Rust Advance

By Code Eater

Contents

Target

2K Likes

200 comments

Things to know

Notes aur Code
Saare notes aur github link PPT ka part hain. Aapko kuch extra topics bhi milenge — unhe bhi freely explore kar sakte ho.

Programs aur Screenshots
Saare programs PPT ke screenshots mein diye gaye hain. Screenshot par click karke aap program dekh aur run kar sakte ho. Agar kisi screenshot mein code clearly nahi dikh raha ho, toh aap AI tools jaise ChatGPT ka use karke screenshot upload karo aur wahi code mang sakte ho.

Video Language aur Navigation
Video Hindi mein record ki gayi hai. Agar aap English mein sun rahe ho, toh settings se Hindi par switch kar sakte ho.Saara content slides ke andar hyperlink kiya gaya hai, jisse aap easily access kar sakte ho.

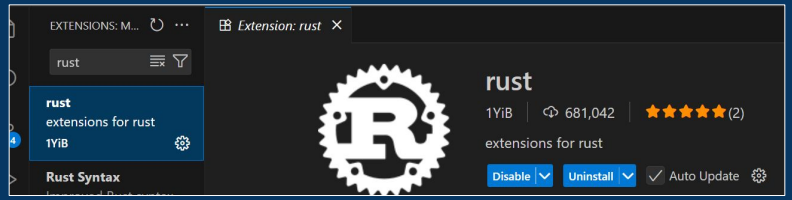
Things to know

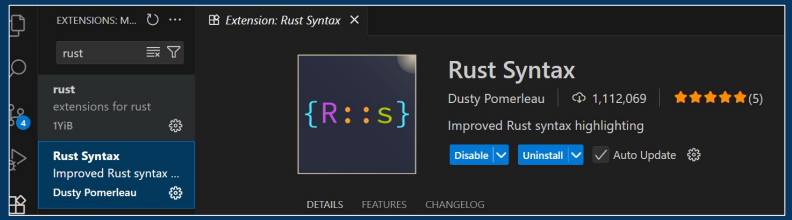
Notes and Topics
All the notes and github links are part of the PPT. You will also find some additional topics—feel free to explore them as well.

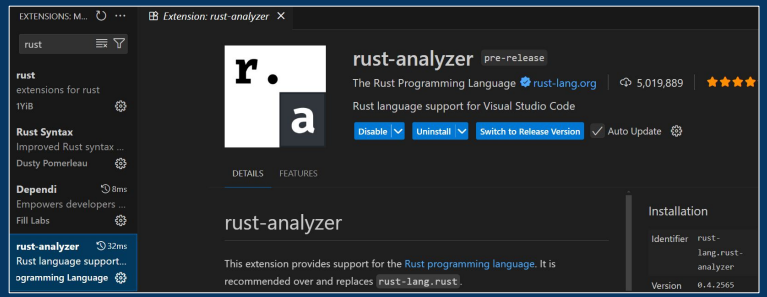
Programs and Screenshots
All the programs are included in the screenshots within the PPT. You can click on a screenshot to access and run the program. If, in some cases, the program is not visible in the screenshot, you can use AI tools like ChatGPT by uploading the screenshot and asking for the corresponding code.

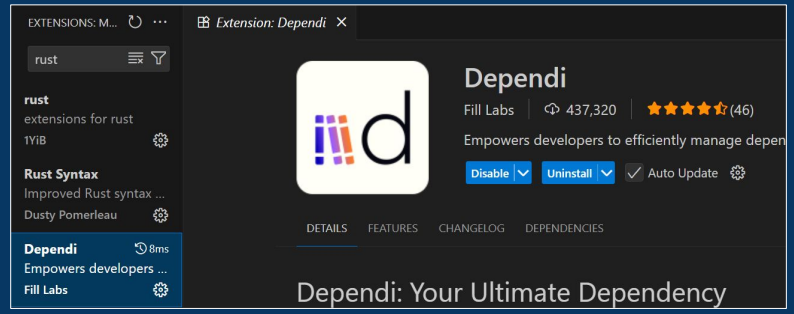
Video Language and Navigation
The video is recorded in Hindi. If you're currently listening in English, you can switch back to Hindi.
All content is hyperlinked within the slides for easy access.

Extensions I am using









Algebraic Data Types

Algebraic Data Types

Struct

Enum

Struct

```rust
#[derive(Debug)]
struct User {
    active: bool,
    username: String,
    email: String,
    sign_in_count: u64,
}

fn main() {
    let mut user1 = User {
        active: true,
        username: String::from("someusername123"),
        email: String::from("someone@example.com"),
        sign_in_count: 1,
    };

    //assigning a different value
    user1.email = String::from("anotheremail@example.com");

    println!("User: {:?}", user1); // Prints in single-line debug format
    println!("User (pretty): {:#?}", user1); // Prints in pretty (multi-line) debug format

    let user2 = User {
        email: String::from("another@example.com"),
        ..user1
    };

    println!("User: {:?}", user2); // Prints in single-line debug format
    println!("User (pretty): {:#?}", user2); // Prints in pretty (multi-line) debug format


}
```

A **struct** in Rust is a user-defined data type that groups together related fields under one name, allowing you to create custom, structured data types. Fields within a struct can have different data types, and you can access them using dot notation.

Ownership - Transfer

```rust
#[derive(Debug)]
1 implementation
struct Student {
    name: String,
    roll: u32,
    pass: bool,
}

▶ Run | ⚙ Debug
fn main() {
    let student1: Student = Student {
        name: String::from("Kshitij"),
        roll: 15,
        pass: true,
    };

    name_change(studentx: student1); // Ownership transferred; cannot use student1 after this
}

fn name_change(mut studentx: Student) {
    studentx.name = String::from("Raj");
    println!("{:?}", studentx); // Print the modified struct
}
```

Ownership - Transfer and Back

```rust
#[derive(Debug)]
1 implementation
struct Student {
    name: String,
    roll: u32,
    pass: bool,
}

▶ Run | ⚙ Debug
fn main() {
    let student1: Student = Student {
        name: String::from("Kshitij"),
        roll: 15,
        pass: true,
    };

    let student1: Student = name_change(studentx: student1); // Ownership moved and returned
    println!("{:?}", student1); // Print the modified struct
}

fn name_change(mut studentx: Student) -> Student {
    studentx.name = String::from("Raj");
    studentx // Return ownership back
}
```

Ownership in Struct

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

fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };

    println!(
        "The area of the rectangle is {} square pixels.",
        area(&rect1)
    );
}

fn area(rectangle: &Rectangle) -> u32 {
    rectangle.width * rectangle.height
}
```

Mutable Reference

```rust
#[derive(Debug)]
1 implementation
struct Student {
    name: String,
    roll: u32,
    pass: bool,
}

▶ Run | ⚙ Debug
fn main() {
    let mut student1: Student = Student {
        name: String::from("Kshitij"),
        roll: 15,
        pass: true,
    };

    name_change(studentx: &mut student1);
    println!("{:?}", student1);
}

fn name_change(studentx: &mut Student) {
    studentx.name = String::from("Raj");
    println!("{:?}", studentx);
}
```

Method Syntax

Methods are similar to functions: we declare them with the fn keyword and a name, they can have parameters and a return value, and they contain some code that’s run when the method is called from somewhere else. Unlike functions, methods are defined within the context of a struct or an enum, and their first parameter is always self, which represents the instance of the struct the method is being called on.

Key Features of Method Syntax

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

impl Rectangle {
    // A method that calculates the area of the rectangle
    fn area(&self) -> u32 {
        self.width * self.height
    }
}

fn main() {
    let rect = Rectangle {
        width: 30,
        height: 50,
    };

    println!("The area of the rectangle is {} square pixels.", rect.area());
}
```

Defined within an `impl` block:

- Methods are defined in an `impl` block for a specific type (e.g., struct, enum).

First Parameter is `self`:

- Methods always take `self`, `&self`, or `&mut self` as the first parameter. This indicates that the method operates on the instance of the type:
  - `self`: Takes ownership of the instance.
  - `&self`: Borrows the instance immutably.
  - `&mut self`: Borrows the instance mutably.

Called using Dot Notation:

- Methods are called using the dot syntax (`instance.method_name()`), which makes them feel intuitive and object-oriented.

Associated Functions

Associated functions in Rust are functions defined within an impl block for a particular type (like a struct, enum, or trait). These functions are associated with the type they are defined for, but they don't require an instance of the type to be called. They are invoked using the :: syntax and are often used for utility functions, constructors, or other operations related to the type as a whole rather than to a specific instance.

Key Characteristics of Associated Functions

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

impl Rectangle {
    // Constructor to create a square
    fn square(size: u32) -> Self {
        Self {
            width: size,
            height: size,
        }
    }
}

let sq = Rectangle::square(3);
println!("Square: width = {}, height = {}", sq.width, sq.height);
```

No `self` parameter:

- Unlike methods, associated functions do not take `self`, `&self`, or `&mut self` as their first parameter because they operate on the type, not on an instance of the type.

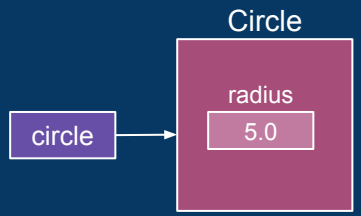
Namespaced by the Type:

- You call them using the type's name followed by `::`, such as `TypeName::function_name()`.

Common Use Cases:

- **Constructors**: Functions like new or custom builders that return new instances of the type.
- **Utility Functions**: Functions that perform calculations or operations relevant to the type but don't need an instance (e.g., `String::from`).

To learn about std crate - Click Here




```rust
struct Circle {
    radius: f64,
}

impl Circle {
    // Associated function
    pub fn new(radius: f64) -> Circle {
        Circle { radius }
    }

    // Instance method
    pub fn area(&self) -> f64 {
        std::f64::consts::PI * self.radius * self.radius
    }
}

fn main() {
    // Call the associated function `new` to create a `Circle` instance
    let circle = Circle::new(5.0);

    // Call the instance method `area` on the `Circle` instance
    println!("The area of the circle is {}", circle.area());
}
```

Methods vs Associated Functions

Feature	Methods	Associated Functions
Definition	Within `impl`, first param `self`	Within `impl`, no `self` param
Access to Instance	Yes	No
Syntax for Calling	instance.method()	TypeName::function()

Types of Enum

Simple Enum

Optional Enum

Result Enum

Types of Enum

Simple Enum

Optional Enum

Result Enum

Enum

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

fn main() {
    let c: Color = Color::Red;

    match c {
        Color::Red => println!("Red"),
        Color::Green => println!("Green"),
        Color::Blue => println!("Blue"),
    }
}
```

Here Red,Green,Blue are called as Variants.
The size of an enum type is determined by the size of its largest variant. Here all are of 1 byte so enum size is 1 byte.

An enum is a data type that has a closed set of allowed values. So for example here, a traffic light enum type can let you have red,yellow,green but nothing else.

