Enums with Data

You can attach data to each variant of an enum.

Enum Variants with Data

```
enum Message {
    Text(String),
    Move { x: i32, y: i32 },
    Quit,
}

fn process_message(msg: Message) {
    match msg {
        Message::Text(content) => println!("Received text: {}", content),
        Message::Move { x, y } => println!("Moving to ({}, {})", x, y),
        Message::Quit => println!("Exiting..."),
    }
}

fn main() {
    let msg1 = Message::Text(String::from("Hello, Rust!"));
    let msg2 = Message::Move { x: 10, y: 20 };
    let msg3 = Message::Quit;

    process_message(msg1);
    process_message(msg2);
    process_message(msg3);
}
```

- Enums can store different types of data per variant.
- More powerful than structs when modeling multiple possibilities.

The Power of Option<T>

Rust doesn't have null values. Instead, it uses **Option<T>** to represent "a value or nothing."

Using Option<T>

```
fn divide(a: f64, b: f64) -> Option<f64> {
    if b == 0.0 {
        None
    } else {
        Some(a / b)
    }
}

fn main() {
    let result = divide(10.0, 2.0);
    match result {
        Some(value) => println!("Result: {}", value),
        None => println!("Cannot divide by zero!"),
    }
}
```

Why use Option<T>?

- Prevents null reference errors
- Forces handling of missing values

Collections

Rust provides several built-in collections to store and manage data efficiently.

Defining an Array

```
fn main() {
    let numbers: [i32; 5] = [1, 2, 3, 4, 5];
    println!("First element: {}", numbers[0]);
    println!("Array length: {}", numbers.len());
}
```

- Arrays have a fixed size ([T; N]).
- Access elements using indexing (numbers[0]).

Slices – Borrowed Views of Arrays

A slice is a dynamically sized view into an array.

```
fn main() {
    let arr = [10, 20, 30, 40, 50];
    let slice: &[i32] = &arr[1..4]; // Slice from index 1 to 3

    println!("Slice: {:?}", slice); // Output: [20, 30, 40]
}
```

- Slices borrow a portion of an array.
- Uses range syntax ([start..end]).

Tuples

Defining and Using Tuples

```
fn main() {
    let person: (String, u32, bool) = (String::from("Alice"), 30, true);

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

- Tuples store multiple types together.
- Use dot notation (**.0**, **.1**, **.2**) to access elements.

Tuple Destructuring

```
fn main() {
    let (x, y, z) = (10, 20, "Rust");
    println!("x: {}, y: {}, z: {}", x, y, z);
}
```

- Destructure tuples into variables easily.

Vectors (Vec<T>)

Creating a Vector

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

    println!("Vector: {:?}", numbers);
    println!("First element: {}", numbers[0]);
}
```

- Vectors are growable arrays.
- Use **.push()** to add elements.

Iterating Over a Vector

```
fn main() {
    let numbers = vec![10, 20, 30];
    for num in &numbers {
        println!("{}", num);
    }
}
```

Iterate using a for loop with & to borrow elements.

Removing Elements

```
fn main() {
    let mut numbers = vec![1, 2, 3, 4, 5];
    numbers.pop(); // Removes last element
    println!("Updated vector: {:?}", numbers);
}
```

Use **.pop()** to remove the last item.

HashMaps (HashMap<K, V>)

Creating and Using a HashMap

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

fn main() {
    let mut scores = HashMap::new();
    scores.insert("Alice", 85);
    scores.insert("Bob", 92);

    println!("Scores: {:?}", scores);
}
```

- Stores key-value pairs.
- .insert(key, value) adds items.

Accessing Values in a HashMap

```
fn main() {
    let mut capitals = HashMap::new();
    capitals.insert("France", "Paris");
    capitals.insert("Japan", "Tokyo");

    match capitals.get("France") {
        Some(city) => println!("Capital of France: {}", city),
        None => println!("Not found!"),
    }
}
```

Use **.get(&key)** to retrieve values safely.

Iterating Over a HashMap

