Rust Exercises Solutions Manual

This document is designed as a companion to the book, providing detailed solutions to each challenge and exercise included in the chapters. Whether you're checking your work or seeking guidance on a particularly tricky problem, this guide aims to clarify the concepts and coding techniques presented in the book.

The purpose of this document is to:

- **Reinforce Learning**: By reviewing solutions, you can deepen your understanding of Rust's syntax, features, and best practices.
- **Offer Insight**: Each solution includes explanations that highlight the key concepts and steps involved, so you can understand not just the *how* but also the *why* behind the solution.
- **Encourage Independent Problem-Solving**: While the guide provides answers, it's best to attempt each challenge on your own first. Use this guide as a resource to validate your approach or to gain new perspectives on alternative methods.

How to Use This Guide

For each challenge, you'll find:

1. **The Challenge Recap**: A brief description of the challenge for quick reference.
2. **Solution Code**: The solution, written in idiomatic Rust, to meet the requirements of the exercise.
3. **Explanation and Analysis**: A breakdown of the solution code, explaining important concepts, choices made, and Rust's approach to solving the problem.

It's recommended to use this guide after completing each chapter's exercises, as reviewing the solutions with fresh insight will help you reinforce your skills in a structured way.

Happy coding, and enjoy your journey through Rust!

TABLE OF CONTENTS

Chapter 2: Basic Concepts

Challenge 1 Solution: Temperature Conversion Program

Step 1: Plan the Program Structure

- We need to:
  - Prompt the user for a temperature value.
  - Ask for the unit of the input temperature.
  - Convert the temperature to the other unit.
  - Display the result.
- We'll create two functions:
  - `celsius_to_fahrenheit(celsius: f64) -> f64`
  - `fahrenheit_to_celsius(fahrenheit: f64) -> f64`

Step 2: Implement the Conversion Functions

```
fn celsius_to_fahrenheit(celsius: f64) -> f64 {
    celsius * 9.0 / 5.0 + 32.0
}

fn fahrenheit_to_celsius(fahrenheit: f64) -> f64 {
    (fahrenheit - 32.0) * 5.0 / 9.0
}
```

Step 3: Write the Main Function

```
use std::io;

fn main() {
    // Prompt the user for a temperature value
    println!("Enter the temperature value:");
    let mut temp_input = String::new();
    io::stdin()
        .read_line(&mut temp_input)
        .expect("Failed to read input");
    let temp_value: f64 = match temp_input.trim().parse() {
        Ok(num) => num,
        Err(_) => {
            println!("Invalid temperature value.");
            return;
        }
    };

    // Ask for the unit of the input temperature
    println!("Is this in Celsius or Fahrenheit? (C/F):");
    let mut unit_input = String::new();
    io::stdin()
        .read_line(&mut unit_input)
```

```
        .expect("Failed to read input");
    let unit = unit_input.trim().to_uppercase();

    // Perform the conversion based on the unit
    if unit == "C" {
        let fahrenheit = celsius_to_fahrenheit(temp_value);
        println!(
            "{:.2}°C is equal to {:.2}°F",
            temp_value, fahrenheit
        );
    } else if unit == "F" {
        let celsius = fahrenheit_to_celsius(temp_value);
        println!(
            "{:.2}°F is equal to {:.2}°C",
            temp_value, celsius
        );
    } else {
        println!("Invalid unit. Please enter 'C' for Celsius or 'F' for Fahrenheit.");
    }
}
```

Explanation:

- **Input Handling:**
    - We use `io::stdin()` to read user input.
    - We trim and parse the temperature value, handling potential parsing errors.
    - We convert the unit input to uppercase to handle both lowercase and uppercase inputs.
- **Control Flow:**
    - We use `if` and `else if` statements to decide which conversion to perform based on the unit.
- **Error Handling:**
    - If the user enters an invalid temperature value or unit, we display an error message and terminate the program gracefully.