We could use integers to achieve the same effect like an integer that could be zero to represent the red light, one to represent yellow light,two to represent green light, but enums are more self-describing.

To define an enum type:

- Specify the enum type,starting with a capital letter.
- Specify allowed values(variants), also starting with capitals.

Enum With Variant

```rust
enum School {
    Number(i32),
    Name(String),
    Unknown,
}

fn school_data() {
    let s: School = School::Unknown;

    match s {
        School::Number(n) => println!("School roll number is {}", n),
        School::Name(s) => println!("School name is {}", s),
        School::Unknown => println!("School location is unknown"),
    }

    let size = std::mem::size_of::<School>();
    println!("The size of enum School is {} bytes", size);
}

fn main() {
    school_data();
}
```


The size of the School enum is 24 bytes due to the largest variant Name(String), which contains a String type.

The String type is a dynamically sized type because it holds a pointer to the actual string data on the heap, along with metadata such as the length of the string.

On a 64-bit architecture, the pointer size is typically 8 bytes.
Additional metadata for the string (typically 16 bytes).
So, the total size for this variant is 8 (pointer size) + 16 (metadata) = 24 bytes.

Enum with impl

```rust
enum Shape {
    Circle(f64),     // Variant with radius
    Rectangle(f64, f64), // Variant with width and height
    Triangle(f64, f64, f64), // Variant with side lengths
}

impl Shape {
    fn area(&self) -> f64 {
        match *self {
            Shape::Circle(radius) => std::f64::consts::PI * radius * radius,
            Shape::Rectangle(width, height) => width * height,
            Shape::Triangle(a, b, c) => {
                let s = (a + b + c) / 2.0;
                (s * (s - a) * (s - b) * (s - c)).sqrt()
            }
        }
    }
}

fn main() {
    let circle = Shape::Circle(3.0);
    let rectangle = Shape::Rectangle(2.0, 4.0);
    let triangle = Shape::Triangle(3.0, 4.0, 5.0);

    println!("Area of the circle: {}", circle.area());
    println!("Area of the rectangle: {}", rectangle.area());
    println!("Area of the triangle: {}", triangle.area());
}
```

std::f64::consts::PI:
This part accesses the constant value of π (pi) defined in the standard library's f64 module. f64 specifies that it's a 64-bit floating-point number (double precision floating-point number).

When you call circle.area(), you're invoking the area() method on the circle instance, which is of type Shape. In Rust, methods take a special first parameter called self, which represents the instance the method is being called on.

So, in the context of circle.area(), self inside the area() method refers to circle, the instance of the Shape enum representing a circle.

Enum

```rust
enum TrafficLight {
    Red,
    Yellow,
    Green,
}

impl TrafficLight {
    fn display_color(&self) {
        match *self {
            TrafficLight::Red => println!("The traffic light is red"),
            TrafficLight::Yellow => println!("The traffic light is yellow"),
            TrafficLight::Green => println!("The traffic light is green"),
        }
    }
}

fn main() {
    let red_light = TrafficLight::Red;
    let yellow_light = TrafficLight::Yellow;
    let green_light = TrafficLight::Green;

    red_light.display_color();
    yellow_light.display_color();
    green_light.display_color();
}
```

&self: This is a reference to the instance of the type on which the method is being called. It's commonly used in method definitions to indicate that the method does not take ownership of the instance. Methods that take &self can be called on immutable references to the type.

*self: This is the dereferenced value of self. When you see *self, it means that you're accessing the value that self refers to. This is typically done when you have a reference to an object and you want to access the object itself.

Unknown

```rust
enum TrafficLight {
    Red,
    Green,
    Yellow,
    Unknown, // Represents an unclear or invalid state
}

fn get_traffic_light_status(input: &str) -> TrafficLight {
    match input {
        "Red" => TrafficLight::Red,
        "Green" => TrafficLight::Green,
        "Yellow" => TrafficLight::Yellow,
        _ => TrafficLight::Unknown, // Handle unclear input
    }
}

fn main() {
    let status = get_traffic_light_status("Blue");

    match status {
        TrafficLight::Red => println!("Stop!"),
        TrafficLight::Green => println!("Go!"),
        TrafficLight::Yellow => println!("Slow down!"),
        TrafficLight::Unknown => println!("Unclear traffic light status."),
    }
}
```

Unknown can act as a default or fallback state, representing cases where none of the other variants (Number or Name) apply. It's often used in contexts where the state is unclear or not explicitly defined.

Unknown Example

```rust
enum InputType {
    Number(i32),
    Name(String),
    Unknown,
}

fn parse_input(input: &str) -> InputType {
    if let Ok(number) = input.parse::<i32>() {
        InputType::Number(number)
    } else if !input.trim().is_empty() {
        InputType::Name(input.to_string())
    } else {
        InputType::Unknown // For unrecognized or empty input
    }
}

fn main() {
    let inputs = vec!["42", "Alice", "   ", "invalid123"];

    for input in inputs {
        match parse_input(input) {
            InputType::Number(num) => println!("Parsed a number: {}", num),
            InputType::Name(name) => println!("Parsed a name: {}", name),
            InputType::Unknown => println!("Input is unclear or invalid"),
        }
    }
```

Suppose you have a program that processes different types of input, such as a number, a name, or an unrecognized/invalid value. You can use an enum with an Unknown variant to handle cases where the input doesn't match the expected types.

Optional Enum

By default, enums in Rust are not optional in the sense that once you define an enum, when you use it, you must specify which variant it holds.

However, if you want to represent the concept of something being optional using enums, you can achieve that by including an additional variant that represents the absence of a value.

For example, you could define an enum called OptionalValue that represents either a value or the absence of a value

In this example:
Some(T) represents the presence of a value of type T.
None represents the absence of a value.

```
enum OptionalValue<T> {
    Some(T),
    None,
}
```

Why:Optional Enum

Optional enums in Rust, such as the Option type, were indeed introduced as a solution to the problem of null pointers, which are a common source of bugs and crashes in languages that allow them, such as C and C++.

In Rust, null pointers are replaced by the Option<T> enum, which can either hold a value of type T or represent the absence of a value. This design choice enforces developers to handle the possibility of absence in a type-safe manner, thereby preventing many null pointer errors at compile time.

Optional Enum Example - 1

```rust
enum OptionalValue<T> {
    Some(T),
    None,
}

fn main() {
    let some_value: OptionalValue<i32> = OptionalValue::Some(42);
    let no_value: OptionalValue<i32> = OptionalValue::None;

    match some_value {
        OptionalValue::Some(value) => println!("Got a value: {}", value),
        OptionalValue::None => println!("Got no value"),
    }

    match no_value {
        OptionalValue::Some(value) => println!("Got a value: {}", value),
        OptionalValue::None => println!("Got no value"),
    }
}
```

Optional Enum Example - 2

```rust
fn main() {
    // Test the function with different user IDs
    let user_id_1 = 1;
    let user_id_2 = 2;

    // Call the function and handle the result
    match get_user_phone(user_id_1) {
        Some(phone) => println!("User {}'s phone: {}", user_id_1, phone),
        None => println!("No phone number available for user {}.", user_id_1),
    }

    match get_user_phone(user_id_2) {
        Some(phone) => println!("User {}'s phone: {}", user_id_2, phone),
        None => println!("No phone number available for user {}.", user_id_2),
    }
}

// Function to simulate retrieving a user's phone number from a database
fn get_user_phone(user_id: u32) -> Option<String> {
    // Simulate a database lookup
    if user_id == 1 {
        Some(String::from("+123456789"))
    } else {
        None
    }
}
```

Result Enum

In Rust, the Result enum is a type that represents the result of a computation that can either succeed (Ok) and return a value or fail (Err) and return an error. It's commonly used for functions that may fail for various reasons.

Here, T represents the type of the value returned on success, and E represents the type of the error returned on failure.

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

Result Enum

```rust
// A function that divides two numbers and returns either the result or an error
fn divide(x: i32, y: i32) -> Result<i32, String> {
    if y == 0 {
        // If the divisor is zero, return an error
        Err(String::from("Division by zero"))
    } else {
        // Otherwise, return the result of division
        Ok(x / y)
    }
}

fn main() {
    // Try to divide two numbers and handle the result
    match divide(10, 2) {
        Ok(result) => {
            println!("Result of division: {}", result);
        }
        Err(error) => {
            eprintln!("Error: {}", error);
        }
    }
}
```


The Result enum and its variants have been brought into scope by the prelude, so we don’t need to specify Result:: before the Ok and Err variants in the match arms.

Generic Type

Generic Type

```rust
use std::fmt::Debug;

fn print_type<T: Debug>(var: T) {
    println!("Hi, {:?}", var);
}

▶ Run | ⚙ Debug
fn main() {
    print_type(var: 5);
    print_type(var: true);
    print_type(var: "hi");
    print_type(var: 3.14);
}
```

In Rust, generic types are types or functions that can work with multiple types, while maintaining type safety at compile time.y using generics, you can write more flexible and reusable code.

You can use any valid identifier to represent a generic type in Rust, not just T. While T is commonly used as a convention for "Type," especially in simple examples, you can use other names that might be more descriptive or appropriate for your context.

Generic Type

This is the only non-repetitive code otherwise else everything is repetitive.

```
largest_i32(list: &[i32]) -> &i32 {
    let mut largest = &list[0];
    for item in list.iter() {
        if item > largest {
            largest = item;
        }
    }
    largest
}

main() {
    let number_list = vec![34, 50, 25, 100, 65];
    let result = largest_i32(&number_list);
    println!("The largest number is {}", result);
}
```

```
fn largest_f64(list: &[f64]) -> &f64 {
    let mut largest = &list[0];
    for item in list.iter() {
        if item > largest {
            largest = item;
        }
    }
    largest
}

fn main() {
    let float_list = vec![1.1, 2.9, 0.4, 3.5, 2.8];
    let result = largest_f64(&float_list);
    println!("The largest float is {}", result);
}
```

```
fn largest_char(list: &[char]) -> &char {
    let mut largest = &list[0];
    for item in list.iter() {
        if item > largest {
            largest = item;
        }
    }
    largest
}

fn main() {
    let char_list = vec!['y', 'm', 'a', 'q'];
    let result = largest_char(&char_list);
    println!("The largest char is {}", result);
}
```

W/O Generic Types

Traits are often used with generics to impose constraints on what types can be used with the generic code. This ensures that the generic types have the necessary capabilities (such as implementing certain methods or operators).

```rust
use std::cmp::PartialOrd;

// Define the generic function 'largest' with the 'PartialOrd' constraint
fn largest<T: PartialOrd>(list: &[T]) -> &T {
    let mut largest = &list[0];
    for item in list.iter() {
        if item > largest {
            largest = item;
        }
    }
    largest
}

fn main() {
    // Test with a list of integers
    let number_list = vec![34, 50, 25, 100, 65];
    let result = largest(&number_list);
    println!("The largest number is {}", result);

    // Test with a list of floating point numbers
    let float_list = vec![1.1, 2.9, 0.4, 3.5, 2.8];
    let result = largest(&float_list);
    println!("The largest float is {}", result);

    // Test with a list of characters
    let char_list = vec!['y', 'm', 'a', 'q'];
    let result = largest(&char_list);
    println!("The largest char is {}", result);
}
```

fn largest<T: PartialOrd>(list: &[T]) -> &T: Defines a function named largest that takes a slice of items of type T. The type T must implement the PartialOrd trait.