```
fn main() {
    let mut prices = HashMap::new();
    prices.insert("Apple", 2);
    prices.insert("Banana", 1);

    for (item, price) in &prices {
        println!("{} costs ${}", item, price);
    }
}
```

Iterate using for (key, value).

Strings (String vs &str)

Rust has two main string types:

- String (owned, mutable, heap-allocated)
- **&str** (borrowed, immutable, stack-allocated)

Creating Strings

```
fn main() {
    let s1 = String::from("Hello, Rust!");
    let s2 = "Hello, world!"; // String slice (&str)

    println!("{}", s1);
    println!("{}", s2);
}
```

- Use **String::from()** to create an owned string.
- String slices **(&str)** are references to static strings.

Concatenating Strings

```
fn main() {
    let hello = String::from("Hello");
    let world = String::from("World");

    let greeting = hello + " " + &world; // Note: hello is moved here
    println!("{}", greeting);
}
```

Use **+** or **.push_str()** to concatenate strings.

Iterating Over a String

```
fn main() {
    let text = "Rust";

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

.chars() iterates over Unicode characters.

Best Practices

- Use Arrays ([T; N]) for fixed-size data
- Slices (&[T]) to borrow parts of an array
- Tuples ((T1, T2)) to store mixed types
- Vectors (Vec<T>) for dynamic, growable lists
- HashMaps (HashMap<K, V>) for key-value storage
- Strings (String vs &str) for text handling

Error Handling

Rust's error handling system is powerful and safe, using **Result<T, E>**, **Option<T>**, and propagation mechanisms like the **?** operator.

Result<T, E> – Handling Recoverable Errors

The **Result<T, E>** type is used when an operation might fail and returns either:

- Ok(value) → Success
- Err(error) → Failure

Basic Example

```
fn divide(x: f64, y: f64) -> Result<f64, String> {
    if y == 0.0 {
        Err(String::from("Cannot divide by zero!"))
    } else {
        Ok(x / y)
    }
}

fn main() {
    match divide(10.0, 2.0) {
        Ok(result) => println!("Result: {}", result),
        Err(err) => println!("Error: {}", err),
    }
}
```

Use match to handle both **Ok** and **Err** cases.

Option<T> – Handling Absence of a Value

The **Option<T>** type is used when a value might be missing. It returns either:

- Some(value) → A value is present
- None → No value

Example: Finding an Element in a Vector

```
fn find_index(numbers: &[i32], target: i32) -> Option<usize> {
    for (index, &num) in numbers.iter().enumerate() {
        if num == target {
            return Some(index);
        }
    }
    None
}

fn main() {
    let numbers = vec![10, 20, 30, 40];

    match find_index(&numbers, 30) {
        Some(index) => println!("Found at index: {}", index),
        None => println!("Not found!"),
    }
}
```

Use **Option<T>** when a function might return "no result."

Unwrapping (unwrap, expect)

Using unwrap()

If you're sure an operation won't fail, you can use .unwrap() to extract the value.

```
fn main() {
    let name = Some("Alice");
    println!("Hello, {}", name.unwrap()); // Works fine
}
```

If the value is None, Rust will panic and crash the program.

Using expect()

expect() works like **unwrap()**, but provides a custom error message.

```
fn main() {
    let age: Option<i32> = None;
    println!("Age: {}", age.expect("No age found!")); // Panics with message
}
```

Use **expect()** only when failure should crash the program.

Error Propagation (? Operator) – Cleaner Code

Instead of handling errors immediately, propagate them up the call stack with ?.

Example: Reading a File with ?