Complete Program:

```
use std::io;

fn celsius_to_fahrenheit(celsius: f64) -> f64 {
    celsius * 9.0 / 5.0 + 32.0
}

fn fahrenheit_to_celsius(fahrenheit: f64) -> f64 {
    (fahrenheit - 32.0) * 5.0 / 9.0
}

fn main() {
    // Prompt the user for a temperature value
    println!("Enter the temperature value:");
    let mut temp_input = String::new();
```

```
    io::stdin()
        .read_line(&mut temp_input)
        .expect("Failed to read input");
    let temp_value: f64 = match temp_input.trim().parse() {
        Ok(num) => num,
        Err(_) => {
            println!("Invalid temperature value.");
            return;
        }
    };

    // Ask for the unit of the input temperature
    println!("Is this in Celsius or Fahrenheit? (C/F):");
    let mut unit_input = String::new();
    io::stdin()
        .read_line(&mut unit_input)
        .expect("Failed to read input");
    let unit = unit_input.trim().to_uppercase();

    // Perform the conversion based on the unit
    if unit == "C" {
        let fahrenheit = celsius_to_fahrenheit(temp_value);
        println!(
            "{:.2}°C is equal to {:.2}°F",
            temp_value, fahrenheit
        );
    } else if unit == "F" {
        let celsius = fahrenheit_to_celsius(temp_value);
        println!(
            "{:.2}°F is equal to {:.2}°C",
            temp_value, celsius
        );
    } else {
        println!("Invalid unit. Please enter 'C' for Celsius or 'F' for Fahrenheit.");
    }
}
```

Challenge 2 Solution: Simple Calculator

Step 1: Plan the Program Structure

- The program needs to:
    - Prompt the user for two numbers.
    - Ask for the desired operation.
    - Perform the calculation.
    - Display the result.
- We'll create separate functions for each arithmetic operation.

Step 2: Implement Arithmetic Functions

```
fn add(a: f64, b: f64) -> f64 {
    a + b
}

fn subtract(a: f64, b: f64) -> f64 {
    a - b
}

fn multiply(a: f64, b: f64) -> f64 {
    a * b
}

fn divide(a: f64, b: f64) -> Result<f64, String> {
    if b == 0.0 {
        Err(String::from("Error: Division by zero"))
    } else {
        Ok(a / b)
    }
}
```

Step 3: Write the Main Function

```
use std::io;

fn main() {
    // Prompt the user for two numbers
    println!("Enter the first number:");
    let mut input1 = String::new();
    io::stdin()
        .read_line(&mut input1)
        .expect("Failed to read input");
    let num1: f64 = match input1.trim().parse() {
        Ok(n) => n,
        Err(_) => {
            println!("Invalid input for the first number.");
            return;
        }
    };

    println!("Enter the second number:");
    let mut input2 = String::new();
    io::stdin()
        .read_line(&mut input2)
        .expect("Failed to read input");
    let num2: f64 = match input2.trim().parse() {
        Ok(n) => n,
        Err(_) => {
            println!("Invalid input for the second number.");
            return;
        }
```

```
    };

    // Ask the user to select an operation
    println!("Select an operation (+, -, *, /):");
    let mut operation = String::new();
    io::stdin()
        .read_line(&mut operation)
        .expect("Failed to read input");
    let operation = operation.trim();

    // Perform the calculation based on the selected operation
    let result = match operation {
        "+" => Ok(add(num1, num2)),
        "-" => Ok(subtract(num1, num2)),
        "*" => Ok(multiply(num1, num2)),
        "/" => divide(num1, num2),
        _ => Err(String::from("Invalid operation selected.")),
    };

    // Display the result or error message
    match result {
        Ok(value) => println!("Result: {}", value),
        Err(e) => println!("{}", e),
    }
}
```

- **Input Handling:**
    - We read the two numbers and parse them as `f64`, handling parsing errors.
    - We read the operation as a string.
- **Control Flow:**
    - We use a `match` statement to decide which function to call based on the operation.
- **Error Handling:**
    - The `divide` function returns a `Result` to handle division by zero.
    - We handle invalid operations by returning an error.