T: This is the generic type parameter. It allows the function to operate on slices of any type, rather than being limited to a specific type like i32 or f64.

PartialOrd: This is a trait bound. It constrains T to types that implement the PartialOrd trait, which allows for comparison operations (<, >, <=, >=). Without this constraint, the compiler wouldn't know if the type T supports comparison.

Traits

Traits

Object in OOPs

Traits

Enum/Structs

Traits

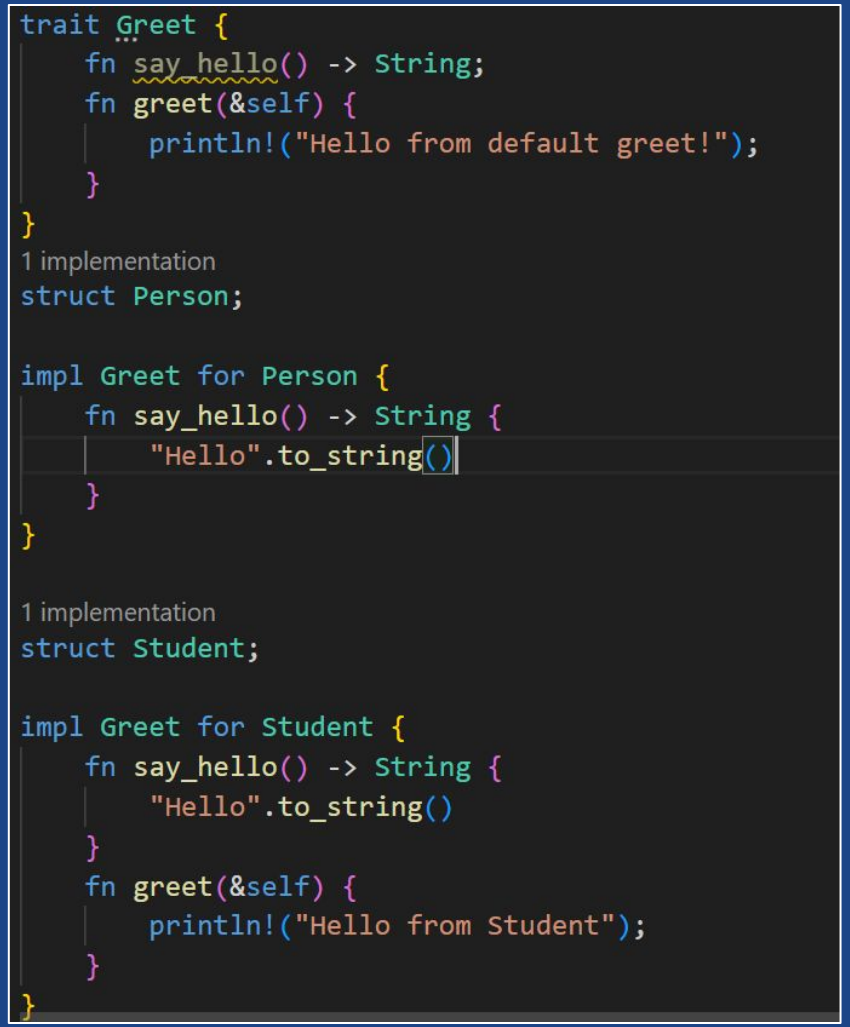
```rust
struct Car {
    name: String,
}

struct SuperCar {
    name: String,
}

trait Speed {
    fn car_speed(&self);
}

impl Speed for Car {
    fn car_speed(&self) {
        println!("{}'s maximum speed is 102 mph", self.name);
    }
}

impl Speed for SuperCar {
    fn car_speed(&self) {
        println!("{}'s maximum speed is 249 mph", self.name);
    }
}

fn main() {
    let scorpio = Car {
        name: String::from("Scorpio"),
    };
    scorpio.car_speed();

    let ferrari = SuperCar {
        name: String::from("Ferrari"),
    };
    ferrari.car_speed();
}
```

Traits in Rust are a way to define shared behavior across different types.

You can think of them as interfaces in other languages. Traits allow you to specify methods that types must implement, providing a way to achieve polymorphism.

```rust
struct Car {
    name: String,
}

trait Speed {
    fn car_speed(&self) {
        println!("Maximum speed is unknown");
    }
}

impl Speed for Car {
    // The implementation is commented out, so it will use the default implementation fro the trait
    // fn car_speed(&self) {
    //     println!("{}'s maximum speed is 102 mph", self.name);
    // }
}

fn main() {
    let scorpio = Car {
        name: String::from("Scorpio"),
    };
    scorpio.car_speed();
}
```

Traits can define default behaviour for a method. So for the above program the output will be “Maximum speed is unknown”.


trait Greet {
fn say_hello() -> String;
fn greet(&self) {
println!("Hello from default greet!");
}
}
1 implementation
struct Person;
impl Greet for Person {
fn say_hello() -> String {
"Hello".to_string()
}
}
1 implementation
struct Student;
impl Greet for Student {
fn say_hello() -> String {
"Hello".to_string()
}
fn greet(&self) {
println!("Hello from Student");
}
}

Assignment - 1: Traits

Assignment - 1

You are building an online learning platform to manage different types of courses. You want to implement a system that can keep track of various courses and provide brief overviews of them.

Task

1. **Define a trait named `Course`**, which should have a method `get_overview` that returns a brief overview of the course as a string.

2. Create two structs:

    - **Workshop** with fields: `title`, `instructor`, and `duration` (in hours).
    - **Seminar** with fields: `title`, `speaker`, and `location`.

3. **Implement the `Course` trait for both `Workshop` and `Seminar`**, providing a suitable overview in each case.

4. Write a function `print_course_overview` that takes a generic parameter of any type that implements the `Course` trait and prints the overview.

The Debug Trait

```rust
#[derive(Debug)]
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let point = Point { x: 10, y: 20 };
    println!("{:?}", point);
}
```

The Debug trait in Rust is part of the standard library and provides a way to format a value for debugging purposes. When you derive the Debug trait, the Rust compiler automatically generates the necessary code for you, but it's helpful to understand how it works behind the scenes.

The Debug Trait

The Debug trait is defined in the standard library as follows:

```
pub trait Debug {
    fn fmt(&self, f: &mut Formatter<'_>) -> Result<(), Error>;
}
```

The fmt method is used to format the value. The Formatter type provides various methods for building the output string, and the method returns a Result to handle any potential errors during formatting.

Manually Implementing Debug

```rust
use std::fmt;

struct Point {
    x: i32,
    y: i32,
}

impl fmt::Debug for Point {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        f.debug_struct("Point")
            .field("x", &self.x)
            .field("y", &self.y)
            .finish()
    }
}

fn main() {
    let point = Point { x: 10, y: 20 };
    println!("{:?}", point);
}
```

When you derive Debug, the compiler generates similar code for you automatically. For example, with #[derive(Debug)] on the Point struct, the compiler would generate something very similar to the manual implementation above, handling all fields and variants.

Array & Vectors

Array

```rust
fn main() {
    let a1 = [10, 11, 12];
    println!("a1 length is {}, first element is {}", a1.len(), a1[0]);

    let mut a2 = [10, 11, 12];
    a2[0] = 9;
    println!("a2 length is {}, first element is {}", a2.len(), a2[0]);

    for elem in a2.iter() {
        println!("  {}", elem);
    }
}
```


Arrays in Rust are fixed-size collections of elements of the same data type. Unlike vectors, arrays have a fixed length determined at compile time, which means their size cannot change dynamically during runtime.

Array

```rust
fn main() {
    // Declare an array named a1 of type [i64; 5] without initializing it
    let a1: [i64; 5];
    // Initialize the array a1 with values [10, 11, 12, 13, 14]
    a1 = [10, 11, 12, 13, 14];

    // Print the debug representation of array a1
    println!("a1 is {:?}", a1);

    // Declare a mutable array named a2 initialized with [32, 32, 32, 32, 32]
    let mut a2 = [32; 5];
    // Modify the first element of a2 to 9
    a2[0] = 9;

    // Print the debug representation of array a2
    println!("a2 is {:?}", a2);
}
```

Vector

```rust
let mut v: Vec<i32> = Vec::new();
```

```rust
let mut v = Vec::<i32>::new();
```

Different ways of creating vectors.

```rust
let v1 = vec![1, 2, 3, 4, 5]; // Creates a vector with initial values
let v2: Vec<i32> = vec![0; 5]; // Creates a vector with 5 elements initialized to 0
```

In Rust, vectors are dynamic arrays that can grow or shrink in size at runtime.

Vector

```rust
let mut _v1: Vec<i32> = Vec::new();
let mut _v2 = Vec::<i32>::new();
let mut v3 = vec![100, 101, 102];

// Index into a vector safely, return an Option<T>
let opt = v3.get(10);
match opt {
    Some(value) => println!("Value: {}", value),
    None => println!("No value"),
}

// Push elements onto the vector
v3.push(1);
v3.push(5);

// Pop the last element from the vector
v3.pop();

// Insert an element at a specific index
v3.insert(0, 99);

// Iterate over the elements of the vector and print each one
for item in &v3 {
    println!("{}", item);
}
```

The &v3 creates an immutable reference to the vector v3. This reference allows the loop to iterate over the elements of v3 without transferring ownership of v3 itself. This means that v3 remains accessible and usable after the loop.

If you were to use v3 directly in the loop without the &, the vector v3 would be moved into the loop, and you wouldn't be able to use it afterward without re-declaring or cloning it. Using a reference (&) allows you to iterate over the elements of the vector without taking ownership of it.

