blockchained
{'india'}

presents

60 Days of Code with

Polkadot

Instructions:

1. Ensure that the task is completed on the very same day it is assigned. If you face any problem, do share it with us in the group at the same moment, so that the doubts are cleared at the earliest.

2. Every week we will complete one module. Each module is further divided into 5 days. We have kept the weekends free so you can catch up on the work over the weekend and attend the doubts session/workshops which will be organised.

3. If you are not able to complete the task, ensure that they are covered in the very week so that the process is not delayed much.

4. We will have live workshops every week. The schedule of the same will be shared at the start of the week and the same is added to your weekly curriculum. Do ensure you are attending these workshops as they are live sessions by industry experts. It is going to help you only in your journey. If by any reason you are not able to attend, inform the team beforehand.

5. Please submit your assignment as these will be necessary to participate in the hiring track.

6. Please update your daily progress by posting on twitter and sending a message on the group When posting on your twitter accounts please tag us **@blockchainedind** and **@Polkadot**, along with the use of following hashtags:
#60DaysofCode #BuildInPublic #Polkadotdevbootcamp

If you have any questions or doubts, please reach out to us on the group or email us at hriday@blocumen.com and manav@blocumen.com.

blockchained {'india'}

Content

Module 2: Introduction to Rust
(June 11 to June 17)

Day 1: Basics of Rust Programming Language
Day 2: Expression and Utilities
Day 3: Iterators, Collections
Day 3: Workshop 2 with Web3 Foundation
Day 4: Ownership,Traits and crates
Day 5: Concurrency and best practices

blockchained {'india'}

Module 2: Introduction to Rust

Module 2 Overview: https://youtu.be/okZc95Izd5U

Day 1

Basics Rust Programming Language

Rust is a programming language that prioritizes safety, speed, and concurrency. It combines the performance and control of low-level languages with the abstractions of high-level languages. It's a safer alternative for C programmers and offers improved performance without sacrificing expressiveness for Python programmers. Rust's safety checks and memory management occur during compile time, ensuring runtime performance isn't affected. It excels in areas like predictable space and time requirements, embedding in other languages, low-level code (e.g., device drivers, operating systems), and web applications (e.g., crates.io, the Rust package registry site).

Getting Started

Install Rust in windows 11:

1. Open a web browser and go to the official Rust website: https://www.rust-lang.org/

2. On the homepage, you'll see a button that says "Install". Click on it to go to the installation page.

3. Scroll down to the section labelled "Installing Rust". Here, you'll find a button that says "Install Rust". Click on it to download the Rust installer.

4. Once the installer is downloaded, locate the file and double-click on it to run the installer.

5. The installer will start and present you with the Rust setup options. By default, the installer will select the recommended settings. You can customize the installation by choosing different components, but for a basic installation, you can proceed with the default options.

6. Click the "Next" button to continue with the installation.

7. On the next screen, you'll be asked to review and accept the licence terms. Read through the terms and click the checkbox to indicate your acceptance. Then click "Next".

8. The installer will ask you to choose the installation location. You can leave it as the default or choose a different location if desired. Click "Next" to proceed.

9. On the next screen, the installer will ask you to select the components you want to install. Again, the default options are usually sufficient for most users. Click "Next" to continue.

10. The installer will ask you to select the default toolchain. Choose the default option unless you have a specific reason to choose a different toolchain. Click "Next" to proceed.

11. The installer will now show a summary of the installation settings. Review the settings to ensure they are correct. If everything looks good, click "Install" to start the installation process.

12. The installer will begin downloading and installing the necessary components. This may take a few minutes to complete.

13. Once the installation is finished, you'll see a "Installation Complete" screen. Make sure the checkbox to add the Rust installation to the system PATH is selected. This will allow you to run Rust commands from the command prompt or terminal. Click "Finish" to complete the installation.

14. Open a new command prompt or terminal window to verify that Rust is installed correctly. Type `rustc --version` and press Enter. You should see the version number of the Rust compiler printed on the screen.

Congratulations! Rust is now successfully installed on your Windows 10 or 11 system. You can start writing and compiling Rust programs using the installed tools and libraries.

Install Rust in MacBook:

1. Open a terminal window. You can do this by navigating to "Applications" -> "Utilities" -> "Terminal".

2. Install Homebrew, if you haven't already, by entering the following command and pressing Enter:

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

3. Once Homebrew is installed, run the following command to update it to the latest version:

```
brew update
```

4. To install Rust, enter the following command and press Enter:

```
brew install rust
```

5. Homebrew will now download and install Rust and its associated tools and libraries. This process may take a few minutes.

6. After the installation is complete, you can verify the installation by typing `rustc --version` in the terminal and pressing Enter. You should see the version number of the Rust compiler printed on the screen.

Congratulations! Rust is now successfully installed on your macOS system using Homebrew. You can start writing and compiling Rust programs using the installed tools and libraries.

Note: If you encounter any issues during the installation process or want more advanced installation options, you can refer to the official Rust documentation for macOS available at https://www.rust-lang.org/tools/install.

Install Rust on Ubuntu

1. Open a terminal window. You can do this by pressing `Ctrl+Alt+T` or searching for "Terminal" in the Ubuntu application launcher.

2. Update the package lists and upgrade the existing packages by entering the following commands one by one and pressing Enter after each:

```
sudo apt update
sudo apt upgrade
```

3. Install the required dependencies for Rust by entering the following command and pressing Enter:

```
sudo apt install build-essential
```

4. Next, download the official Rust installation script by entering the following command and pressing Enter:

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

5. The command will download the script and start the installation process. You will be prompted with a message about the installation. Press `1` and then Enter to proceed with the default installation.

6. The installation script will now download and install Rust and its associated tools and libraries. This process may take a few minutes.

7. Once the installation is complete, the terminal will display a message indicating that Rust has been installed successfully.

8. To start using Rust, you need to add the `cargo` and `rustc` executables to your system's PATH. This can be done by entering the following command and pressing Enter:

```
source $HOME/.cargo/env
```

9. Finally, verify the installation by typing `rustc --version` in the terminal and pressing Enter. You should see the version number of the Rust compiler printed on the screen.

Congratulations! Rust is now successfully installed on your Ubuntu system. You can start writing and compiling Rust programs using the installed tools and libraries.

Note: If you encounter any issues during the installation process or want more advanced installation options, you can refer to the official Rust documentation for Ubuntu available at https://www.rust-lang.org/tools/install.

Hello world - example

1. Open a text editor of your choice and create a new file. Save it with a `.rs` extension, for example, `main.rs`.

2. In the newly created file, add the following code:

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

3. Save the file.
4. Open a terminal window and navigate to the directory where you saved the `main.rs` file.

5. To compile the Rust program, enter the following command and press Enter:

```
rustc main.rs
```

6. The Rust compiler (`rustc`) will generate an executable file with the same name as the source file (`main`). You should see a new file called `main` or `main.exe` in the same directory.

7. To run the compiled program, enter the following command in the terminal and press Enter:

```
./main
```

Note: On Windows, you may need to use `

```
main.exe
```

instead of `./main` to run the executable.
8. You should see the output
`Hello, World!`
printed in the terminal.

Day -2

Expression and Utilities

Datatypes

Type System in Rust

Sure! Here's a more concise and readable version of the content with correct code formatting:

Rust's type system represents the different types of values supported by the language. It serves two main purposes: checking value validity and providing code hinting and automated documentation.

In Rust, every value has a specific data type. The compiler can automatically infer the type based on the assigned value.

To declare a variable in Rust, use `let` followed by the variable name and an optional explicit type annotation.

Example:

```
fn main() {
   let company_string = "TutorialsPoint";  // String type
   let rating_float = 4.5;               // Float type
   let is_growing_boolean = true;        // Boolean type
   let icon_char = '♥';                  // Unicode character type

   println!("Company name: {}", company_string);
   println!("Company rating: {}", rating_float);
   println!("Is company growing: {}", is_growing_boolean);
   println!("Company icon: {}", icon_char);
}
```

In the above example, the variable data types are inferred from the assigned values. For instance, `company_string` is assigned a string value, so its type is inferred as a string.

The `println!` macro is used to print output to the console. It takes a placeholder `{}` and the variable or constant to be printed. The placeholder will be replaced by the corresponding value when printing.

Scalar Types in Rust

Scalar types in Rust represent single values and include the following four primary types:

1. Integer: Represents whole numbers without fractional components. Integers can be signed (allowing negative and positive values) or unsigned (allowing only positive values). Rust provides various integer types, such as `i8`, `u8`, `i32`, `u32`, etc.

