# Vectors, Structs, Enums, Slices

- Vectors
  - A **vector** ( `Vec<T>` ) is a **growable, heap-allocated** list. Unlike arrays, which have a fixed size, vectors can **grow or shrink** at runtime.
  -

```
fn main() {
    let mut v = Vec::new();       // Create empty vector
    v.push(10);
    v.push(20);
    v.push(30);

    println!("{:?}", v);          // Output: [10, 20, 30]

    let third = v[2];
    println!("Third element is: {}", third);

    for val in &v {
        println!("Value: {}", val);
    }
}
```

  - Slices
    - A **slice** is a **view into a sequence** (like an array or a vector). It lets you borrow a **part** of a collection **without taking ownership**.
      - Slices:
      - Are **references** ( `&[T]` )
      - Don’t own data — they just point to it
      - Have a **start and length**, not an end index

    Exercise : (s2,s3,s4) in the example of vs code

```

//Internaly it is just pointer and length
struct Slice<T> {
    ptr: *const T,
    len: usize,
}
```

| | Concept | Description |
|---|---|---|
| 1 | Slice Type | `&[T]` or `&str` |
| 2 | Doesn’t own data | ✅ True |
| 3 | Read-only | ✅ (unless you use `&mut` ) |

| | | |
|---|---|---|
| 4 | Fast | ✅ No data copied |

- Structs
  - A **`struct`** is a **user-defined type** in Rust that lets you group together related data.
  - method vs associated functions
    -

</> Plain Text

```
struct User {
    name: String,
    age: u32,
    active: bool,
}

fn main() {
    let user1 = User {
        name: String::from("Alice"),
        age: 30,
        active: true,
    };

    println!("Name: {}, Age: {}, Active: {}", user1.name, user1.age, user1.active);
}



```

- Enums
  - When you need to model something that can be of different kinds, enums are the way to go
  -

</> Plain Text

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

fn main() {
    let signal = TrafficLight::Green;
    print_light(signal);
}


```

  - Exercise : use if , vs match vs if let

## Match Expression

- Just like switch case
  - **Exhaustive Checking**
  - **Pattern Matching**
  - **Destructuring**
  - **Guards**
  -

</> Rust

```rust
match value {
    pattern1 => expression1,
    pattern2 => expression2,
    // ...
    _ => default_expression,
}
```

Rust

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

fn main() {

    // simple matching

    let statusCode = 300;

    match statusCode {
        200 => println!("Success"),
        300 => println!("Seems like redirect"),
        _ => println!("Something else"),
    }

    // enum

    enum Coin {
        Penny,
        Nickel,
        Dime,
        Quarter,
    }

    fn value_in_cents(coin: Coin) -> u8 {
        match coin {
            Coin::Penny => 1,
            Coin::Nickel => 5,
            Coin::Dime => 10,
            Coin::Quarter => 25,
        }
    }


    // destructuring
    let point = Point { x: 0, y: 7 };

    match point {
        Point { x, y: 0 } => println!("On the x axis at {}", x),
        Point { x: 0, y } => println!("On the y axis at {}", y),
        Point { x, y } => println!("On neither axis: ({}, {})", x, y),
    }


    // matching with guards

    let num = Some(4);
    match num {
        Some(x) if x < 5 => println!("Less than five: {}", x),
        Some(x) => println!("{}", x),
        None => (),
    }
}
```

# The Option Enum in Rust

The `Option` enum is one of Rust's most important and frequently used types. It's a built-in enum that represents the concept of an optional value - every value is either "Some" value or "None" (no value).