Array Read and Write

```rust
fn main() {
    let mut arr: [&str; 2] = ["Hello", "World"];
    read_arr(&arr);
    write_arr(&mut arr);
    println!("{:?}", arr);
}

fn read_arr(arr: &[&str; 2]) {
    println!("{:?}", arr);
}

fn write_arr(arr: &mut [&str; 2]) {
    arr[0] = "New";
}
```

Vector Read and Write

```rust
fn main() {
    let mut vrr: Vec<&str> = vec!["Hello", "World"];
    read_arr(&vrr);
    write_arr(&mut vrr);
    println!("{:?}", vrr);
}

fn read_arr(vrr: &[&str]) {
    println!("{:?}", vrr);
}

fn write_arr(vrr: &mut Vec<&str>) {
    vrr.push("New");
}
```

```rust
fn main() {
    let mut vrr: Vec<&str> = vec!["Hello", "World"];

    // Read the original vector
    read_arr_one(&vrr);

    // Modify the vector
    write_arr_one(&mut vrr);

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

fn read_arr_one(vrr1: &[&str]) {
    // Print the slice of string references
    println!("Read Arr One: {:?}", vrr1);

    // Call another function to read the same slice
    read_arr_two(vrr1);
}

fn read_arr_two(vrr2: &[&str]) {
    // Print the slice of string references
    println!("Read Arr Two: {:?}", vrr2);
}

fn write_arr_one(vrr3: &mut Vec<&str>) {
    // Add a new string reference to the vector
    vrr3.push("New");

    // Call another function to modify the vector
    write_arr_two(vrr3);
}

fn write_arr_two(vrr4: &mut Vec<&str>) {
    // Add another new string reference to the vector
    vrr4.push("Few");
}
```

- `vrr1`: It is a reference to a slice of string slices (`&[&str]`). Specifically, it references the same data as `vrr` but does not have ownership. In `read_arr_one`, it's just a borrowed reference to `vrr`.

- `vrr2`: Similar to `vrr1`, it's also a reference to a slice of string slices (`&[&str]`). It's passed from `read_arr_one` to `read_arr_two`, essentially carrying the same reference to the data in `vrr`.

- `vrr3`: It's a mutable reference to the vector `vrr`. This means it can modify the vector `vrr`. In `write_arr_one`, `vrr3` is used to push the string slice `"New"` into the vector `vrr`, extending its length by one element.

- `vrr4`: Like `vrr3`, it's also a mutable reference to the vector `vrr`. In `write_arr_two`, `vrr4` is used to push the string slice `"Few"` into the vector `vrr`, further extending its length by one element.

So, at each point of printing, here's what they contain:

- `vrr1` and `vrr2`: They both contain references to the same data inside `vrr`, which includes the string slices `"Hello"` and `"World"`.

- `vrr3`: After `write_arr_one` function is executed, it contains references to the string slices `"Hello"`, `"World"`, and `"New"`.

- `vrr4`: After `write_arr_two` function is executed, it contains references to the string slices `"Hello"`, `"World"`, `"New"`, and `"Few"`.

Iterators

Iterators

Iterators in Rust are a powerful abstraction for working with sequences of data. They provide a way to process elements one at a time without requiring manual indexing or loops.

Iterators are lazy by design and are often used with the iterator adapters (e.g., `map`, `filter`, `collect`) for functional-style transformations.

Refer this - Click Here

Why Iterators?

In Rust, you cannot directly loop through an array using for without explicitly converting it into an iterator because Rust enforces strict ownership and borrowing rules. Rust requires an **iterator** to iterate over collections like arrays or vectors. Unlike some languages (e.g., Python, JavaScript)

```rust
let arr = [1, 2, 3];

for item in arr {
    println!("{}", item);
}
```

```rust
let mut iter: IntoIter<i32, 3> = arr.into_iter(); //converting array into iterator
while let Some(item: i32) = iter.next() {
    println!("{}", item);
}
```

Iterators

```
let a = [1, 2, 3];

let mut iter = a.iter();

// A call to next() returns the next value...
assert_eq!(Some(&1), iter.next());
assert_eq!(Some(&2), iter.next());
assert_eq!(Some(&3), iter.next());

// ... and then None once it's over.
assert_eq!(None, iter.next());

// More calls may or may not return `None`. Here, they always will.
assert_eq!(None, iter.next());
assert_eq!(None, iter.next());
```

Iterators

```rust
fn main() {
    let arr = [1, 2, 3];

    for item in arr.iter() {
        println!("{}", item);
    }
}
```

- `[1,2,3]` is an array (of type `[i32; 3]`).
- `.iter()` returns an **iterator** over **references** to the elements (`&i32`), not the values themselves.

Iterators

```
let arr = [1, 2, 3];

for item in arr.into_iter() {
    println!("{}", item);
}
```

If you want to iterate over **owned values**, use `.into_iter()`

Iterators

```rust
let mut arr = [1, 2, 3];

for item in arr.iter_mut() {
    *item += 1;
}

println!("{:?}", arr); // [2, 3, 4]
```

- If you want **mutable references**, use `.iter_mut()`

Why Iterators?

```
▶ Run | ⚙ Debug
fn main() {
    let arr: Vec<i32> = vec![1, 2, 3];

    for item: &i32 in arr.iter() {
        println!("{}", item);
    }
    println!("{:?}", arr); //arr is accessible
}
```

- `item` is of type `&i32` (a reference).
  The array remains **accessible** after the loop.
  No ownership transfer → Safe and efficient.
- **Use Case:** When you only need to read the elements.

```
fn main() {
    let arr: Vec<i32> = vec![1, 2, 3];

    for item: i32 in arr.into_iter() {
        println!("{}", item);
    }
    println!("{:?}", arr); //arr not accessible
}
```

- `item` is of type `i32` (owned).
- **The array is moved** (if it were a `Vec`, it would be emptied).
- **Not possible to use `arr` after this in case of `Vec<T>`.**

```
let mut arr = [1, 2, 3];

for item in arr.iter_mut() {
    *item += 1;
}

println!("{:?}", arr); // [2, 3, 4]
```

- `item` is of type `&mut i32` (mutable reference).
- Modifies elements **without moving the array**.
- **Use Case:** When you need to modify elements.

Iterators

The `&` in `Some(&1)` is present because the `iter()` method on arrays (`a.iter()`) produces an **iterator of references**, not the actual values.

Detailed Explanation

1. **`a.iter()` Behavior**:
   - The `.iter()` method creates an iterator that borrows elements from the array. It doesn't move or copy the values; instead, it yields **references** to the elements.
   - This allows you to iterate over the array without taking ownership, enabling further use of the array after iteration.
2. **Return Type of `iter.next()`**:
   - The `Iterator` trait's `next()` method for `.iter()` returns `Option<&T>` (an `Option` containing a reference to the value).
   - For example, in your case, `iter.next()` returns `Some(&1)`, where `&1` is a reference to the first element of the array.

Iterators

If you want to iterate and work with **values** instead of references, you can use `.into_iter()` (consumes the array, yielding values) or `cloned()` (creates a copy of the values):

```
let mut iter = a.into_iter();
assert_eq!(Some(1), iter.next());
```

```
let mut iter = a.iter().cloned();
assert_eq!(Some(1), iter.next());
```

Assignment - 2 : Iterators

Assignment: Implement a Custom Iterator in Rust

Problem Statement

You are required to implement a simple counter using Rust that behaves like an iterator. Your task is to define a `Counter` struct and implement the `Iterator` trait for it so that it returns the numbers from 1 to 5, one by one.

Requirements

1. Create a struct named `Counter` with a single field `count` of type u32.
2. Implement a method `new()` that initializes `Counter` with `count = 0`.
3. Implement the `Iterator` trait for `Counter`:
    - Set the associated `Item` type to u32.
    - Implement the `next()` method to:
        - Increment `count` each time it is called.
        - Return `Some(count)` while `count <= 5`.
        - Return `None` once the counter exceeds 5.
4. In the `main` function, use the iterator to print numbers 1 through 5 to the console.

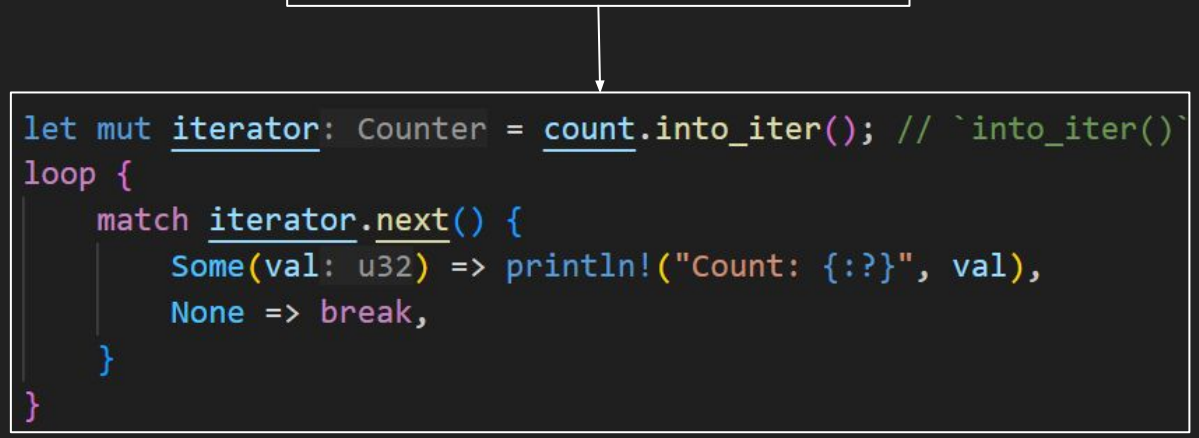
Iterators Assignment

```rust
// Define the Counter struct
2 implementations
struct Counter {
    count: u32,
}

// Implement methods for Counter
impl Counter {
    fn new() -> Counter {
        Counter { count: 0 }
    }
}

// Implement the Iterator trait for Counter
impl Iterator for Counter {
    type Item = u32;

    fn next(&mut self) -> Option<Self::Item> {
        self.count += 1;
        if self.count <= 5 {
            Some(self.count)
        } else {
            None
        }
    }
}
```

```rust
// Main function to use the iterator
▶ Run | ⚙ Debug
fn main() {
    let counter: Counter = Counter::new();

    for number: u32 in counter {
        println!("{}", number);
    }
}
```

Iterators behind the scenes

```rust
for val: u32 in count {
    println!("Count: {:?}", val);
}
```



```rust
let mut iterator: Counter = count.into_iter(); // `into_iter()`
loop {
    match iterator.next() {
        Some(val: u32) => println!("Count: {:?}", val),
        None => break,
    }
}
```

type

In **Rust**, the keyword `type` is used to create a **type alias**. A **type alias** gives a new name to an existing type, making complex types easier to read or use.

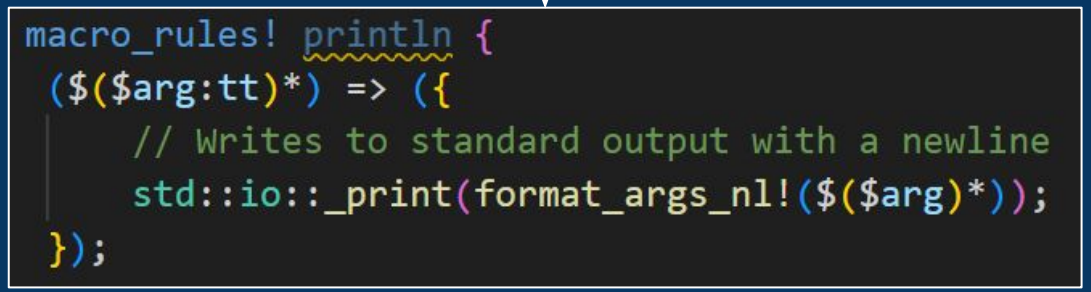
```
type NewName = ExistingType;
```

```
type Kilometers = i32;

fn add_distance(x: Kilometers, y: Kilometers) -> Kilometers {
    x + y
}

fn main() {
    let a: Kilometers = 5;
    let b: Kilometers = 10;
    println!("Total: {}", add_distance(a, b)); // Total: 15
}
```

Macros

Macros

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



```
macro_rules! println {
 ($($arg:tt)*) => ({
     // Writes to standard output with a newline
     std::io::_print(format_args_nl!($($arg)*));
 });
```

Macros

In Rust, macros are a way to perform metaprogramming—writing code that generates other code.