2. Floating-point: Represents numbers with fractional components. Rust has two floating-point types: `f32` (single precision) and `f64` (double precision). The default type for floating-point values is `f64`.

3. Booleans: Has two possible values, `true` and `false`. Booleans are used to represent logical conditions and are declared using the `bool` keyword.

4. Characters: Represents individual Unicode characters, including numbers, alphabets, Unicode characters, and special characters. Characters are declared using the `char` keyword.

Each scalar type has its own range of values and memory requirements.

Integer

Integers in Rust represent whole numbers without fractional components. They can be signed (allowing negative and positive values) or unsigned (allowing only positive values). Rust provides various integer types with different sizes, such as `i8`, `u8`, `i32`, `u32`, etc. There are also architecture-dependent types, `isize` and `usize`, used for indexing collections.

Example:

```
fn main() {
  let result = 10;      // i32 by default
  let age: u32 = 20;
  let sum: i32 = 5 - 15;
  let mark: isize = 10;
  let count: usize = 30;

  println!("Result value: {}", result);
```

```
    println!("Sum is {} and age is {}", sum, age);
    println!("Mark is {} and count is {}", mark, count);
}
```

In the above example, the variable `result` is assigned a value without explicitly specifying its type. Rust infers its type as `i32` by default. Other variables like `age`, `sum`, `mark`, and `count` are explicitly annotated with their respective types.

Integer types have specific value ranges they can represent, based on their size. Signed integers can store numbers from -(2^(n-1)) to 2^(n-1) - 1, where n is the number of bits that variant uses. Unsigned integers can store numbers from 0 to (2^n) - 1.

Rust detects integer overflow and panics at runtime when a value exceeds the defined range for the integer type.

Floating-Point

Floating-point types in Rust represent numbers with fractional components. Rust provides two floating-point types: `f32` (single precision) and `f64` (double precision). By default, floating-point literals are inferred as `f64`.

Example:

```
fn main() {
    let result = 10.00;       // f64 by default
    let interest: f32 = 8.35;
    let cost: f64 = 15000.600;  // double precision

    println!("result value is {}", result);
    println!("interest is {}", interest);
    println!("cost is {}", cost);
}
```

In the above example, the variable `result` is assigned a floating-point value without explicitly specifying its type. Rust infers its type as `f64` by default. The variables `interest` and `cost` are explicitly annotated as `f32` and `f64`, respectively.

Automatic type casting is not allowed in Rust, so you cannot assign a value of one type to a variable of a different type without explicitly converting it.

Booleans

Booleans in Rust represent logical conditions and can have two possible values: `true` or `false`. They are declared using the `bool` keyword.

```
fn main() {
  let is_fun: bool = true;
  println!("Is Rust Programming Fun? {}", is_fun);
}
```

In the above example, the variable `is_fun` is declared as a boolean and assigned the value `true`. It is then printed using the `println!` macro

Characters

Characters in Rust represent individual Unicode characters. Rust's `char` type supports Unicode Scalar Values, allowing it to represent a wide range of characters, including numbers, alphabets, Unicode, and special characters

Example:

```
fn main() {
  let special_character = '@'; // default
  let alphabet: char = 'A';
  let emoji: char = '😁';
  println!("special character is {}", special_character);
  println!("alphabet is {}", alphabet);
  println!("emoji is {}", emoji);
}
```

In the above example, the variables `special_character`, `alphabet`, and `emoji` are declared as characters and assigned their respective values. The characters are then printed using the `println!` macro.

Rust's `char` type represents Unicode Scalar Values, which range from U+0000 to U+D7FF and U+E000 to U+10FFFF inclusive.

Variables

Variables in Rust are used to store and manipulate data. They have a specific name, a type, and a value. Once a variable is declared, its value can be changed, but its type remains the same throughout its scope. Rust follows a static typing approach, where variables must have their types declared at compile time.
To declare a variable in Rust, you use the `let` keyword, followed by the variable name and an optional type annotation. Here's an example:

```
fn main() {
    let message = "Hello, World!";  // Inferred type: &str
    let age: u32 = 30    // Explicit type annotation: unsigned 32-bit integer
    println!("{}", message);
    println!("Age: {}", age);
}
```

In the above example, `message` is a variable of type `&str`, which is a string slice representing a string literal. The type is inferred by the compiler based on the assigned value. `age` is a variable of type `u32`, which is an explicitly annotated unsigned 32-bit integer
Mutable and immutable;
Rust also allows variables to be mutable by using the `mut` keyword. Mutable variables can be reassigned with a new value within the same scope.

Example:

```
fn main() {
    let mut count = 0;   // Mutable variable
    println!("Count: {}", count);
    count = 5; // Reassigning value to the mutable variable
    println!("Updated count: {}", count);
}
```

In this example, `count` is a mutable variable initially assigned the value 0. Later, it is reassigned the value 5. The `mut` keyword indicates that the variable is mutable.

Rust also supports shadowing, where you can declare a new variable with the same name as an existing variable. This allows you to redefine the variable's type and value within the same scope. Here's an example:

```
fn main() {
    let count = 5;
    println!("Count: {}", count);
    let count = count * 2;   // Shadowing the previous `count`
    println!("Updated count: {}", count);
}
```

In this example, a new variable `count` is shadowing the previous `count` variable. The new `count` variable has a different value, obtained by multiplying the previous value by 2.
Remember, variables in Rust are immutable by default, and if you need mutability, you must declare them with the `mut` keyword. Shadowing allows you to reuse variable names while changing their types and values within the same scope.
Rules for Naming a Variable

- The name of a variable can be composed of letters, digits, and the underscore character.
- It must begin with either a letter or an underscore.
- Upper and lowercase letters are distinct because Rust is case-sensitive.

Constants

Constants in Rust are values that are immutable and cannot be changed once they are assigned. They are similar to variables but with a few key differences. Constants must be explicitly typed, and their values must be known at compile time.

To declare a constant in Rust, you use the `const` keyword, followed by the constant name, the type annotation, and the assigned value. Here's an example:

```
const PI: f32 = 3.14;
const MAX_ATTEMPTS: u8 = 5;
fn main() {
    println!("Value of PI: {}", PI);
    println!("Maximum attempts: {}", MAX_ATTEMPTS);
}
```

In the above example, `PI` is a constant of type `f32`, which represents the value of π (pi). `MAX_ATTEMPTS` is a constant of type `u8`, which represents the maximum number of attempts.

Constants are always written in uppercase, and their values cannot be changed. They are useful for expressing values that should remain constant throughout the program and for providing self-documenting names for important values.

Unlike variables, constants can be declared in any scope, including global scope. Here's an example of a constant declared in the global scope:

```
const APP_NAME: &str = "MyApp";

fn main() {
    println!("Welcome to {}", APP_NAME);
}
```

In this example, `APP_NAME` is a constant of type `&str`, which represents the name of an application. It is declared in the global scope and can be accessed from any part of the program.

Constants are evaluated at compile time, and their values are directly substituted into the code. This allows for efficient performance and optimization.

It's important to note that constants differ from mutable variables and immutable variables (declared with `let`) in that their values are always known and cannot be changed. Constants are useful when you have values that should remain constant throughout the execution of the program and when you want to express important values with self-explanatory names.

Strings

In Rust, strings are sequences of Unicode scalar values encoded as UTF-8. They are represented by the `String` type, which is a growable, mutable, and heap-allocated string. Rust also provides string slices (`&str`), which are immutable references to a sequence of UTF-8 bytes.

To create a new `String` in Rust, you can use the `String::from` function or the `to_string` method. Here's an example:

```
fn main() {
    let hello = String::from("Hello, ");
    let name = "Alice";
    let greeting = hello + name;

    println!("{}", greeting);
}
```

In this example, a new `String` called `hello` is created using `String::from`. The `name` variable is a string slice (`&str`). We can concatenate the two using the `+` operator, which appends the `name` to the `hello` string, resulting in a new `String` called `greeting`. Finally, the `greeting` is printed to the console.

String Functions
Rust provides various methods and functions for working with strings. Here are a few commonly used string functions:

len()
Returns the length of a string in bytes.

```
fn main() {
    let message = String::from("Hello, Rust!");
    let length = message.len();

    println!("Length: {}", length);
}
```

is_empty()
Returns if a string is empty.

```
fn main() {
    let message = String::from("");
    let is_empty = message.is_empty();

    println!("Is empty: {}", is_empty);
}
```

chars()

Returns an iterator over the Unicode characters of a string.

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

    for c in message.chars() {
        println!("{}", c);
    }
}
```

to_uppercase()

Converts a string to uppercase.