Complete Program:

```
use std::io;

fn add(a: f64, b: f64) -> f64 {
    a + b
}

fn subtract(a: f64, b: f64) -> f64 {
    a - b
}

fn multiply(a: f64, b: f64) -> f64 {
    a * b
}
```

```rust
fn divide(a: f64, b: f64) -> Result<f64, String> {
    if b == 0.0 {
        Err(String::from("Error: Division by zero"))
    } else {
        Ok(a / b)
    }
}

fn main() {
    // Prompt the user for two numbers
    println!("Enter the first number:");
    let mut input1 = String::new();
    io::stdin()
        .read_line(&mut input1)
        .expect("Failed to read input");
    let num1: f64 = match input1.trim().parse() {
        Ok(n) => n,
        Err(_) => {
            println!("Invalid input for the first number.");
            return;
        }
    };

    println!("Enter the second number:");
    let mut input2 = String::new();
    io::stdin()
        .read_line(&mut input2)
        .expect("Failed to read input");
    let num2: f64 = match input2.trim().parse() {
        Ok(n) => n,
        Err(_) => {
            println!("Invalid input for the second number.");
            return;
        }
    };

    // Ask the user to select an operation
    println!("Select an operation (+, -, *, /):");
    let mut operation = String::new();
    io::stdin()
        .read_line(&mut operation)
        .expect("Failed to read input");
    let operation = operation.trim();

    // Perform the calculation based on the selected operation
    let result = match operation {
        "+" => Ok(add(num1, num2)),
        "-" => Ok(subtract(num1, num2)),
        "*" => Ok(multiply(num1, num2)),
```

```
        "/" => divide(num1, num2),
        _ => Err(String::from("Invalid operation selected.")),
    };

    // Display the result or error message
    match result {
        Ok(value) => println!("Result: {}", value),
        Err(e) => println!("{}", e),
    }
}
```

Challenge 3 Solution: Grade Classification

Step 1: Plan the Program Structure

- The program needs to:
    - Prompt the user for a numeric score.
    - Validate the input.
    - Determine the letter grade.
    - Display the result.
- We'll create a function `classify_grade(score: u32) -> char` to encapsulate the logic.

Step 2: Implement the Grade Classification Function

```
fn classify_grade(score: u32) -> char {
    if score >= 90 && score <= 100 {
        'A'
    } else if score >= 80 && score <= 89 {
        'B'
    } else if score >= 70 && score <= 79 {
        'C'
    } else if score >= 60 && score <= 69 {
        'D'
    } else {
        'F'
    }
}
```

Alternative Using `match` Statement

```
fn classify_grade(score: u32) -> char {
    match score {
        90..=100 => 'A',
        80..=89 => 'B',
        70..=79 => 'C',
        60..=69 => 'D',
        0..=59 => 'F',
```

```
        _ => 'X', // Invalid score
    }
}
```

Step 3: Write the Main Function

```
use std::io;

fn main() {
    // Prompt the user for a numeric score
    println!("Enter the numeric score (0 to 100):");
    let mut score_input = String::new();
    io::stdin()
        .read_line(&mut score_input)
        .expect("Failed to read input");
    let score: u32 = match score_input.trim().parse() {
        Ok(n) if n <= 100 => n,
        _ => {
            println!("Invalid score. Please enter a number between 0 and 100.");
            return;
        }
    };

    // Determine the letter grade
    let grade = classify_grade(score);

    // Display the result
    println!("The letter grade is '{}'.", grade);
}
```

Explanation:

- **Input Handling:**
  - We read the score and parse it as `u32`.
  - We check if the score is within the acceptable range (0 to 100).
- **Control Flow:**
  - We use either an `if-else` chain or a `match` statement to determine the grade.
- **Error Handling:**
  - We handle invalid inputs by displaying an error message and terminating the program gracefully.

Complete Program:

```
use std::io;

fn classify_grade(score: u32) -> char {
    match score {
        90..=100 => 'A',
        80..=89 => 'B',
        70..=79 => 'C',
```

```
        60..=69 => 'D',
        0..=59 => 'F',
        _ => 'X', // Invalid score, but we've already validated input
    }
}

fn main() {
    // Prompt the user for a numeric score
    println!("Enter the numeric score (0 to 100):");
    let mut score_input = String::new();
    io::stdin()
        .read_line(&mut score_input)
        .expect("Failed to read input");
    let score: u32 = match score_input.trim().parse() {
        Ok(n) if n <= 100 => n,
        _ => {
            println!("Invalid score. Please enter a number between 0 and 100.");
            return;
        }
    };

    // Determine the letter grade
    let grade = classify_grade(score);

    // Display the result
    println!("The letter grade is '{}'.", grade);
}
```

Chapter 3: Understanding Ownership and Borrowing

Challenge 1 Solution: The Magical Book Borrowing System

Let's tackle this challenge step by step, keeping in mind Rust's ownership rules and borrowing concepts.