Unlike functions, macros operate at compile-time, which means they are expanded by the compiler before the program is run.

Types of Macros

Rust has two main types of macros:

1. **Declarative Macros**: Defined using `macro_rules!`, these are the most common and easier to use. Example - println!(),panic!(),vec![ ] etc.
2. **Procedural Macros**: More advanced and used for custom derive implementations or attribute-like and function-like macros.

Declarative Macro

```
macro_rules! say_hello {
    () => {
        println!("Hello, world!");
    };
}

fn main() {
    say_hello!(); // This expands to println!("Hello, world!");
}
```

Declarative Macro With Parameters

```rust
macro_rules! repeat_message {
    ($msg:expr, $times:expr) => {
        for _ in 0..$times {
            println!("{}", $msg);
        }
    };
}

fn main() {
    repeat_message!("Rust is awesome!", 3);
}
```

In Rust macros, the $ symbol and the expr are part of the syntax for declarative macros defined with macro_rules!.

The $ is used in macros to define **placeholders** for values or patterns. These placeholders represent the parts of the code you want to match and expand during macro invocation.

The $ is used in macros to define **placeholders** for values or patterns. These placeholders represent the parts of the code you want to match and expand during macro invocation. `$value` is a placeholder that gets replaced by the argument 42

Fragment Specifier

Common fragment specifiers include:

- **`expr`**: Matches any valid Rust expression (e.g., 42, `"hello"`, `1 + 2`, etc.).
- **`ident`**: Matches an identifier (e.g., `x`, `my_var`, `Foo`).
- **`ty`**: Matches a type (e.g., `i32`, `String`, `Vec<i32>`).
- **`pat`**: Matches a pattern (e.g., `Some(x)`, `42`).
- **`block`**: Matches a block of code (e.g., `{ println!("Hello"); }`).
- **`tt`**: Matches a single token tree, which can be anything valid in Rust's syntax.

expr

```
macro_rules! add {
    ($x:expr, $y:expr) => {
        $x + $y
    };
}

fn main() {
    let sum = add!(5, 10); // $x is 5 and $y is 10
    println!("Sum: {}", sum); // Output: Sum: 15
}
```

The `expr` is a **fragment specifier** in Rust's macro syntax. It tells the macro what kind of Rust syntax it should expect to match at that position.

- `$x:expr` matches the first argument (`5`).
- `$y:expr` matches the second argument (`10`).
- `expr` means both `5` and `10` are treated as Rust expressions.

ty

```
macro_rules! create_vector {
    ($type:ty) => {
        fn new_vector() -> Vec<$type> {
            Vec::new()
        }
    };
}

create_vector!(i32); // Expands to a function that creates a Vec<i32>

fn main() {
    let my_vec = new_vector();
    println!("Created a vector of type i32: {:?}", my_vec);
}
```

`$type:ty`: The `ty` fragment matches the input type (`i32` in this case).

The macro generates a function `new_vector` that returns a vector of the specified type.

`create_vector!(i32)` expands to:

```
fn new_vector() -> Vec<i32> {
    Vec::new()
}
```

name

```
macro_rules! create_struct {
    ($name:ident, $type:ty) => {
        struct $name {
            value: $type,
        }

        impl $name {
            fn new(value: $type) -> Self {
                Self { value }
            }

            fn get_value(&self) -> $type {
                self.value
            }
        }
    };
}

// Create a struct named MyStruct with a field of type i32
create_struct!(MyStruct, i32);

fn main() {
    let my_instance = MyStruct::new(42);
    println!("Value: {}", my_instance.get_value());
}
```

`$name:ident`: Matches an identifier for the struct name (e.g., `MyStruct`).

`$type:ty`: Matches the type of the field (e.g., `i32`).

`create_struct!(MyStruct,i32)` expands to:

```
struct MyStruct {
    value: i32,
}
impl MyStruct {
    fn new(value: i32) -> Self {
        Self { value }
    }

    fn get_value(&self) -> i32 {
        self.value
    }
}
```

Procedural Macro

Procedural macros allow for more complex code generation. They come in three forms:

- **Custom Derive Macros**: Used to derive traits for structs and enums.
- **Attribute-like Macros**: Applied to items like functions or modules.
- **Function-like Macros**: Look like function calls but process their input.

Custom derive procedural macro

```rust
use serde::Serialize;

#[derive(Serialize)]
struct User {
    name: String,
    age: u32,
}

fn main() {
    let user = User {
        name: "Alice".to_string(),
        age: 30,
    };
    println!("{}", serde_json::to_string(&user).unwrap());
}
```

In this case, the #[derive(Serialize)] macro generates the code required to serialize the User struct to JSON.

Function-like procedural macro

```rust
use regex::Regex;

fn main() {
    let re = Regex::new(r"\d+").unwrap(); // Create a regex pattern
    let text = "There are 42 apples";

    // Check if the regex matches the text
    if re.is_match(text) {
        println!("Match found!");
    } else {
        println!("No match.");
    }
}
```

r"\d+" matches one or more digits in a string.

Attribute-Like Macros

Attribute-like macros in Rust are used to apply custom behavior to Rust items such as functions, structs, modules, and more. These macros look like attributes (i.e., `#[some_macro]`), and they can modify or generate code for the item they are applied to.

A common example of an attribute-like macro is `#[derive]`, which automatically implements certain traits (like `Debug`, `Clone`, `Serialize`, etc.) for a struct or enum.

Error Handling

Error Categories

Recoverable Error - A recoverable error, such as if you try to open a file and that operation fails because the file doesn't exist, you might want to create the file instead of terminating the process.

Unrecoverable Error - Unrecoverable errors are always symptoms of bugs, such as trying to access a location beyond the end of an array, and so we want to immediately stop the program.

Note - Most languages don't distinguish between these two kinds of errors and handle both in the same way, using mechanisms such as exceptions. Rust doesn't have exceptions. Instead, it has the type Result<T, E> for recoverable errors and the panic! macro that stops execution when the program encounters an unrecoverable error.

Unrecoverable Errors with panic!

There are two ways to cause a panic in practice: by taking an action that causes our code to panic (such as accessing an array past the end) or by explicitly calling the panic! macro.

```rust
fn main() {
    panic!("crash and burn");
}
```



```rust
fn main() {
    let v = vec![1, 2, 3];

    v[100];
}
```

Recoverable Errors with Result

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

T represents the type of the value that will be returned in a success case within the Ok variant, and E represents the type of the error that will be returned in a failure case within the Err variant.

Recoverable Errors with Result

The `File::create` function in Rust's `std::fs` module returns a value of the type `Result<File, std::io::Error>`.

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

fn main() {
    match File::create("invalid\0path/hello.txt") {
        Ok(_file) => println!("File 'hello.txt' created successfully."),
        Err(error) => println!("Error creating file: {:?}", error),
    }
}
```

Err(error) if the operation fails, where error is an instance of std::io::Error.

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

fn main() {
    match File::create("hello.txt") {
        Ok(file) => println!("File 'hello.txt' created successfully."),
        Err(error) => println!("Error creating file: {:?}", error),
    }
}
```

Ok(file) if the operation succeeds, where file is an instance of std::fs::File

Recoverable Errors

```rust
use std::fs::File;
use std::io::ErrorKind;

fn main() {
    match File::create("invalid\0path/hello.txt") {
        Ok(_) => println!("File created successfully."),
        Err(error) => match error.kind() {
            ErrorKind::InvalidInput => panic!("Invalid path provided!"),
            other_error => panic!("An error occurred: {:?}", other_error),
        },
    }
}
```


We can use the kind method of the std::io::Error type to determine the type of error. This method returns an ErrorKind enum, which provides variants like NotFound, PermissionDenied, InvalidInput, etc.

`error.kind()`:

- This method retrieves the `ErrorKind` of the error, which is an enum representing categories of I/O errors.

Matching on `ErrorKind`:

- Check if the error is `ErrorKind::InvalidInput` (used for invalid file paths).
- Use the _ or specific variants for other types of errors, such as `PermissionDenied`, `NotFound`, etc.

unwrap

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

fn main() {
    let file = File::create("invalid\0path/hello.txt").unwrap();
    println!("File created: {:?}", file);
}
```

unwrap() with Result, but it will panic if the Result is Err. Using unwrap() without proper context is generally discouraged in production code since it doesn't allow for customized error handling or meaningful error messages.

How `unwrap()` Works:

- If the `Result` is `Ok`, it extracts the value inside.
- If the `Result` is `Err`, it panics with a default message showing the error.

unwrap_or_else

```rust
fn main() {
    let option: Option<i32> = Some(42);
    let value: i32 = option.unwrap_or_else(|| 0); // returns 42

    let none_option: Option<i32> = None;
    let value: i32 = none_option.unwrap_or_else(|| 0); // returns 0, no panic

    // You can also have custom logic in the closure
    let none_option: Option<i32> = None;
    let value: i32 = none_option.unwrap_or_else(|| {
        println!("No value found!");
        100 // returns 100 after printing the message
    });
}
```

It allows you to provide a fallback or default value (or perform some logic) when the Option or Result is None or Err, respectively, without panicking.

Customizable unwrap

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

fn main() {
    let _file = File::create("invalid\0path/hello.txt").unwrap_or_else(|error| {
        if error.kind() == std::io::ErrorKind::InvalidInput {
            panic!("Invalid path provided!");
        } else {
            panic!("An unexpected error occurred: {:?}", error);
        }
    });
}
```

The `|error|` in the `unwrap_or_else` method is a **closure** in Rust. Closures are anonymous functions that can capture variables from their surrounding scope.

In this specific context, `|error|` is the syntax for a closure that takes one argument, `error`, representing the `Err` value from the `Result`.

expect

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

fn main() {
    let file = File::create("invalid\0path/hello.txt").expect("file could not be created");
    println!("File created: {:?}", file);
}
```

the expect method lets us also choose the panic! error message. Using expect instead of unwrap and providing good error messages can convey your intent and make tracking down the source of a panic easier.

In production-quality code, most Rustaceans choose expect rather than unwrap and give more context about why the operation is expected to always succeed.

Error Propagation

Error propagation allows a function to pass errors to the calling code instead of handling them directly, giving more flexibility to the caller.