```
fn main() {
    let message = String::from("Hello, Rust!");
    let uppercase = message.to_uppercase();
    println!("Uppercase: {}", uppercase);
}
```

These are just a few examples of the string functions available in Rust. The Rust standard library provides a rich set of methods and functions for manipulating and working with strings, including substring operations, searching, replacing, and more.

Remember that Rust's string type, `String`, is mutable and growable, while string slices, `&str`, are immutable. The Rust compiler provides many safety guarantees when working with strings, such as preventing buffer overflows and handling Unicode encoding properly.

Operators

Operators in Rust are symbols or keywords that perform various operations on values, such as arithmetic, logical, bitwise, assignment, comparison, and more. They allow you to manipulate and combine values in meaningful ways.

Arithmetic Operators
Rust provides the following arithmetic operators:

- Addition (`+`): Adds two values together.
- Subtraction (`-`): Subtracts one value from another.
- Multiplication (`*`): Multiplies two values.
- Division (`/`): Divides one value by another.
- Remainder (`%`): Calculates the remainder of the division.
- Unary Negation (`-`): Negates a value.

Here's an example showcasing arithmetic operators:

```
fn main() {
    let a = 5;
    let b = 3;

    let sum = a + b;
    let difference = a - b;
    let product = a * b;
    let quotient = a / b;
    let remainder = a % b;
    let negation = -a;

    println!("Sum: {}", sum);
    println!("Difference: {}", difference);
    println!("Product: {}", product);
    println!("Quotient: {}", quotient);
    println!("Remainder: {}", remainder);
    println!("Negation: {}", negation);
}
```

Comparison Operators

Rust provides the following comparison operators:

- Equal to (`==`): Checks if two values are equal.
- Not equal to (`!=`): Checks if two values are not equal.
- Greater than (`>`): Checks if one value is greater than another.
- Greater than or equal to (`>=`): Checks if one value is greater than or equal to another.
- Less than (`<`): Checks if one value is less than another.

- Less than or equal to (`<=`): Checks if one value is less than or equal to another.

Here's an example showcasing comparison operators:

```
fn main() {
    let a = 5;
    let b = 3;

    let equals = a == b;
    let not_equals = a != b;
    let greater_than = a > b;
    let greater_than_or_equal = a >= b;
    let less_than = a < b;
    let less_than_or_equal = a <= b;

    println!("Equals: {}", equals);
    println!("Not Equals: {}", not_equals);
    println!("Greater Than: {}", greater_than);
    println!("Greater Than or Equal: {}", greater_than_or_equal);
    println!("Less Than: {}", less_than);
    println!("Less Than or Equal: {}", less_than_or_equal);
}
```

Logical Operators
Rust provides the following logical operators:

- Logical AND (`&&`): Returns `true` if both operands are `true`.
- Logical OR (`||`): Returns `true` if at least one of the operands is `true`.
- Logical NOT (`!`): Negates a boolean value.

Here's an example showcasing logical operators:

```
fn main() {
    let a = true;
    let b = false;
    let and_result = a && b;
    let or_result = a || b;
    let not_a = !a;
    println!("AND Result: {}", and_result);
    println!("OR Result: {}", or_result);
```

```
    println!("NOT A: {}", not_a);
}
```

Assignment Operators
Rust provides shorthand assignment operators for performing operations and assigning the result to a variable. For example:

- `+=` performs addition and assignment.
- `-=` performs subtraction and assignment.
- `*=` performs multiplication and assignment.
- `/=` performs division and assignment.
- `%=`` performs remainder

and assignment.

Here's an example showcasing assignment operators:

```
fn main() {
    let mut a = 5;
    a += 3;  // equivalent to a = a + 3
    println!("a: {}", a);
    a -= 2;  // equivalent to a = a - 2
    println!("a: {}", a);
    a *= 4;  // equivalent to a = a * 4
    println!("a: {}", a);
    a /= 2;  // equivalent to a = a / 2
    println!("a: {}", a);
    a %= 3;  // equivalent to a = a % 3
    println!("a: {}", a);
}
```

These are just a few examples of the operators available in Rust. Rust supports many other operators, including bitwise operators, range operators, logical assignment operators, and more. Understanding and using operators effectively is crucial for performing computations and manipulating values in your Rust programs.

Bitwise Operators

Rust provides bitwise operators for manipulating individual bits in integers. These operators work on the binary representation of the numbers.

- Bitwise AND (`&`): Performs a bitwise AND operation on two integers, resulting in a new integer with bits set where both operands have bits set.
- Bitwise OR (`|`): Performs a bitwise OR operation on two integers, resulting in a new integer with bits set where at least one of the operands has bits set.
- Bitwise XOR (`^`): Performs a bitwise XOR (exclusive OR) operation on two integers, resulting in a new integer with bits set where exactly one of the operands has bits set.
- Bitwise NOT (`!`): Performs a bitwise NOT operation on an integer, flipping all the bits.

Here's an example showcasing bitwise operators:

```
fn main() {
    let a = 0b1010;  // binary representation of 10
    let b = 0b1100;  // binary representation of 12
    let and_result = a & b;
    let or_result = a | b;
    let xor_result = a ^ b;
    let not_a = !a;
    println!("AND Result: {:b}", and_result);  // Output: 0b1000
    println!("OR Result: {:b}", or_result);    // Output: 0b1110
    println!("XOR Result: {:b}", xor_result);  // Output: 0b0110
    println!("NOT A: {:b}", not_a);          // Output: -1011 (two's complement
representation)
}
```

Range Operator
Rust provides the range operator `..` and `..=` for creating ranges of values.

- `start..end` creates a range from `start` (inclusive) to `end` (exclusive).
- `start..=end` creates a range from `start` (inclusive) to `end` (inclusive).

Here's an example showcasing range operators:

```
fn main() {
    // Exclusive range
    for num in 1..5 {
        println!("{}", num);
    }
    // Output: 1, 2, 3, 4

    // Inclusive range
    for num in 1..=5 {
        println!("{}", num);
    }
    // Output: 1, 2, 3, 4, 5
}
```

Deref Operator (`*`)

The dereference operator `*` is used to access the value pointed to by a reference or a smart pointer.

```
fn main() {
    let number = 42;
    let number_ref = &number;
    println!("{}", *number_ref);  // Output: 42
}
```

Indexing Operator (`[]`)

The indexing operator `[]` is used to access individual elements in arrays, vectors, and other types that support indexing.

```
fn main() {
    let fruits = ["apple", "banana", "orange"];
    println!("{}", fruits[1]);  // Output: banana
}
```

Functions

Functions in Rust are blocks of reusable code that perform specific tasks. They allow you to encapsulate functionality, modularize your code, and promote code reusability. Let's explore function definition, function calls, and the concepts of call by value and call by reference in Rust.

Function Definition
In Rust, you can define functions using the `fn` keyword followed by the function name, parentheses for parameters, and a return type annotation (if applicable). Here's the general syntax of a function definition:

```
fn function_name(parameter1: Type1, parameter2: Type2, ...) -> ReturnType {
    // Function body
    // Code to perform the desired functionality
    // Return statement (if applicable)
}
```

Here's an example of a simple function that calculates the square of a number:

```
fn square(num: i32) -> i32 {
    let result = num * num;
    result  // Implicit return
}
```

Function Call
To invoke a function in Rust, you simply write the function name followed by parentheses and pass the required arguments (if any).

Here's an example:

```
fn main() {
    let number = 5;
    let square_result = square(number);
```

```
    println!("Square of {} is {}", number, square_result);
}
```

The function `square()` is called with the `number` variable as an argument. The return value is stored in the `square_result` variable and then printed.

Call by Value:
In Rust, function arguments are passed by value by default. It means that when you pass a value to a function, a copy of that value is made, and any modifications made to the parameter within the function do not affect the original value in the calling code.

Here's an example:

```
fn add_one(num: i32) -> i32 {
    num + 1  // Increment the parameter by 1
}

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

In the above example, the `add_one()` function takes an `i32` parameter and increments it by 1. However, the original value of `number` in the `main()` function remains unchanged.

Call by Reference

If you want to modify the original value of a variable within a function, you can use references. Rust provides the `&` symbol to create references to values. By passing a reference to a function, you can enable call by reference behavior.

Here's an example:

```
fn add_one_by_ref(num: &mut i32) {
    *num += 1;  // Increment the value pointed by the reference
}
fn main() {
    let mut number = 5;
    add_one_by_ref(&mut number);
    println!("Modified value: {}", number);
}
```

In the above example, the `add_one_by_ref()` function takes a mutable reference to an `i32` parameter (`&mut i32`). By using `&mut` before the parameter type, we indicate that we want to pass the reference as mutable. The `*num += 1` line increments the value pointed to by the reference. As a result, the original `number` variable in the `main()` function is modified.