## Definition

The `Option` enum is defined in Rust's standard library as:

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

```

Where:

- `Some(T)` represents a value of type `T`
- `None` represents the absence of a value

## Why Option Exists

Rust doesn't have `null` references like many other languages. Instead, it uses `Option` to:

1. Explicitly handle the case where a value might be absent
2. Force developers to consider both cases (Some/None)
3. Eliminate null pointer exceptions at compile time

## Basic Usage

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

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

```

## Common Methods

`Option` provides many useful methods:

1. **`unwrap()`**: Gets the value if Some, panics if None (avoid in production)

```rust
let x = Some(5);
println!("{}", x.unwrap()); // 5

```

1. **`unwrap_or(default)`**: Gets the value or returns a default

```rust
let x: Option<i32> = None;
println!("{}", x.unwrap_or(0)); // 0

```

1. **`map(f)`**: Applies a function to the contained value if Some

```rust
let x = Some(5);
let y = x.map(|v| v * 2); // Some(10)

```

1. **`and_then(f)`**: Chains operations that might return None

```rust
fn sqrt(x: f64) -> Option<f64> {
    if x >= 0.0 { Some(x.sqrt()) } else { None }
}

let x = Some(4.0).and_then(sqrt); // Some(2.0)

```

1. **`is_some()`/`is_none()`**: Check variants

```rust
let x = Some(5);
println!("{}", x.is_some()); // true

```

## Pattern Matching with Option

The most robust way to handle Options is with `match`:

```rust
fn print_number(maybe_num: Option<i32>) {
    match maybe_num {
        Some(num) => println!("Number: {}", num),
        None => println!("No number provided"),
    }
}

```

### When to Use Option

Use `Option` when:

- A function might not return a meaningful value
- A struct field might be empty
- You're working with values that could be missing
- You want to avoid null pointer errors

### Advantages over null

1. **Type safety**: The compiler forces you to handle both cases
2. **Explicit**: Code clearly shows where values might be missing
3. **Rich API**: Many helper methods for common operations
4. **No runtime cost**: `Option` is optimized to have no overhead

# The Result Type in Rust

The `Result` type is Rust's primary way to handle operations that might fail. It's an enum similar to `Option`, but instead of just `Some` /`None`, it has `Ok` for success and `Err` for failure cases.

### Basic Definition

Rust

```rust
enum Result<T, E> {
    Ok(T),    // Contains success value
    Err(E),   // Contains error value
}

```

### Key Differences from Option

1. **More expressive** - Carries error information
2. **Standardized error handling** - Used throughout Rust's stdlib
3. **For recoverable errors** - Unlike panics which are for unrecoverable errors

### Basic Usage Examples

#### 1. Simple Result Handling

Rust

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

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

```

### 2. File Operations (Common Real-World Use)

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

fn read_file(path: &str) -> Result<String, std::io::Error> {
    let mut file = File::open(path)?;
    let mut contents = String::new();
    std::io::Read::read_to_string(&mut file, &mut contents)?;
    Ok(contents)
}

```

### 3. Chaining Results

```rust
fn process_data(path: &str) -> Result<(), String> {
    let data = read_file(path).map_err(|e| format!("File error: {}", e))?;
    let parsed = parse_data(&data)?;
    save_results(parsed)?;
    Ok(())
}

```

## Common Methods

1. **`unwrap()`** - Gets the value if Ok, panics if Err (avoid in production)
2. **`unwrap_or(default)`** - Gets value or returns default
3. **`map(f)`** - Transforms Ok value
4. **`map_err(f)`** - Transforms Err value
5. **`and_then(f)`** - Chains operations that might fail
6. **`?` operator** - Early return on error

## The ? Operator

The question mark operator is syntax sugar for:

```rust
match result {
    Ok(v) => v,
    Err(e) => return Err(e.into()),
}

```

Example:

```rust
fn get_user(id: u32) -> Result<User, Error> {
    let conn = connect_db()?;       // ? returns if error
    let user = query_user(id, &conn)?;
    Ok(user)
}

```

## When to Use Result vs Option

Use `Result` when:

- The operation might fail
- You need to convey why it failed
- The caller should handle the failure

Use `Option` when:

- A value might logically be absent
- No explanation is needed for absence
- The "error case" is a normal part of program logic

## Converting Between Result and Option

```rust
// Result to Option
let maybe_value: Option<i32> = result.ok();

// Option to Result
let result: Result<i32, &str> = option.ok_or("Missing value");

```

The `Result` type is fundamental to Rust's error handling philosophy, forcing explicit handling of error cases while providing ergonomic ways to work with them.