```rust
use std::fs::File;
use std::io;

fn create_file(file_path: &str) -> Result<File, io::Error> {
    // Attempt to create a file and propagate errors with `?`
    let file = File::create(file_path)?;
    Ok(file)
}

fn main() {
    match create_file("invalid\0path/hello.txt") {
        Ok(_) => println!("File created successfully."),
        Err(e) => println!("Failed to create file: {:?}", e),
    }
}
```

The ? operator propagates the error from `File::create` to the calling function (`main`) if it fails.

If successful, it returns the `File`.

Unwrap() vs Unwrap_or_else

https://chatgpt.com/share/6818a7ff-cb64-800f-95f9-183a10e20bbb

Project - 1: CLI Parser

What Are We Trying to Do?

The goal is to:

1. **Read a Markdown file** (like `README.md`).
2. **Parse and convert** the Markdown into **HTML** using a Markdown parser.
3. **Wrap the resulting HTML** in a full HTML document (with `<html>`, `<head>`, etc.).
4. **Print it or save it** to a file.

CLAP

When you run a program from the **command line (terminal)**, you can pass **extra inputs** like:

```
my_program --name Alice --age 25
```

These **extra inputs** after the program name (like `--name Alice`) are called **command-line arguments**.

But your program doesn't automatically understand them — it just sees a list of strings like:

```
["my_program", "--name", "Alice", "--age", "25"]
```

CLAP

To **make sense of these strings**, you need to **parse** them — extract values like:

- Name = "Alice"
- Age = 25

This process of turning raw strings from the terminal into meaningful variables is called Command Line Argument Parsing.

Why do we need a parser in Rust?

In Rust, the raw arguments are accessible via:

```
std::env::args()
```

But using it manually is tedious, error-prone, and lacks features like:

- Checking if required arguments are missing
- Showing help messages
- Handling different formats (like `--name`, `-n`, or positional values)

That's where **Clap** comes in — it is a **command line argument parser** library that helps Rust programs **understand user inputs** from the terminal easily and safely.

CLAP crate

To **pass arguments to *your program***, you use `--` to tell Cargo: "Hey, everything after this goes to the binary you're running, not to Cargo itself."

Without `--` Cargo would think `--name` is an argument for **Cargo itself**, which it doesn't recognize—so you'd get an error.

```rust
use clap::Parser;

/// Simple program to greet a person
#[derive(Parser, Debug)]
#[command(name = "greeter", version, about = "Says hello", long_about = None)]
5 implementations
struct Args {
    /// Name of the person to greet
    #[arg(short, long)]
    name: String,
}

▶ Run | ⚙ Debug
fn main() {
    let args: Args = Args::parse();

    println!("Hello, {}!", args.name);
}
```

```
cargo run -- --name Alice
```

```
Hello, Alice!
```

```
cargo run -- --help
```

```
Says hello

Usage: clap.exe --name <NAME>

Options:
  -n, --name <NAME>  Name of the person to greet
  -h, --help         Print help
  -V, --version      Print version
```

Specifies a **short flag** (single character) for the argument.This is used with a single dash `-` on the command line.Example - greeter -n Alice
Specifies a **long flag** (full word) for the argument. This is used with two dashes `--`. Example - greeter --name Alice

Options

`Options` comes from the `pulldown_cmark` crate — a popular Markdown parser in Rust.

It is a **bitflag struct** that lets you **enable extra features** beyond the basic Markdown syntax. These features are not always turned on by default.

```
options.insert(Options::ENABLE_STRIKETHROUGH);
```

This tells the parser to support ~~strikethrough~~ syntax (e.g., `~~this~~` becomes ~~this~~).

Without this option, if your Markdown contains `~~text~~`, it would just stay as `~~text~~` instead of being converted to `<del>text</del>` in HTML.

PathBuf

`PathBuf` is a type provided by Rust's standard library (`std::path::PathBuf`) that represents a **file system path** in an **owned and growable** way.

Think of it like a smarter, platform-independent version of a `String` that understands file paths.

Why Use `PathBuf` Instead of `String`?

Because `PathBuf`:

- Is **cross-platform safe** (handles `/` vs `\` automatically).
- Can easily join paths (`push`, `join`).
- Works with file APIs like `fs::read_to_string`, `fs::write`, etc.
- Can be converted to/from `&str`, `OsStr`, or `String`.

html::push_html

This is converting **parsed Markdown** into **HTML**.

```
html::push_html(&mut html_output, parser);
```

Here's what it does step by step:

1. **`parser`**

This is an iterator created by `MarkdownParser::new_ext(...)` — it processes the Markdown text and yields **events** (like start of a paragraph, text, end of a header, etc.).

2. **`html::push_html(...)`**

This function **consumes** that stream of events and **writes HTML** into the `html_output` string.

html::push_html

`markdown_input`: raw Markdown text (e.g., `"# Hello\nThis is a **test**."`)

`parser`: turns that Markdown into a stream of "paragraph started", "text node: 'Hello'", etc.

`html_output`: empty string that you want to fill with HTML

`html::push_html(...)`: takes those parsed events and writes the equivalent HTML into the string

```
# Hello
This is a **test**.
```

push_html →

```
<h1>Hello</h1>
<p>This is a <strong>test</strong>.</p>
```

CLAP Project

Full Project - Click Here

Closures

Closures

In Rust, **closures** are anonymous functions that can capture variables from their surrounding scope. Closures can be stored in variables, passed as arguments, or returned from functions.

Characteristics of Closures:

1. They can capture variables from their environment.
2. They can infer types of arguments and return values, but you can also specify them explicitly.
3. Closures implement one of the following traits depending on how they use captured variables:
    - `Fn`: Borrows variables immutably.
    - `FnMut`: Borrows variables mutably.
    - `FnOnce`: Takes ownership of captured variables.

Closures

`|x: i32| x + 1` is the closure.

`|x|` defines the closure's input parameter.

The body `x + 1` returns the result.

```rust
fn main() {
    // A closure that adds 1 to its input
    let add_one = |x: i32| x + 1;

    // Using the closure
    let result = add_one(5);
    println!("5 + 1 = {}", result); // Output: 5 + 1 = 6
}
```

Closures implementing FnMut Trait

```rust
fn main() {
    let mut counter = 0;

    let mut increase = || {
        counter += 1; // The closure changes `counter`
        println!("Counter is now: {}", counter);
    };

    increase(); // Counter is now: 1
    increase(); // Counter is now: 2
}
```

Closures & Ownership Transfer

```rust
fn main() {
    let x: String = String::from("hello");
    let consume_and_return_x: impl FnOnce() -> String = || x;

    println!("{}", x);

    let y: String = consume_and_return_x();

    println!("{}", y);

    let z: String = consume_and_return_x();
    println!("{}", z);
}
```

After this point, x is no longer accessible outside the closure because ownership has been transferred.

Error! x has been moved into the closure, so this line is invalid and will cause a compilation error.

The first call to `consume_and_return_x()` moves ownership of x out of the closure into y.
Now, y owns the `String` and can be printed.

Error! `consume_and_return_x` was already called once, consuming x permanently.
The closure no longer has x, so calling it again results in a **use-after-move** error

Closures & Reference

```rust
fn main() {
    let x: String = String::from("hello");
    let consume_and_return_x: impl Fn() -> &String = || &x;

    println!("{}", x);

    let y: &String = consume_and_return_x();

    println!("{}", y);

    let z: &String = consume_and_return_x();
    println!("{}", z);
}
```

x is borrowed instead of moved, allowing multiple calls.

Closures with Vectors

```rust
fn main() {
    let numbers = vec![3, 1, 4, 2, 5];
    let even_numbers: Vec<i32> = numbers.into_iter().filter(|x| x % 2 == 0).collect();
    println!("{:?}", even_numbers); // Prints: [4, 2]
}
```

Iterator adapter methods

filter

```rust
fn main() {
    let numbers = vec![3, 1, 4, 2, 5];
    let even_numbers: Vec<i32> = numbers.into_iter().filter(|x| x % 2 == 0).collect();
    println!("{:?}", even_numbers); // Prints: [4, 2]
}
```

Assignment - 3 : Iterator Methods & Closures

Assignment - 3

Double the numbers

- Given a vector of integers, use `map` to return a new vector where each number is doubled.
- Example: `vec![1, 2, 3]` → `vec![2, 4, 6]`

Filter even numbers

- Given a list of integers, use `filter` to return only the even numbers.
- Example: `vec![1, 2, 3, 4]` → `vec![2, 4]`

Sum of all numbers

- Use `reduce` or `fold` to compute the sum of all elements in a vector.
- Example: `vec![1, 2, 3, 4]` → `10`

Assignment - 3

Given the following vector of integers: let numbers = vec![1, 2, 3, 4, 5];

Step-by-step Goals:

1. **Convert the vector into an iterator** using `.iter()`.
2. **Filter only the odd numbers** from the list using `.filter(...)`.
3. **Increment each odd number by 1** using `.map(...)`.
4. **Find the first item in the result** that is equal to 6 using `.find(...)`.
5. **Print the final result** using `println!`.

```rust
fn main() {
    let arr: Vec<i32> = vec![1, 2, 3, 4, 6, 8];
    let filter_arr: Vec<i32> = arr.into_iter().filter(|x: &i32| x % 2 == 0).collect();
    println!("Filtered vector:{:?}", filter_arr);
}
```

```rust
fn main() {
    let arr: Vec<i32> = vec![1, 2, 3];
    let double_arr: Vec<i32> = arr.iter().map(|x: &i32| 2 * x).collect();
    println!("Double vector:{:?}", double_arr);
}
```

```rust
fn main() {
    let arr: Vec<i32> = vec![1, 2, 3];
    match arr.into_iter().reduce(|a: i32, b: i32| a + b) {
        Some(sum: i32) => println!("Sum is {}", sum),
        None => println!("This is none"),
    };
}
```

```rust
fn main() {
    let numbers = vec![1, 2, 3, 4, 5];

    let result = numbers
        .iter()
        .filter(|x| **x % 2 != 0)  // keep only odd numbers → [1, 3, 5]
        .map(|num| num + 1)        // increment each → [2, 4, 6]
        .find(|x| *x == 6);        // find first item equal to 6 → Some(6)

    println!("{:?}", result.unwrap());  // prints 6
}
```

Why Iterators are Lazy?

```rust
fn main() {
    let numbers: Vec<i32> = vec![1, 2, 3, 4, 5];

    // Nothing is executed yet
    let lazy_iter: impl Iterator<Item = i32> = numbers.iter().map(|x: &i32| {
        println!("Mapping {}", x);
        x * 2
    });

    println!("Before consuming iterator");

    // Now the iterator is consumed
    for value: i32 in lazy_iter {
        println!("Got: {}", value);
    }
}
```

What does "lazy" mean?

It means that:

- `.map()`, `.filter()`, etc. only describe what *should happen*.
- The actual action happens only when you consume the iterator.