Step 1: Define the `Book` Struct

```
#[derive(Debug)]
struct Book {
    title: String,
}
```

We derive `Debug` to allow easy printing of `Book` instances.

Step 2: Define the `Library` Struct

```
struct Library {
    books: Vec<Book>,        // Available books
    borrowed: Vec<Book>,     // Borrowed books
}
```

The library has two collections: one for available books and one for borrowed books.

Step 3: Implement Methods for `Library`

Constructor and `add_book` Method

```
impl Library {
    fn new() -> Self {
        Self {
            books: Vec::new(),
            borrowed: Vec::new(),
        }
    }

    fn add_book(&mut self, title: &str) {
        let book = Book {
            title: title.to_string(),
        };
        self.books.push(book);
    }
```

`add_book` adds a new book to the library's collection.

Borrowing a Book

```
fn borrow_book(&mut self, title: &str) {
    // Find the position of the book in the available books
    if let Some(pos) = self.books.iter().position(|b| b.title == title) {
        // Remove the book from available books
        let book = self.books.remove(pos);
        // Add the book to borrowed books
        self.borrowed.push(book);
        println!("You have borrowed '{}'", title);
    } else {
        println!("'{}' is not available to borrow", title);
    }
}
```

We search for the book in the `books` vector.

If found, we remove it (ownership moves from `books` to `book`), then push it to `borrowed`.

Returning a Book

```
fn return_book(&mut self, title: &str) {
    // Find the position of the book in the borrowed books
    if let Some(pos) = self.borrowed.iter().position(|b| b.title == title) {
        // Remove the book from borrowed books
        let book = self.borrowed.remove(pos);
        // Add the book back to available books
        self.books.push(book);
        println!("You have returned '{}'", title);
    } else {
        println!("'{}' was not borrowed from this library", title);
    }
}
}
```

Similar to `borrow_book`, but in reverse.

Step 4: Demonstrate the Borrowing System

```
fn main() {
    let mut library = Library::new();
    library.add_book("The Rust Programming Language");
    library.add_book("The Book of Magic");

    library.borrow_book("The Rust Programming Language"); // Success
    library.borrow_book("The Rust Programming Language"); // Book already borrowed
    library.return_book("The Rust Programming Language"); // Book returned
    library.borrow_book("The Rust Programming Language"); // Success
}
```

Output:

```
You have borrowed 'The Rust Programming Language'
'The Rust Programming Language' is not available to borrow
You have returned 'The Rust Programming Language'
You have borrowed 'The Rust Programming Language'
```

Full Solution Code:

```rust
#[derive(Debug)]
struct Book {
    title: String,
}

struct Library {
    books: Vec<Book>,        // Available books
    borrowed: Vec<Book>,     // Borrowed books
}

impl Library {
    fn new() -> Self {
        Self {
            books: Vec::new(),
            borrowed: Vec::new(),
        }
    }

    fn add_book(&mut self, title: &str) {
        let book = Book {
            title: title.to_string(),
        };
        self.books.push(book);
    }

    fn borrow_book(&mut self, title: &str) {
        if let Some(pos) = self.books.iter().position(|b| b.title == title) {
            let book = self.books.remove(pos);
            self.borrowed.push(book);
            println!("You have borrowed '{}'", title);
        } else {
            println!("'{}' is not available to borrow", title);
        }
    }

    fn return_book(&mut self, title: &str) {
        if let Some(pos) = self.borrowed.iter().position(|b| b.title == title) {
            let book = self.borrowed.remove(pos);
            self.books.push(book);
            println!("You have returned '{}'", title);
        } else {
            println!("'{}' was not borrowed from this library", title);
```

```
        }
    }
}

fn main() {
    let mut library = Library::new();
    library.add_book("The Rust Programming Language");
    library.add_book("The Book of Magic");

    library.borrow_book("The Rust Programming Language"); // Success
    library.borrow_book("The Rust Programming Language"); // Book already borrowed
    library.return_book("The Rust Programming Language"); // Book returned
    library.borrow_book("The Rust Programming Language"); // Success
}
```

Explanation:

- We use ownership transfer (move semantics) when moving books between `books` and `borrowed`.
- The `remove` method moves the book out of the vector, transferring ownership.
- This ensures that a book can't be in both `books` and `borrowed` at the same time.
- Rust's ownership rules prevent us from accidentally having multiple mutable references to the same `Book` instance.