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

fn read_file(filename: &str) -> Result<String, io::Error> {
    let mut file = File::open(filename)?; // Propagates error if file is missing
    let mut contents = String::new();
    file.read_to_string(&mut contents)?;
    Ok(contents)
}

fn main() {
    match read_file("hello.txt") {
        Ok(text) => println!("File contents: {}", text),
        Err(err) => println!("Error: {}", err),
    }
}
```

- **?** automatically returns an error if File::open or read_to_string fails.
- Only use **?** inside functions that return **Result<T, E>** or **Option<T>!**

Traits & Generics

Rust's Traits and Generics allow for code reuse, flexibility, and type abstraction.

Traits – Defining Shared Behavior

A trait in Rust is like an interface in other languages. It defines a set of methods that types can implement.

Defining a Trait

```
trait Greet {
    fn greet(&self) -> String;
}
```

Greet is a trait that defines a greet method.

Implementing a Trait

```
struct Person {
    name: String,
}

impl Greet for Person {
    fn greet(&self) -> String {
        format!("Hello, my name is {}!", self.name)
    }
}

fn main() {
    let p = Person { name: "Alice".to_string() };
    println!("{}", p.greet()); // Hello, my name is Alice!
}
```

Any struct that implements Greet must define greet().

Traits & Default Implementations

You can provide default method implementations inside a trait.

```
trait Greet {
    fn greet(&self) -> String {
        String::from("Hello!") // Default behavior
    }
}
```

Now, types implementing this trait don't have to define greet() unless they want to override it.

Traits & Default Implementations

You can provide default method implementations inside a trait.

```
trait Greet {
    fn greet(&self) -> String {
        String::from("Hello!") // Default behavior
    }
}
```

Traits with Multiple Implementations

A struct can implement multiple traits.

```
trait Walk {
    fn walk(&self);
}

trait Talk {
    fn talk(&self);
}

struct Robot;

impl Walk for Robot {
    fn walk(&self) {
        println!("Robot is walking...");
    }
}

impl Talk for Robot {
    fn talk(&self) {
        println!("Beep Boop!");
    }
}

fn main() {
    let r = Robot;
    r.walk();
    r.talk();
}
```

Multiple trait implementations work just like multiple interfaces in **OOP**.

Generics – Writing Flexible Code

Generics allow **functions**, **structs**, and **enums** to work with any type.

```
fn identity<T>(value: T) -> T {
    value
}

fn main() {
    println!("{}", identity(42)); // Works with integers
    println!("{}", identity("Rust")); // Works with strings
}
```

T is a placeholder for a type, making identity() work with any type.

Generics in Structs

```
struct Point<T> {
    x: T,
    y: T,
}

fn main() {
    let int_point = Point { x: 10, y: 20 }; // Works with integers
    let float_point = Point { x: 10.5, y: 20.3 }; // Works with floats
}
```

T allows Point<T> to store any type!

Generics in Traits

A trait can also use generics!

```
trait Add<T> {
    fn add(&self, other: T) -> T;
}

impl Add<i32> for i32 {
    fn add(&self, other: i32) -> i32 {
        self + other
    }
}

fn main() {
    let result = 10.add(20);
    println!("Sum: {}", result); // Sum: 30
}
```

Generics inside traits allow flexible implementations for different types.

Generics with where Clause

When constraints get complex, the where clause keeps things readable.

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

fn print_debug<T>(value: T)
where T: Debug
{
    println!("{:?}", value);
}

fn main() {
    print_debug(42);  // Prints: 42
    print_debug("Rust"); // Prints: "Rust"
}
```

T: Debug ensures that T implements the Debug trait, allowing it to be printed.

Modules & Crates

Modules (mod, use) – Organizing Code

Defining a Module

A module allows us to group related functions, structs, and traits together.

```
mod greetings {
    pub fn hello() {
        println!("Hello, Rustacean!");
    }
}
```

pub makes **hello()** accessible outside the module.

Using a Module (use)

To call a function from another module:

```
mod greetings {
    pub fn hello() {
        println!("Hello, Rustacean!");
    }
}

fn main() {
    greetings::hello(); // Calls the function from the module
}
```

greetings::hello() accesses **hello()** inside the greetings module.