```
Mapping 1
Got: 2
Mapping 2
Got: 4
Mapping 3
Got: 6
Mapping 4
Got: 8
Mapping 5
Got: 10
```

Packages & Crates

Packages & Crates

- Package
  - Often synonymous with the "project"
  - One or more crates that provide a set of functionality. More often than not, you will be dealing with only 1 crate in simple programs.
  - Contains a Cargo.toml file that describes how to build those crates
- Crate
  - A crate is a single computational unit - either a binary or a library
  - Binary crate (has a fn main, can be compiled into an executable)
  - Library crate (does not have a fn main, cannot be compiled into an executable)

Packages & Crates*

- cargo new <name>
  - creates a new Package with 1 Binary Crate represented by main.rs.
  - This is the option we use most of the time to create a new project
- cargo new <name> --lib
  - creates a new Package with 1 Library crate represented by lib.rs. There is no main.rs created.
  - We use this option to create an external library of code that can be used by another project

Modules and Use

Modules & Use

Rust finds the function in this sequence:

- within the current file
- looks for a ‘.rs’ file with the same name as the module name in the current folder
- looks for a folder with the module name and a file mod.rs inside, there it looks for the code.

Modules & Use

- Modules
  - Allow for the organization of code into groups for readability and re-use
  - Allow for the grouping of related definitions together
  - Flexibility for controlling the public / private visibility of items (such as functions), so as to limit their visibility to outside code
  - Can also hold definitions of other items inside, including other modules

Modules & Use

within the current file

```
// main.rs

fn add() {}

fn sub() {}

fn main() {

   add();

   sub();

}
```

Modules & Use

looks for a '.rs' file with the same name as the module name in the current folder

main.rs U

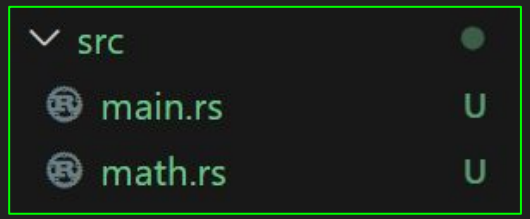
src > main.rs > ...

```rust
mod math;
use math::add;

▶ Run | ⚙ Debug
fn main() {
    add();
    math::sub();
}
```

math.rs U

src > math.rs > ...

```rust
pub fn add() {
    println!("{}", 5 + 6);
}
pub fn sub() {
    println!("{}", 6 - 5);
}
```



Modules & Use

looks for a folder with the module name and a file mod.rs inside, there it looks for the code.

main.rs U

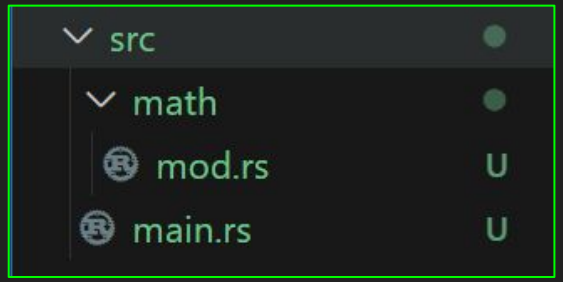
src > main.rs > ...

```rust
mod math;
use math::add;

▶ Run | ⚙ Debug
fn main() {
    add();
    math::sub();
}
```

math.rs U

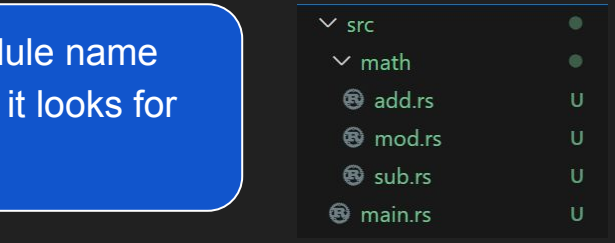
math.rs > ...

```rust
pub fn add() {
    println!("{}", 5 + 6);
}
pub fn sub() {
    println!("{}", 6 - 5);
}
```



Modules & Use

looks for a folder with the module name and a file mod.rs inside, there it looks for the code.




```rust
// main.rs


// declare module

mod math;

// use * to grant access to everything in math module

use math::*;


fn main() {

  add();

  sub();

}
```

```rust
// add.rs

pub fn add() {}
```

```rust
// sub.rs

pub fn sub() {}
```

```rust
// mod.rs

mod add;

mod sub;



pub use add::*;

pub use sub::*;
```

crate

main.rs U

src > main.rs > ...

```rust
/// If `add.rs` is a sibling file,
/// you must explicitly declare it
/// so the compiler knows it is a module.
mod add;

mod math;
use math::*;
▶ Run | ⚙ Debug
fn main() {
    let a: u8 = 5;
    let b: u8 = 6;
    add_u8(x: a, y: b);
}
```

math.rs U

src > math.rs > ...

```rust
use crate::add;

pub fn add_u8(x: u8, y: u8) {
    add::add_num(num_one: x, num_two: y);
}
```

add.rs U

src > add.rs > ...

```rust
pub fn add_num(num_one: u8, num_two: u8) {
    println!("Result is {}", num_one + num_two);
}
```

Hashmaps

Hashmap

Key	Value
Ravi	100
Raju	10
Lalu	1

Hashmap

In Rust, a **HashMap** is a **key-value store**, similar to dictionaries in Python or maps in Java.

Key Features:

- Stores data as key-value pairs.
- Fast lookup using a hash function.
- Keys must be **unique**.
- Provided by Rust's standard library in `std::collections`.

```rust
use std::collections::HashMap;
▶ Run | ⚙ Debug
fn main() {
    let mut students: HashMap<String, u32> = HashMap::new();
    students.insert(k: "Ravi".to_owned(), v: 100);
    students.insert(k: "Raju".to_owned(), v: 10);
    students.insert(k: "Lalu".to_owned(), v: 1);

    for (student: &String, marks: &u32) in students.iter() {
        println!("Student name:{:?} marks={}", student, marks);
    }

    students.insert(k: "Raju".to_owned(), v: 200);

    match students.get("Raju") {
        Some(marks: &u32) => println!("Found:{}", marks),
        None => println!("Not Found"),
    }
}
```

Smart Pointers

Smart Pointers

In **Rust**, a **smart pointer** is a data structure that not only **acts like a pointer** (i.e., it allows access to data) but also **manages ownership and memory** according to specific rules.

Smart Pointers

Smart Pointer	Description
`Box<T>`	Allocates data on the heap with single ownership. Lightweight.
`Rc<T>`	Reference-counted pointer for **shared ownership** in single-threaded scenarios.
`Arc<T>`	Atomic reference-counted pointer for **shared ownership across threads**.
`RefCell<T>`	Allows **interior mutability** at **runtime** in a single-threaded context.
`Mutex<T>`	Provides **interior mutability with thread safety**, blocks access when locked.
`RwLock<T>`	Similar to `Mutex<T>`, but allows **multiple readers or one writer** at a time.

Box <T>

Box<T> is a smart pointer that provides heap allocation for values. It's often the simplest way to store data on the heap rather than the stack.

```rust
fn main() {
    let b: Box<i32> = Box::new(10); // `10` is stored on the heap
    println!("b = {}", b);
}
```

Lifetimes

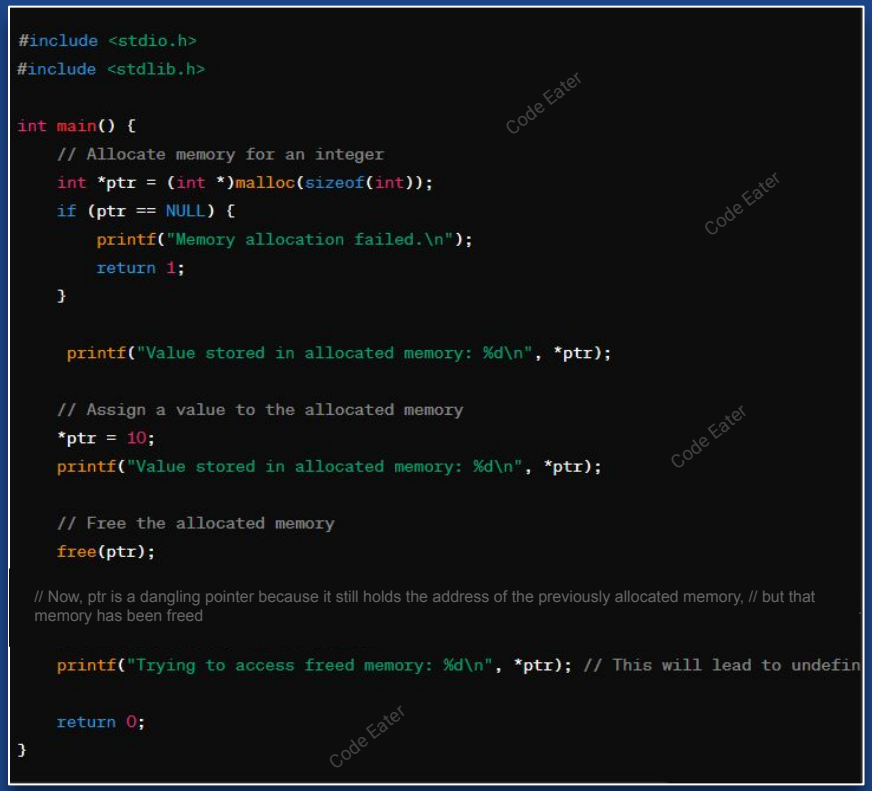
Lifetime - Three Golden Points

The primary objective of lifetimes in Rust is to avoid dangling references

Rust reject code that could lead to memory safety issues.

When there is a reference, there may be a chance of a dangling reference — and Rust uses **lifetimes** to prevent that.

Dangling Pointers in C



Dangling Reference

A dangling reference occurs when a program tries to access data that has already been deallocated or gone out of scope, leading to undefined behavior and potential memory safety issues.

By tracking the scope for which a reference is valid, Rust's borrow checker can enforce rules at compile time, guaranteeing memory safety without runtime overhead.

Lifetimes

```rust
fn main() {
    let i = 3; // Lifetime for `i` starts.
    //
    { //
        let borrow1 = &i; // `borrow1` lifetime starts.
        //
        println!("borrow1: {}", borrow1); //
    } // `borrow1` ends.
    //
    //
    { //
        let borrow2 = &i; // `borrow2` lifetime starts.
        //
        println!("borrow2: {}", borrow2); //
    } // `borrow2` ends.
    //
} // Lifetime ends.
```

A lifetime is a construct the compiler (or more specifically, its borrow checker) uses to ensure all borrows are valid.

Specifically, a variable's lifetime begins when it is created and ends when it is destroyed.

While lifetimes and scopes are often referred to together, they are not the same.

Lifetime Issue

```rust
fn main() {
    let r: &i32; // Introduce reference: `r`.
    let i: i32 = 1; // Introduce scoped value: `i`.
    r = &i; // Store reference of `i` in `r`.
    println!("{}", r); // `r` still refers to `i`.
}
```