Call by value and call by reference provide different ways to pass arguments to functions, depending on whether you want to modify the original value or work with a copy. Understanding these concepts is crucial when designing Rust functions and deciding how arguments should be handled.

Day -3: Iterators and Collections

Conditional Statements

Decision-making in Rust is done using conditional statements, primarily `if`, `if-else`, and `match` expressions. These constructs allow you to control the flow of your program based on certain conditions. Let's explore each of these in detail.

`if` statement
The `if` statement is used to execute a block of code if a condition is true. The syntax is as follows:

```
if condition {
    // Code to execute if the condition is true
} else {
    // Code to execute if the condition is false
}
```

Here's an example:

```
fn main() {
    let number = 5;
    if number < 0 {
        println!("Number is negative");
    } else {
        println!("Number is non-negative");
    }
}
```

In this example, if the `number` is less than 0, the first block of code will execute, printing "Number is negative". Otherwise, the second block of code will execute, printing "Number is non-negative".

`else-if` statement:

You can chain multiple conditions using `else if` statements to handle multiple scenarios. The syntax is as follows:

```
if condition1 {
   // Code to execute if condition1 is true
} else if condition2 {
   // Code to execute if condition2 is true
} else {
   // Code to execute if both condition1 and condition2 are false
}
```

Here is an Example:

```
fn main() {
    let number = 10;
    if number < 0 {
        println!("Number is negative");
    } else if number > 0 {
        println!("Number is positive");
    } else {
        println!("Number is zero");
    }
}
```

In this example, the program checks multiple conditions to determine if the number is negative, positive, or zero.

`match` expression

The `match` expression is used for pattern matching and provides a powerful way to perform different actions based on the value of a variable. It's especially useful when dealing with multiple cases. The syntax is as follows:

```
match variable {
   pattern1 => {
      // Code to execute if variable matches pattern1
   },
   pattern2 => {
      // Code to execute if variable matches pattern2
   },
   _ => {
      // Code to execute if no patterns match
```

```
    }
}
```

Here's an example:

```
fn main() {
    let fruit = "apple";
    match fruit {
        "apple" => println!("It's an apple"),
        "banana" => println!("It's a banana"),
        _ => println!("It's something else"),
    }
}
```

In this example, the program matches the value of `fruit` with different patterns. If `fruit` is "apple", it will print "It's an apple". If `fruit` is "banana", it will print "It's a banana". Otherwise, it will print "It's something else".

These decision-making constructs allow you to control the flow of your Rust program based on specific conditions. By using `if` statements, `if-else` statements, and `match` expressions, you can create dynamic and flexible code that responds to different scenarios.

Loops

Loops in Rust are used to repeat a block of code until a certain condition is met. Rust provides three main types of loops: `loop`, `while`, and `for`. Let's explore each of them in detail along with code examples.

`loop` loop

The `loop` loop executes a block of code repeatedly until an explicit `break` statement is encountered. It's useful when you need to create an infinite loop or when you want to break out of the loop based on a condition. Here's the syntax:

```
loop {
    // Code to repeat indefinitely until a break statement
    // is encountered or a condition is met.
```

```
    if condition {
        break;
    }
}
```

Example:

```
fn main() {
    let mut count = 0;
    loop {
        println!("Current count: {}", count);
        count += 1;
        if count == 5 {
            break;
        }
    }
}
```

In this example, the `loop` loop repeats the code inside indefinitely until the `count` variable reaches 5. When `count` is equal to 5, the `break` statement is encountered, and the loop is exited.

`while` loop

The `while` loop executes a block of code as long as a condition is true. It's used when you want to repeat a block of code based on a condition. Here's the syntax:

```
while condition {
    // Code to execute as long as the condition is true
}
```

Example:

```
fn main() {
    let mut count = 0;
    while count < 5 {
        println!("Current count: {}", count);
        count += 1;
    }
}
```

In this example, the `while` loop repeats the code inside as long as the `count` variable is less than 5. The `count` variable is incremented with each iteration.

`for` loop

The `for` loop is used to iterate over a sequence or a range of values. It's commonly used when you know the exact number of iterations. Here's the syntax:

```
for variable in iterable {
  // Code to execute for each element in the iterable
}
```

Example:

```
fn main() {
    let fruits = ["apple", "banana", "orange"];

    for fruit in fruits.iter() {
        println!("Fruit: {}", fruit);
    }
}
```

In this example, the `for` loop iterates over each element in the `fruits` array, and the `fruit` variable takes the value of each element in each iteration.

These loop constructs allow you to repeat code execution based on different conditions or for iterating over a sequence of values. By using the `loop` loop, `while`

loop, and `for` loop, you can control the flow of your program and perform repetitive tasks efficiently in Rust.

Tuples

Tuples in Rust are a way to group together multiple values of different types into a single compound value. They are useful when you want to store and manipulate a fixed set of values as a single entity. Tuples are created by enclosing the values in parentheses and separating them with commas. Let's dive into tuples in more detail with code examples.

Tuple Creation

Tuples are created by enclosing the values in parentheses and separating them with commas.

Here's an example:

```
fn main() {
    let person = ("John", 25, true);
    println!("{:?}", person);
}
```

In this example, a tuple `person` is created with three elements: a string `"John"`, an integer `25`, and a boolean `true`. The `println!` macro is used to print the tuple.

Tuple Access

You can access individual elements of a tuple using dot notation followed by the index. The index starts from 0.

Here's an example:

```
fn main() {
    let person = ("John", 25, true);
    println!("Name: {}", person.0);
    println!("Age: {}", person.1);
```

```
    println!("Is Active: {}", person.2);
}
```

In this example, we access the first element of the tuple using `person.0`, the second element using `person.1`, and the third element using `person.2`.

Tuple Destructuring

You can also destructure a tuple by assigning its elements to individual variables.

Here's an example:

```
fn main() {
    let person = ("John", 25, true);
    let (name, age, is_active) = person;
    println!("Name: {}", name);
    println!("Age: {}", age);
    println!("Is Active: {}", is_active);
}
```

In this example, we destructure the `person` tuple and assign its elements to variables `name`, `age`, and `is_active`. We can then use these variables independently.

Returning Tuples from Functions

Functions in Rust can return tuples, allowing you to return multiple values from a function.

Here's an example:

```
fn get_person() -> (String, u32, bool) {
   let name = String::from("John");
   let age = 25;
   let is_active = true;
   (name, age, is_active)
}
```

```
fn main() {
    let person = get_person();
    println!("Name: {}", person.0);
    println!("Age: {}", person.1);
    println!("Is Active: {}", person.2);
}
```

In this example, the `get_person()` function returns a tuple containing the name, age, and is_active status of a person. We then assign the returned tuple to the `person` variable and access its elements.

Tuples in Rust are a convenient way to group multiple values into a single compound value. They allow you to work with fixed sets of values and can be accessed, destructured, and returned from functions.

Arrays

Arrays in Rust are a fixed-size collection of elements of the same type. They are useful when you need to store and manipulate a fixed number of elements. In Rust, arrays have a fixed length determined at compile-time and cannot be resized. Let's explore arrays in more detail with code examples.

Array Creation

Arrays in Rust are created by specifying the type and the number of elements enclosed in square brackets.

Here's an example:

```
fn main() {
    let numbers: [i32; 4] = [10, 20, 30, 40];
    println!("{:?}", numbers);
}
```

In this example, an array `numbers` of type `i32` with a length of 4 is created. It contains the elements `10`, `20`, `30`, and `40`. The `println!` macro is used to print the array.

Array Access

You can access individual elements of an array using square brackets and the index. The index starts from 0. Here's an example:

```
fn main() {
    let numbers: [i32; 4] = [10, 20, 30, 40];
    println!("First element: {}", numbers[0]);
    println!("Second element: {}", numbers[1]);
}
```

In this example, we access the first element of the array using `numbers[0]` and the second element using `numbers[1]`.

Array Length

You can get the length of an array using the `len()` method.

Here's an example:

```
fn main() {
    let numbers: [i32; 4] = [10, 20, 30, 40];
    println!("Array length: {}", numbers.len());
}
```

In this example, the `len()` method is called on the `numbers` array to get its length, which is then printed.

Array Iteration

You can iterate over the elements of an array using a `for` loop or other iterator methods.

Here's an example:

```
fn main() {
    let numbers: [i32; 4] = [10, 20, 30, 40];
    for num in numbers.iter() {
```

```
        println!("Number: {}", num);
    }
}
```

In this example, we use the `iter()` method to create an iterator over the elements of the `numbers` array, and then iterate over them using a `for` loop.

