execute in the meantime

7.2.3 Actors

Rust also provides support for the actor model of concurrency, which is a popular model for building concurrent systems. Actors are isolated units of computation that communicate with each other by sending messages.

7.2.4 Atomic operations

Rust provides atomic operations, which allow you to modify shared data safely without needing to use locks or other synchronization primitives.

8. How to create package

Packages can be created using the Cargo package manager, which is built into Rust. Cargo comes pre-installed with Rust.

We can use cargo to create a package. A package contains one or more crates that provides a set of functionality.

Note: A package can contain many binary crates, but at most only one library crate.

8.1 Creating a Binary Package in Rust

To create a binary package, we can use the cargo command in the terminal.

$ cargo new hello_world --bin
Output:-

Created binary (application) `hello_world` package
We create a binary package hello_world using cargo and the --bin option. It is the default cargo behavior.

Let's look at the contents of the hello_world package.

```
hello_world
├── Cargo.toml
└── src
    └── main.rs
```

Here,

hello_world is the package directory
Cargo.toml is a file that contains metadata about the crate, such as its name, version, and dependencies
src/main.rs is the crate root and contains the source code of the binary package

8.2 Creating a Library Package in Rust

Similarly, we can create a library package in Rust using cargo.

```
$ cargo new hello_world_lib --lib
```

Output:-

```
Created library `hello_world_lib` package
```

We create a library package hello_world_lib using cargo and the --lib option.

Let's look at the contents of the hello_world_lib package.

```
hello_world_lib
├── Cargo.toml
└── src
    └── lib.rs
```

Here,

hello_world_lib is the package directory
Cargo.toml is a file that contains metadata about the crate, such as its name, version, and dependencies
src/lib.rs is the crate root and contains the source code of the library package
A package can contain src/main.rs and src/lib.rs. In this case, it has two crates: a binary and a library, both with the same name as the package. For example,

```
hello_world
├── Cargo.toml
└── src
    ├── lib.rs
    └── main.rs
```

Note: Cargo by convention passes the crate root files to the Rust compiler to build the library or binary.

9. References

1. https://www.programiz.com/rust/crate-and-package
2. https://www.javatpoint.com/rust-tutorial
3. https://www.tutorialspoint.com/rust/index.htm
4. https://doc.rust-lang.org/book/title-page.html
5. https://en.wikipedia.org/wiki/Rust_(programming_language)