```rust
fn main() {
    let r: &i32; // Introduce reference: `r`.
    {
        let i: i32 = 1; // Introduce scoped value: `i`.
        r = &i; // Store reference of `i` in `r`.
    } // `i` goes out of scope and is dropped.

    println!("{}", r); // `r` still refers to `i`.
}
```

1. You created a resource.
2. You lend a reference to the resource to someone say X.
3. You decided to deallocate the resource while the reference to the resource still exist.
4. X decides to use the resource.

Borrower Checker

```rust
fn main() {
    let r: &i32; // Introduce reference: `r`.
    {
        let i: i32 = 1; // Introduce scoped value: `i`.
        r = &i; // Store reference of `i` in `r`.
    } // `i` goes out of scope and is dropped.

    println!("{}", r); // `r` still refers to `i`.
}
```

The borrow checker is a component of the Rust compiler that examines the scopes (lifetimes) of variables and references to determine if all borrows are valid. It compares the lifetime of a reference with the lifetime of the data it points to. If a reference attempts to outlive the data it refers to, the borrow checker will prevent the code from compiling.

For example, if a variable r is a reference to i, and i goes out of scope before r is used, the borrow checker will identify this as an invalid borrow because x "does not live long enough" for r to be valid for its entire scope.

Lifetimes vs Scope

```
fn main() {
    let x: i32 = 5;              // -------------------+-- 'b
                                 //                    |
    let r: &i32 = &x;            // --+--'a            |
                                 //   |                |
    println!("r: {}", r);        //   |                |
}                                // -------------------+
```

'b (x) outlives 'a (r) → Allowed

```
fn main() {
    let r: &i32;                 // -------------------+-- 'a
                                 //                    |
    {                            //                    |
        let x: i32 = 5;          //     --+-- 'b       |
        r = &x;                  //       |            |
    }                            //     --+            |
                                 //                    |
    println!("r: {}", r);        //                    |
}                                // -------------------+
```

'a (r) outlives 'b (x) → Not allowed

Lifetimes vs Scope

lifetime_of_reference <= lifetime_of_data_it_points_to

Lifetimes vs Scope

Concept	Applies to
Scope	A variable or reference
Lifetime	A reference

Lifetimes are named regions of code that a reference must be valid for. Click Here.

the lifetimes will coincide with scopes. This is because our examples are simple. The more complex cases where they don't coincide

Specifically, a variable's lifetime begins when it is created and ends when it is destroyed. While lifetimes and scopes are often referred to together, they are not the same. Click Here.

The borrow has a lifetime that is determined by where it is declared. As a result, the borrow is valid as long as it ends before the lender is destroyed.

However, the scope of the borrow is determined by where the reference is used.

Assignment - 4 : Lifetime

Lifetimes Assignment

```
fn main() {
    let x = 10;
    let y = get_val();

    println!("x+y:{}", x + y);
}
fn get_val() -> &i32 {
    let y = 5;
    &y
}
```

A. It will compile and print: x+y : 15

B. It will not compile due to a lifetime error

C. It will panic at runtime due to accessing a freed variable

D. It will compile, but result in undefined behavior

Explicit Annotations

foo<'a> - `foo` has a lifetime parameter `'a`

This lifetime syntax indicates that the lifetime of foo may not exceed that of 'a.

foo<'a, 'b> - `foo` has lifetime parameters `'a` and `'b`

In this case, the lifetime of foo cannot exceed that of either 'a or 'b

Explicit Annotations

```
// `print_refs` takes two references to `i32` which have different
// lifetimes `'a` and `'b`. These two lifetimes must both be at
// least as long as the function `print_refs`.
fn print_refs<'a, 'b>(x: &'a i32, y: &'b i32) {
    println!("x is {} and y is {}", x, y);
}

▶ Run | ⚙ Debug
fn main() {
    // Create variables to be borrowed below.
    let (four: i32, nine: i32) = (4, 9);

    // Borrows (`&`) of both variables are passed into the function.
    print_refs(x: &four, y: &nine);
    // Any input which is borrowed must outlive the borrower.
    // In other words, the lifetime of `four` and `nine` must
    // be longer than that of `print_refs`.
}
```

The borrower" here is referring to the *entire function* `print_refs`, because it's the one **borrowing** the references `&four` and `&nine`.

Lifetime Parameter

```rust
fn main() {
    let result: &i32;
    let x: i32 = 10;

    let y: i32 = 20;
    result = larger(m: &x, n: &y);

    println!("{}", result);
}

fn larger(m: &i32, n: &i32) -> &i32 {
    if m > n { m } else { n }
}
```

Rust's **borrow checker** needs to know how long the returned reference will live, **relative to the inputs**. You're telling Rust: "I'll return a reference, but I won't say whether it's tied to `m` or `n`.

This leads to ambiguity:

- Does the returned reference live as long as `m`?
- Or as long as `n`?
- Or the shorter of the two?

Lifetime Parameter

```
fn main() {
    let result: &i32;
    let x: i32 = 10;

    let y: i32 = 20;
    result = larger(m: &x, n: &y);

    println!("{}", result);
}

fn larger<'a>(m: &'a i32, n: &'a i32) -> &'a i32 {
    if m > n { m } else { n }
}
```

Both `m` and `n` must live at least as long as `'a`, and the return value will live as long as `'a` too

Lifetime Parameter

' a represents the lifetime of the reference s1.
' b represents the lifetime of the reference s2.

These lifetimes tell the compiler:

1. The reference s1 has a lifetime ' a, and it must be valid for at least that long.
2. The reference s2 has a lifetime ' b, and it must be valid for at least that long.

```rust
fn main() {
    let result: &str;
    let s1: String = String::from("Hello");

    let s2: String = String::from("Rust");
    result = longest(&s1, &s2); // ❌ ERROR: `s2` does not live long enough

    println!("{}", result); // ❌ ERROR: `result` holds an invalid reference
}

fn longest<'a>(s1: &'a str, s2: &'a str) -> &'a str {
    if s1.len() > s2.len() {
        s1
    } else {
        s2
    }
}
```

```rust
fn main() {
    let result: &str;
    let s1: String = String::from("Hello");
    {
        let s2: String = String::from("Rust");
        result = longest(&s1, &s2); // ❌ ERROR: `s2` does not live long enough
    }

    println!("{}", result); // ❌ ERROR: `result` holds an invalid reference
}

fn longest<'a>(s1: &'a str, s2: &'a str) -> &'a str {
    if s1.len() > s2.len() {
        s1
    } else {
        s2
    }
}
```

Note - It's important to understand that lifetime annotations are descriptive, not prescriptive. This means that how long a reference is valid is determined by the code, not by the annotations. The annotations, however, give information about lifetimes to the compiler that uses them to check the validity of references. The compiler can do so without annotations in simple cases, but needs the programmer's support in complex scenarios.

Lifetime

Lifetime annotations don't change how long references live, but rather describe the relationships between the lifetimes of multiple references.

No Reference No Tension

```
struct Path {
    point_x: i32,
    point_y: i32,
}

fn main() {
    let p_x = 10;
    let p_y = 20;
    let game = Path { point_x: p_x, point_y: p_y };
    println!("x = {}, y = {}", game.point_x, game.point_y);
}
```

Without any references there will be no errors.

Lifetime Error

```rust
struct Path {
    point_x: & i32,
    point_y: & i32,
}

fn main() {
    let p_x = 10;
    let p_y = 20;
    let game = Path { point_x: &p_x, point_y: &p_y };
    println!("x = {}, y = {}", game.point_x, game.point_y);
}
```

expected named lifetime parameter

The program fails because of **lifetime issues with references** in the `Path` struct. Specifically:

1. The fields `point_x` and `point_y` in the `Path` struct are references (`&i32`), but the struct does not have an explicit **lifetime annotation** to specify how long the references should be valid. Without this, the compiler cannot guarantee the safety of the references.
2. In Rust, references must always have a valid lifetime, ensuring that they do not outlive the values they point to. The fields `point_x` and `point_y` in the `Path` struct are referencing local variables (`p_x` and `p_y`) that are defined inside the `main` function. Once `p_x` and `p_y` go out of scope, the references in the `Path` struct become invalid.

Lifetime Error Resolved

```rust
struct Path<'a> {
    point_x: &'a i32,
    point_y: &'a i32,
}

fn main() {
    let p_x = 10;
    let p_y = 20;
    let game = Path { point_x: &p_x, point_y: &p_y };
    println!("x = {}, y = {}", game.point_x, game.point_y);
}
```

With explicit lifetime

Explanation:

1. The `'a` lifetime annotation in `struct Path<'a>` tells the compiler that the references in `Path` must have the same lifetime `'a`.
2. The references to `p_x` and `p_y` are valid for the duration of the `main` function, so this satisfies the lifetime requirement.

static

‘static

`'static` means: **"This thing lives forever – as long as the program runs."**So, if something has a `'static` lifetime, it will **never be removed from memory** until the program ends.

`The 'static lifetime is a special lifetime in Rust that denotes that a reference can live for the entire duration of the program.`

```
let s: &'static str = "hello";
```

`"hello"` is a **string literal and of type &’static**

It is **stored in the program’s memory permanently**.

This string is **not created at runtime**. Instead:

✅ Rust puts it **directly into the program’s compiled file** (your `.exe` or binary).
✅ When you run the program, this string is **already there in memory**.
✅ It is **never deleted** until the program ends.

static

```rust
// A function which takes no arguments, but has a lifetime parameter `'a`.\
fn failed_borrow<'a>() {
    let _x: i32 = 12;

    // ERROR: `_x` does not live long enough
    let _y: &'a i32 = &_x;
    // Attempting to use the lifetime `'a` as an explicit type annotation
    // inside the function will fail because the lifetime of `&_x` is shorter
    // than that of `_y`. A short lifetime cannot be coerced into a longer one.
}

▶ Run | ⚙ Debug
fn main() {
    failed_borrow();
    // `failed_borrow` contains no references to force `'a` to be
    // longer than the lifetime of the function, but `'a` is longer.
    // Because the lifetime is never constrained, it defaults to `'static`.
}
```

'a is a generic lifetime parameter - Inside this function, we will use references that live at least as long as 'a.

This function doesn't **accept** any references from outside (no parameters), so 'a is unconstrained.

When 'a is unconstrained, **Rust defaults it to `'static`**, the longest possible lifetime.

_x is a **local variable**, and its lifetime is **limited to the function block**.

&_x is a reference to _x, which lives only **until the end of this function**.

But _y is declared with type `&'a i32`, and 'a is defaulting to `'static` (very long-lived).

You are trying to store a short-lived reference (&_x) in a variable that promises to be long-lived (`_y: &'a i32`), which is **not allowed**.

Project - 2 : Rust Server

Project - 2 : Rust Server

Full Project - Click Here

Thank You