Arrays in Rust provide a way to store and access a fixed number of elements of the same type. They are created with a fixed length at compile-time and cannot be resized. You can access individual elements, get the length of an array, and iterate over its elements using various Rust features.

Structure

Structures, also known as structs, in Rust allow you to define custom data types by grouping together multiple related values of different types into a single entity. They are similar to classes or records in other programming languages. Let's explore structures in more detail with code examples.

Structure Definition

You can define a structure in Rust using the `struct` keyword followed by the name of the structure and the list of fields enclosed in curly braces. Here's an example:

```
struct Person {
    name: String,
    age: u32,
    is_active: bool,
}
```

In this example, we define a structure named `Person` with three fields: `name` of type `String`, `age` of type `u32`, and `is_active` of type `bool`.

Structure Instantiation

To create an instance of a structure, you can use the structure's name followed by the field names and their corresponding values.

Here's an example:

```
fn main() {
    let person = Person {
        name: String::from("John"),
        age: 25,
        is_active: true,
    };
    println!("{:?}", person);
}
```

In this example, we create an instance of the `Person` structure named `person` and assign values to its fields.

Structure Field Access

You can access the fields of a structure using dot notation followed by the field name.

Here's an example:

```
fn main() {
    let person = Person {
        name: String::from("John"),
        age: 25,
        is_active: true,
    };
    println!("Name: {}", person.name);
    println!("Age: {}", person.age);
}
```

In this example, we access the `name` and `age` fields of the `person` structure using dot notation.

Mutable Structures

You can make a structure mutable by using the `mut` keyword. This allows you to modify the values of its fields.

Here's an example:

```
fn main() {
    let mut person = Person {
        name: String::from("John"),
        age: 25,
        is_active: true,
    };
    person.age = 30; // Modify the value of the age field
    println!("Modified Age: {}", person.age);
}
```

In this example, we make the `person` structure mutable using the `mut` keyword, and then modify the value of its `age` field.

Structures in Rust provide a way to define custom data types by grouping together related values. You can create instances of structures, access their fields, and modify their values. Structures are a powerful tool for organizing and manipulating data in Rust.

Slices

Slices in Rust provide a way to reference a contiguous sequence of elements in a collection (such as an array, vector, or string) without owning the data. Slices are useful when you want to work with a portion of the data rather than the whole collection. Let's explore slices in more detail with code examples.

Slice Definition

A slice in Rust is represented by a range of indices that specify the start and end positions of the slice. The syntax for creating a slice is `&[start..end]`, where `start` is the index of the first element to include in the slice, and `end` is the index of the first element to exclude from the slice.

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

```
}
```

In this example, we define an array `numbers` and create a slice `slice` that includes elements from index `1` up to, but not including, index `4`. The slice references the original array's data.

Immutable Slices

By default, slices are immutable, meaning you cannot modify the elements they reference. This ensures data integrity. Here's an example:

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

    // Attempting to modify the slice will result in a compilation error
    // slice[0] = 10;
}
```

In this example, trying to modify the elements of the `slice` will result in a compilation error because the slice is immutable.

Mutable Slices:

You can create mutable slices by using the `&mut` keyword. Mutable slices allow you to modify the referenced elements.

Here's an example:

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

    // Modifying the elements of the slice
    slice[0] = 10;
    slice[1] = 20;
    slice[2] = 30;
    println!("{:?}", slice);
}
```

In this example, we create a mutable slice `slice` that references a portion of the `numbers` array. We can modify the elements of the slice without changing the original array.

Slices provide a flexible way to work with portions of data in collections. They allow you to reference elements without taking ownership of the data and provide both immutable and mutable views of the data. Slices are commonly used when you need to pass a subset of a collection to a function or when you want to manipulate a portion of a collection efficiently.

Enums

Enums, short for enumerations, in Rust are a powerful feature that allows you to define a type by enumerating its possible values. Enums are useful when you want to represent a value that can be one of a limited set of options. Let's explore enums in more detail with code examples.

Enum Definition:

To define an enum in Rust, you use the `enum` keyword followed by the name of the enum and a list of its variants enclosed in curly braces.

Here's an example:

```
enum Color {
    Red,
    Green,
    Blue,
}
```

In this example, we define an enum named `Color` with three variants: `Red`, `Green`, and `Blue`. Each variant represents a possible value of the `Color` enum.

Enum Values

You can create values of an enum by using its variants as constructors.

Here's an example:

```
fn main() {
    let red = Color::Red;
    let green = Color::Green;
    let blue = Color::Blue;

    println!("{:?}", red);
    println!("{:?}", green);
    println!("{:?}", blue);
}
```

In this example, we create variables `red`, `green`, and `blue` of type `Color` by using the enum variants as constructors. We then print their values using `println!` macro.

Enum with Data

Enums can also have associated data with each variant. This allows you to attach additional values to the enum variants.

Here's an example:

```
enum Shape {
    Circle(f64),
    Rectangle(f64, f64),
    Triangle(f64, f64, f64),
}
```

In this example, we define an enum named `Shape` with three variants: `Circle` that takes a single `f64` value (radius), `Rectangle` that takes two `f64` values (length and width), and `Triangle` that takes three `f64` values (side lengths).

Enum Pattern Matching

Pattern matching is a powerful feature in Rust that allows you to handle different cases of an enum.

Here's an example:

```
fn print_area(shape: Shape) {
    match shape {
        Shape::Circle(radius) => {
            let area = std::f64::consts::PI * radius * radius;
            println!("Area of the circle: {}", area);
        }
        Shape::Rectangle(length, width) => {
            let area = length * width;
            println!("Area of the rectangle: {}", area);
        }
        Shape::Triangle(side1, side2, side3) => {
            // Calculate area using Heron's formula
            let s = (side1 + side2 + side3) / 2.0;
            let area = (s * (s - side1) * (s - side2) * (s - side3)).sqrt();
            println!("Area of the triangle: {}", area);
        }
    }
}
```

In this example, we define a function `print_area` that takes a `Shape` enum and uses pattern matching to calculate and print the area based on the variant of the shape.

Enums in Rust provide a concise way to represent a value that can be one of a limited set of options. They can be used with or without associated data, and pattern matching allows you to handle different cases of the enum effectively. Enums are a powerful tool for expressing intent and providing type safety in Rust programs.

Collections

Collections in Rust are data structures that can store multiple values of different types. They provide dynamic storage and can grow or shrink as needed. Rust provides several collection types, including vectors, arrays, strings, and hash maps.

Vectors

Vectors, represented by the `Vec<T>` type, are dynamically sized arrays in Rust. They allow you to store and manipulate a variable number of elements.

```
fn main() {
    let mut numbers: Vec<i32> = Vec::new(); // Create an empty vector
    numbers.push(1); // Add an element to the vector
    numbers.push(2);
    numbers.push(3);

    println!("{:?}", numbers); // Print the vector

    // Access elements of the vector
    let first_element = numbers[0];
    println!("First element: {}", first_element);

    // Iterate over the vector
    for number in &numbers {
        println!("{}", number);
    }
}
```

In this example, we create a vector `numbers` and add elements to it using the `push` method. We can access elements using indexing and iterate over the vector using a `for` loop.

Arrays

Arrays in Rust are fixed-size collections with elements of the same type. They are represented by `[T; N]`, where `T` is the type of the elements, and `N` is the number of elements.

```
fn main() {
    let numbers: [i32; 3] = [1, 2, 3]; // Create an array
    println!("{:?}", numbers); // Print the array
    // Access elements of the array
    let first_element = numbers[0];
    println!("First element: {}", first_element);
    // Iterate over the array
    for number in &numbers {
        println!("{}", number);
    }
}
```

In this example, we create an array `numbers` with three elements and access elements using indexing. We can also iterate over the array.

Strings:

Strings in Rust represent a sequence of Unicode scalar values. Rust has two types of strings: `String` (a heap-allocated string) and `&str` (a string slice).

```
fn main() {
    let mut hello = String::from("Hello"); // Create a String
    hello.push_str(", world!"); // Append to the string

    println!("{}", hello); // Print the string

    // Iterate over the characters of the string
    for c in hello.chars() {
        println!("{}", c);
    }
}
```

In this example, we create a `String` and append a string slice using `push_str`. We can iterate over the characters of the string using `chars`.

Hash Maps

Hash maps, represented by the `HashMap<K, V>` type, store key-value pairs and allow efficient lookup, insertion, and removal based on the keys.

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

fn main() {
    let mut scores = HashMap::new(); // Create an empty hash map
    scores.insert(String::from("Alice"), 100); // Insert a key-value pair
    scores.insert(String::from("Bob"), 90);

    println!("{:?}", scores); // Print the hash map

    // Access a value using the key
    let alice_score = scores.get("Alice");
```

```
    println!("Alice's score: {:?}", alice_score);

    // Iterate over the key-value pairs
    for (name, score) in &scores {
        println!("{}: {}", name, score);
    }
}
```

In this example, we create a hash map `scores`, insert key-value pairs, and access values using keys.

We can also iterate over the key-value pairs using a `for` loop.

Modules

Modules in Rust allow you to organize code into separate namespaces and scopes. They help in organizing and separating concerns in larger projects.

Here's an example of using modules in Rust:

```
// Define a module named 'my_module'
mod my_module {
    // Define a function inside the module
    pub fn greet() {
        println!("Hello from my_module!");
    }
}

// Use the function from the module
fn main() {
    my_module::greet();
}
```

In this example, we define a module named `my_module` using the `mod` keyword. Inside the module, we define a function `greet()`. By prefixing the function with `pub`, we make it accessible outside the module. In the `main` function, we call the `greet()` function using the module path `my_module::greet()`.

Modules help in organizing code, preventing naming conflicts, and providing encapsulation. They are an essential tool for structuring Rust projects and improving code maintainability.

Workshop Details

Second Workshop by Bader Youssef from Web 3 Foundation at 7:00 pm IST. Joining Link:

https://us02web.zoom.us/j/86366812314?pwd=SFNmQUlvT0tRaHlDaVYrN3l5bzJVQT09

Day -4 Ownership,Traits and crates

Ownership

Ownership is a unique feature of Rust that helps enforce memory safety and eliminates the need for manual memory management like explicit allocation and deallocation.

Ownership Rules in Rust:

1. Each value in Rust has a variable that is called its owner.
2. There can only be one owner at a time.
3. When the owner goes out of scope, the value will be dropped.

Ownership Example

```
fn main() {
    let s = String::from("Hello"); // s is the owner of the string
    println!("{}", s); // s can be used here
   // When s goes out of scope, the memory will be freed    automatically
} // s is dropped here
```

In this example, the variable `s` owns the `String` value "Hello". The `String` type is a dynamically allocated string that is capable of storing and manipulating text. When `s` goes out of scope at the end of the block, Rust automatically calls the `drop` function, and the memory occupied by the string is freed.

Ownership Transfer

```
fn main() {
    let s1 = String::from("Hello");
    let s2 = s1; // Ownership of s1 is transferred to s2

    // println!("{}", s1); // Error! s1 is no longer valid

    println!("{}", s2); // s2 can be used here
}
```

In this example, ownership of the string is transferred from `s1` to `s2` using a simple assignment. After the transfer, `s1` is no longer valid, and attempting to use it will

result in a compile-time error. However, `s2` can still be used because it now owns the string.

Ownership and Functions:

```
fn main() {
    let s = String::from("Hello");
    take_ownership(s); // Ownership of s is transferred to the function

    // println!("{}", s); // Error! s is no longer valid
}

fn take_ownership(some_string: String) {
    println!("{}", some_string); // some_string can be used here
} // some_string is dropped here
```

In this example, the `take_ownership` function takes ownership of the `String` parameter. The ownership transfer happens when the function is called. After the transfer, the original variable `s` in the `main` function is no longer valid, as it has been moved to `some_string` inside the function.

Ownership and References:

```
fn main() {
    let s = String::from("Hello");
    calculate_length(&s); // Pass a reference to s

    println!("{}", s); // s can still be used here
}

fn calculate_length(s: &String) -> usize {
    s.len() // s is borrowed here
}
```

In this example, we pass a reference to the `String` `s` to the `calculate_length` function using the `&` symbol. The function takes a reference (`&String`) instead of taking ownership. By using references, we can pass values to functions without transferring ownership. The borrowed value `s` can be used within the function

without taking ownership, and the original variable `s` in the `main` function remains valid.

Ownership is a powerful concept in Rust that ensures memory safety and eliminates common bugs such as use-after-free and double-free errors. By following the ownership rules, Rust guarantees memory safety without the need for a garbage collector or manual memory management.

Borrowing

Borrowing is a key concept in Rust that allows you to temporarily borrow a reference to a value without taking ownership. This enables multiple references to access the same data while maintaining memory safety and avoiding data races.

Borrowing Rules in Rust

1. A borrowed reference cannot outlive the owner.
2. Only one mutable reference or multiple immutable references can exist at a time.
3. Mutable references cannot coexist with any other references.

Immutable Borrowing Example:

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

    // Immutable borrowing - reference to the value is borrowed
    let len = calculate_length(&s);

    println!("Length of '{}' is {}", s, len);
}

fn calculate_length(s: &String) -> usize {
    s.len()
}
```

In this example, the `calculate_length` function takes an immutable reference to a `String` as its parameter using `&`. This allows the function to borrow the value without taking ownership. The borrowed reference `s` can be used to access the

value of the string, in this case, to calculate its length. The original `String` `s` in the `main` function remains valid and can still be used after the function call.

Mutable Borrowing Example:

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

    // Mutable borrowing - mutable reference to the value is borrowed
    change_string(&mut s);

    println!("Modified string: {}", s);
}

fn change_string(s: &mut String) {
    s.push_str(", World!");
}
```

In this example, the `change_string` function takes a mutable reference to a `String` using `&mut`. This allows the function to borrow a mutable reference to the value and modify it. The borrowed mutable reference `s` can be used to append the string with ", World!". Since `s` is mutable, the modification is allowed. The original `String` `s` in the `main` function is modified and reflected in the output.

Borrowing allows you to pass references to values without transferring ownership, enabling efficient and safe sharing of data. By following the borrowing rules, Rust ensures that references are used correctly and prevents common issues like data races and dangling references.

Traits

Traits are a feature in Rust that define shared behavior across different types. They allow you to define a set of methods that can be implemented by various types, enabling code reuse and providing a form of polymorphism.

Defining a Trait

```
// Define a trait named `Drawable` with a single method `draw`
trait Drawable {
```

```
    fn draw(&self);
}
```

In this example, we define a trait named `Drawable` using the `trait` keyword. The trait declares a single method `draw` that takes a reference to `self`. This means that any type implementing the `Drawable` trait must provide an implementation of the `draw` method.

Implementing a Trait

```
// Implement the `Drawable` trait for the `Circle` struct
struct Circle {
    radius: f64,
}
impl Drawable for Circle {
    fn draw(&self) {
        println!("Drawing a circle with radius {}", self.radius)
    }
}
```

In this example, we define a `Circle` struct and implement the `Drawable` trait for it using the `impl` keyword. The implementation provides the required `draw` method that prints a message about drawing a circle with the given radius.

Using Traits

```
fn draw_shape(shape: &dyn Drawable) {
    shape.draw();
}

fn main() {
    let circle = Circle { radius: 5.0 };
    draw_shape(&circle);
}
```

In this example, we define a function `draw_shape` that takes a reference to any type that implements the `Drawable` trait. We can pass different types that implement `Drawable` to this function, and it will invoke the `draw` method on the given object.

In the `main` function, we create a `Circle` object and pass it to `draw_shape`. Since `Circle` implements `Drawable`, it can be used in place of a `Drawable` trait object.

Traits provide a way to define common behavior across types and enable code reuse through polymorphism. They allow you to write generic code that works with any type that satisfies the requirements of the trait. Traits are a powerful tool for designing flexible and reusable code in Rust.

Crates

In Rust, crates are a unit of code distribution and packaging. They allow you to share code libraries, modules, and applications with others. Crates can be published on the central package registry called crates.io or used privately within projects.

Creating a Crate:

To create a new crate, you'll need to follow these steps:

1. Initialize a new Rust project using the `cargo new` command:

```
$ cargo new my_crate
```

2. Navigate into the crate directory

```
$ cd my_crate
```

3. Edit the `Cargo.toml` file to specify the crate's metadata, dependencies, and other configuration options.

4. Write your code in the appropriate source files within the `src` directory.

5. Build the crate using `cargo build`:

```
$ cargo build
```

6. If everything compiles successfully, you can find the built crate in the `target/debug` directory.

Using Crates:

To use an existing crate in your Rust project, you'll need to follow these steps:

1. Add the crate dependency to your `Cargo.toml` file under the `[dependencies]` section:

```
[dependencies]
my_crate = "0.1.0"
```

2. Import and use the crate in your code:

```
use my_crate::some_module;

fn main() {
    some_module::some_function();
}
```

3. Build and run your project using `cargo build` and `cargo run` respectively.

Publishing a Crate

To publish your crate on crates.io, you'll need to follow these steps:

1. Create an account on crates.io if you haven't already.

2. Update the `Cargo.toml` file with the appropriate metadata, including the crate's name, version, author, and description.

3. Create a new version of your crate using `cargo package`

```
$ cargo package
```

4. Publish the crate using `cargo publish`:

```
$ cargo publish
```

Note: Publishing a crate requires ownership of the crate name on crates.io.

By publishing your crate, others can easily use and depend on it in their own Rust projects. Crates.io serves as a central repository for discovering, sharing, and managing Rust crates.

Remember to adhere to best practices, such as following semantic versioning for versioning your crate, documenting your code, and maintaining compatibility and stability to ensure a smooth experience for crate users.

Using a Existing Crate

Here's an example of how to use an existing crate from crates.io in a Rust project:

1. Add the crate dependency to your `Cargo.toml` file. Let's use the popular `rand` crate as an example:

```
[dependencies]
rand = "0.8.4"
```

2. Import and use the crate in your code:

```
use rand::Rng;

fn main() {
    let mut rng = rand::thread_rng();
    let random_number: u32 = rng.gen_range(1..=10);
    println!("Random number: {}", random_number);
}
```

In this example, we import the `Rng` trait from the `rand` crate, which provides methods for generating random numbers. We then create an instance of the random number generator using `rand::thread_rng()`, and finally, generate a random number within the range of 1 to 10 using the `gen_range` method.

3. Build and run your project using `cargo build` and `cargo run` respectively:

```
$ cargo build
$ cargo run
```

The project will fetch the `rand` crate from crates.io and build it along with your code. The random number will be generated and printed when you run the project.

By adding the desired crate dependency to your `Cargo.toml` file and importing the necessary modules and types, you can easily incorporate existing crates from crates.io into your Rust projects, expanding the functionality and capabilities of your code.

Day -5 Concurrency and best practices

Generic Types

In Rust, generic types allow you to write code that can be reused with different types, providing flexibility and enabling code abstraction. Generics eliminate the need to write duplicate code for similar functionality but different types.

Defining Generic Types

To define a generic type in Rust, you use angle brackets (`<>`) followed by one or more generic parameters. These parameters represent the placeholder types that will be replaced with concrete types when the code is used.

```
struct Pair<T> {
    first: T,
    second: T,
}

fn main() {
    let pair_of_integers = Pair { first: 1, second: 2 };
    let pair_of_strings = Pair { first: "Hello", second: "World" };
}
```

In the above example, we define a `Pair` struct that has a single generic parameter `T`. The `T` can represent any type. We then create instances of `Pair` with different types: `pair_of_integers` with `i32` and `pair_of_strings` with `&str`.

Generic Functions

You can also define generic functions that operate on generic types:

```
fn print_pair<T>(pair: Pair<T>) {
    println!("First: {}, Second: {}", pair.first, pair.second);
}


fn main() {
    let pair_of_integers = Pair { first: 1, second: 2 };
    let pair_of_strings = Pair { first: "Hello", second: "World" };
```

```
    print_pair(pair_of_integers);
    print_pair(pair_of_strings);
}
```

In this example, we define a generic function `print_pair` that takes a `Pair<T>` as an argument. The function can accept a `Pair` with any type `T`. We then call `print_pair` with both `pair_of_integers` and `pair_of_strings`.

Constraints and Trait Bounds

Rust allows you to specify constraints on generic types using trait bounds. Trait bounds specify that the generic type must implement certain traits.

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

fn print_and_return<T: Display>(value: T) -> T {
    println!("Value: {}", value);
    value
}

fn main() {
    let value = print_and_return(42);
    println!("Returned value: {}", value);
}
```

In this example, the generic function `print_and_return` has a trait bound `T: Display`, which means the generic type `T` must implement the `Display` trait. The function prints the value and returns it. We call `print_and_return` with an integer value and then print the returned value.

Generic types and functions provide code reusability and enable you to write flexible and abstract code that works with various types. They are a powerful feature of Rust that promotes code efficiency and expressiveness.

Input output

Certainly! Input and output (I/O) operations in Rust are primarily handled through the `std::io` module, which provides functionalities to read input from the user and write

output to the console or files. Here's a detailed note about I/O in Rust with code examples:

Standard Input (stdin):

To read input from the user through the console, you can use the `stdin` function from the `std::io` module, along with the `BufRead` trait:

```
use std::io::{self, BufRead};

fn main() {
    let stdin = io::stdin();
    let mut buffer = String::new();

    println!("Enter your name:");
    stdin.lock().read_line(&mut buffer).expect("Failed to read line");

    println!("Hello, {}!", buffer);
}
```

In this example, we create a new `stdin` instance and a mutable `buffer` to store the user's input. The `read_line` method reads the user's input and appends it to the `buffer`. Finally, we print a greeting message with the user's name.

Standard Output (stdout):

To write output to the console, you can use the `println!` macro, which is similar to the `println` function in other programming languages:

```
fn main() {
    let name = "Alice";
    let age = 25;

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

In this example, we use the `println!` macro to print the values of `name` and `age` to the console. The `{}` placeholders are replaced with the corresponding values when printing.

File I/O

Rust provides several types and functions for file I/O operations. Here's an example of reading and writing files:

```
use std::io::{self, Read, Write};
use std::fs::File;

fn main() -> io::Result<()> {
    let mut file = File::open("input.txt")?;
    let mut contents = String::new();

    file.read_to_string(&mut contents)?;
    println!("File contents: {}", contents);

    let mut output_file = File::create("output.txt")?;
    output_file.write_all(contents.as_bytes())?;

    Ok(())
}
```

In this example, we open the file `input.txt` using `File::open`, read its contents into a `String`, and then print the contents. We then create a new file `output.txt` using `File::create` and write the contents to it using `write_all`.

The `io::Result` type is used to handle potential errors that may occur during I/O operations. The `?` operator is used to propagate the error to the calling function.

Package Manager

In Rust, the package manager is called Cargo. Cargo is a command-line tool that helps manage Rust projects, including dependency management, building, testing, and running projects. Here's a detailed note about Cargo in Rust:

Initializing a New Rust Project:

To start a new Rust project using Cargo, you can use the `cargo new` command followed by the project name:

```
$ cargo new my_project
```

This command creates a new directory called `my_project` with the basic project structure and files necessary to get started.

Managing Dependencies:

Cargo makes it easy to manage dependencies for your Rust projects. You define dependencies in the `Cargo.toml` file, which acts as the manifest for your project. Here's an example of a `Cargo.toml` file with a dependency:

```
[dependencies]
rand = "0.8.4"
```

In this example, we specify a dependency on the `rand` crate with version 0.8.4. Cargo will automatically download and manage the dependency for you when you build the project.

Building and Running a Project:

To build a Rust project using Cargo, you can use the `cargo build` command:

```
$ cargo build
```

This command compiles the project and its dependencies, producing an executable binary. The resulting binary is placed in the `target/debug` directory by default.

To run the project, you can use the `cargo run` command:

```
$ cargo run
```

This command automatically builds the project if necessary and then runs the resulting binary.

Testing a Project:

Cargo provides built-in support for testing Rust projects. You can place test functions within the `tests` directory or use the `#[cfg(test)]` attribute to mark test

functions within your project's source files. To run tests, you can use the `cargo test` command:

```
$ cargo test
```

This command runs all the tests in the project and reports the results.

Publishing a Crate:

If you've developed a Rust library and want to publish it as a crate for others to use, Cargo provides a simple way to publish to the crates.io registry. You need to create an account on crates.io and run the `cargo publish` command after completing the necessary crate metadata in the `Cargo.toml` file.

```
$ cargo publish
```

Cargo will package your crate, upload it to the crates.io registry, and make it available for others to use.

Cargo.lock File:

When you build or update your project's dependencies, Cargo generates a `Cargo.lock` file. This file locks the versions of your project's dependencies, ensuring that subsequent builds use the same versions. You should commit the `Cargo.lock` file to your version control system to ensure reproducible builds.

Cargo is a powerful and convenient package manager in Rust. It simplifies the management of dependencies, building, testing, and running projects. With Cargo, you can easily create, develop, and distribute your Rust projects with confidence.

Iterators and closures

Iterators and closures are important concepts in Rust that allow you to work with collections and perform functional programming-style operations. Here's a detailed note about iterators and closures in Rust:

Iterators in Rust

Iterators in Rust provide a way to iterate over a collection or a sequence of elements. They allow you to perform various operations on the elements, such as mapping,

filtering, and reducing. Iterators are lazy, which means they only compute values as needed.

Creating an Iterator:

You can create an iterator in Rust using the `iter` method or the `into_iter` method. Here's an example of creating an iterator from a vector:

```
let numbers = vec![1, 2, 3, 4, 5];
let mut iter = numbers.iter(); // Creates an iterator over the vector
```

Using Iterator Methods:

Once you have an iterator, you can use various iterator methods to perform operations on the elements. Some commonly used methods include `map`, `filter`, `fold`, `collect`, and more. Here's an example of using the `map` and `filter` methods:

```
let squares: Vec<_> = numbers.iter().map(|x| x * x).collect(); // squares = [1, 4, 9, 16, 25]

let even_numbers: Vec<_> = numbers.iter().filter(|x| x % 2 == 0).collect(); // even_numbers = [2, 4]
```

Closures in Rust:

Closures, also known as anonymous functions, are a way to define functions on the fly without giving them a separate name. They capture variables from their surrounding environment and can be used as arguments or returned values.

Creating a Closure:

Closures in Rust are defined using the `|...|` syntax. Here's an example of a closure that adds two numbers:

```
let add_numbers = |x, y| x + y;
let sum = add_numbers(5, 10); // sum = 15
```

Using Closures with Iterators:

Closures are often used with iterators to define custom operations on elements. Here's an example of using a closure with the `map` method:

```
let numbers = vec![1, 2, 3, 4, 5];
let doubled_numbers: Vec<_> = numbers.iter().map(|x| x * 2).collect(); //
doubled_numbers = [2, 4, 6, 8, 10]
```

Closures can capture variables from their surrounding scope. For example, you can capture variables from the outer scope inside a closure and use them within the closure. This feature is called variable capture.

```
let base_number = 10;
let add_to_base = |x| x + base_number;
let result = add_to_base(5); // result = 15
```

Closures provide flexibility and expressiveness when working with collections and performing functional programming operations in Rust. They allow you to write concise and powerful code.

It's worth noting that closures in Rust have different traits (`Fn`, `FnMut`, and `FnOnce`) depending on how they capture variables and how they are used. The traits determine the level of access to captured variables and how the closure can be called.

By combining iterators and closures, you can perform a wide range of operations on collections, transforming and filtering data in a concise and efficient manner.

Smart Pointers

Smart pointers are an important concept in Rust that enable more flexible and controlled memory management. They provide additional capabilities beyond regular references (`&`) by encapsulating a value and providing additional metadata and behavior. Here's a detailed note about smart pointers in Rust:

Smart pointers in Rust are types that behave like regular pointers but have additional capabilities and guarantees. They are implemented as structs that wrap a value and provide additional functionality through their associated methods.

There are several smart pointers available in the Rust standard library, including `Box`, `Rc`, `Arc`, `Cell`, `RefCell`, and `Mutex`. Each smart pointer has its own characteristics and use cases.

1. Box:

`Box<T>` is the simplest and most commonly used smart pointer in Rust. It allows you to allocate values on the heap rather than the stack, enabling dynamic memory allocation and providing ownership semantics. It's often used to create recursive data structures or when you need to transfer ownership of a value to another scope.

Example:

```
let x = Box::new(5); // Allocates an integer on the heap
println!("Value: {}", *x); // Dereference the Box to access the value
```

2. Rc:

`Rc<T>` (Reference Counted) is a smart pointer that enables multiple ownership of the same data. It keeps track of the number of references to a value and deallocates the memory when the last reference goes out of scope. It's useful when you need to share data between multiple parts of the program that only need read access.

Example:

```
use std::rc::Rc;

let shared_data = Rc::new(42); // Creates a reference-counted smart pointer
let clone1 = Rc::clone(&shared_data); // Creates a new reference to the same data
let clone2 = Rc::clone(&shared_data); // Creates another reference

println!("Reference count: {}", Rc::strong_count(&shared_data)); // Prints the
reference count
```

3. Arc:

`Arc<T>` (Atomic Reference Counted) is similar to `Rc<T>` but provides thread-safe reference counting. It can be safely shared across multiple threads and ensures that the reference count is correctly updated. It's used in concurrent or multi-threaded programs where data needs to be shared across threads.

Example:

```
use std::sync::Arc;
use std::thread;

let shared_data = Arc::new(42);

let thread1 = {
    let data = Arc::clone(&shared_data);
    thread::spawn(move || {
        println!("Thread 1: {}", *data);
    })
};

let thread2 = {
    let data = Arc::clone(&shared_data);
    thread::spawn(move || {
        println!("Thread 2: {}", *data);
    })
};

thread1.join().unwrap();
thread2.join().unwrap();
```

Certainly! Here are explanations of other smart pointers in Rust:

4. Cell:

`Cell<T>` is a smart pointer that provides interior mutability, allowing you to mutate values even when you only have an immutable reference to them. It's commonly used when you need to mutate data behind shared references, such as in multi-threaded environments.

Example:

```
use std::cell::Cell;

let x = Cell::new(5);
let y = &x;

x.set(10); // Mutates the value inside Cell

println!("Value: {}", y.get()); // Accesses the mutated value
```

5. RefCell:

`RefCell<T>` is similar to `Cell<T>` but provides dynamic borrowing rules at runtime. It allows you to have multiple mutable references to a value, enforcing borrowing rules at runtime rather than compile-time. It's commonly used when you need interior mutability with more flexible borrowing.

Example:

```
use std::cell::RefCell;

let x = RefCell::new(5);

let y = x.borrow_mut();
// Mutable borrowing of the value inside RefCell

println!("Value: {}", *y); // Accesses the borrowed value
```

6. Mutex:

`Mutex<T>` is a smart pointer that provides mutual exclusion, allowing multiple threads to safely access shared data. It enforces exclusive access through locking and unlocking mechanisms. It's commonly used in concurrent or multi-threaded programs to synchronize access to shared resources.

Example:

```
use std::sync::Mutex;
use std::thread;

let shared_data = Mutex::new(5);

let thread1 = {
    let data = shared_data.lock().unwrap();
    println!("Thread 1: {}", *data);
};

let thread2 = {
    let data = shared_data.lock().unwrap();
    println!("Thread 2: {}", *data);
};

thread::spawn(move || thread1).join().unwrap();
thread::spawn(move || thread2).join().unwrap();
```

These are additional smart pointers in Rust that provide specific functionality to handle different scenarios. Each smart pointer has its own characteristics and usage patterns, enabling you to choose the appropriate one for your specific needs. By leveraging smart pointers, you can achieve safer and more flexible memory management in Rust.

Concurrency

Certainly! Concurrency in Rust refers to the ability to execute multiple tasks or computations concurrently. Rust provides several abstractions and tools to handle concurrency effectively. Here's a detailed note about concurrency in Rust:

1. Threads:

Rust supports threading through the `std::thread` module. You can create and manage threads to execute different tasks concurrently. Each thread runs in parallel and can communicate with other threads through shared memory or message passing.

Example:

```
use std::thread;

fn main() {
    let thread1 = thread::spawn(|| {
        println!("Hello from Thread 1!");
    });

    let thread2 = thread::spawn(|| {
        println!("Hello from Thread 2!");
    });

    thread1.join().expect("Thread 1 panicked");
    thread2.join().expect("Thread 2 panicked");
}
```

2. Message Passing:

Rust provides message passing for inter-thread communication using channels. Channels allow threads to send and receive messages or data. The `std::sync::mpsc` module provides a multi-producer, single-consumer channel implementation.

Example:

```
use std::sync::mpsc;
use std::thread;

fn main() {
    let (tx, rx) = mpsc::channel();

    let thread1 = thread::spawn(move || {
        let message = String::from("Hello from Thread 1!");
        tx.send(message).expect("Sending message failed.");
    });



    let thread2 = thread::spawn(move || {
```

```
    let received = rx.recv().expect("Receiving message failed.");
    println!("{}", received);
  });

  thread1.join().expect("Thread 1 panicked");
  thread2.join().expect("Thread 2 panicked");
}
```

3. Atomic Operations:

Rust provides atomic types from the `std::sync::atomic` module, which allow for shared mutable state between threads without the need for locks. Atomic types can be modified atomically without data races.

Example:

```
use std::sync::atomic::{AtomicUsize, Ordering};
use std::thread;

fn main() {
  let counter = AtomicUsize::new(0);

  let thread1 = thread::spawn(move || {
    counter.fetch_add(1, Ordering::SeqCst);
  });

  let thread2 = thread::spawn(move || {
    counter.fetch_add(1, Ordering::SeqCst);
  });

  thread1.join().expect("Thread 1 panicked");
  thread2.join().expect("Thread 2 panicked");

  let final_count = counter.load(Ordering::SeqCst);
  println!("Counter: {}", final_count);
}
```

These are some examples of concurrency in Rust using threads, message passing, and atomic operations. Rust's strong ownership and borrowing model, along with these concurrency abstractions, help ensure thread safety and prevent data races.