Challenge 2 Solution: Finding the Longest Word

Let's write a function that takes a string slice and returns a string slice corresponding to the longest word.

Step 1: Writing the `find_longest_word` Function

```
fn find_longest_word(s: &str) -> &str {
    let mut longest = "";
    let words = s.split_whitespace();

    for word in words {
        if word.len() > longest.len() {
            longest = word;
        }
    }

    longest
}
```

- We split the input string slice `s` into words using `split_whitespace()`, which returns an iterator over `&str` slices.
- We iterate over each word, comparing its length to the current longest.
- Since we're working with string slices, we don't take ownership of the original string.

Step 2: Handling Lifetimes

Rust needs to ensure that the returned string slice is valid. By default, the lifetime of the returned `&str` is tied to the input `&str`.

Our function signature:

```
fn find_longest_word(s: &str) -> &str
```

The lifetime of the output `&str` is implicitly tied to the input `&str`.

Step 3: Demonstrating the Function

```
fn main() {
    let sentence = String::from("The quick brown fox jumps over the lazy dog");
    let longest = find_longest_word(&sentence);
    println!("The longest word is '{}'", longest); // Outputs: 'jumps'

    let sentence2 = "A marvelous night for a moondance";
    let longest2 = find_longest_word(sentence2);
    println!("The longest word is '{}'", longest2); // Outputs: 'marvelous'
}
```

Output:

```
The longest word is 'jumps'
The longest word is 'marvelous'
```

Full Solution Code:

```
fn find_longest_word(s: &str) -> &str {
    let mut longest = "";
    let words = s.split_whitespace();

    for word in words {
        if word.len() > longest.len() {
            longest = word;
        }
    }

    longest
}

fn main() {
    let sentence = String::from("The quick brown fox jumps over the lazy dog");
    let longest = find_longest_word(&sentence);
    println!("The longest word is '{}'", longest);

    let sentence2 = "A marvelous night for a moondance";
    let longest2 = find_longest_word(sentence2);
    println!("The longest word is '{}'", longest2);
}
```

Explanation:

- We efficiently find the longest word without taking ownership of the original string.
- The function works with string slices, which are references to parts of a string.
- Lifetimes are managed implicitly because the output lifetime is tied to the input lifetime.

Challenge 3 Solution: Stack vs. Heap - The Number Mystery

Let's explore stack and heap allocation by creating numbers stored in different ways.

Step 1: Implementing `create_stack_number`

```
fn create_stack_number() -> i32 {
    let x = 42; // Stored on the stack
    x
}
```

Simple function that returns an `i32` stored on the stack.

Step 2: Implementing `create_heap_number`

```
fn create_heap_number() -> Box<i32> {
    let x = Box::new(72); // `Box` allocates the `i32` on the heap
    x
}
```

`Box::new` allocates the value on the heap and returns a `Box<i32>` pointing to it.

Step 3: Demonstrating in `main`

```
fn main() {
    let stack_num = create_stack_number();
    let heap_num = create_heap_number();

    println!("Stack number: {}", stack_num);
    println!("Heap number: {}", heap_num);

    // Move ownership of heap number
    let moved_heap_num = heap_num; // `heap_num` is now invalid
    // println!("Heap number: {}", heap_num); // This would cause a compile-time error

    println!("Moved heap number: {}", moved_heap_num);

    // Demonstrate that moving `i32` does not invalidate the original
    let moved_stack_num = stack_num; // Copy occurs for `i32`
    println!("Original stack number: {}", stack_num);
    println!("Moved stack number: {}", moved_stack_num);
}
```

Output:

```
Stack number: 42
Heap number: 72
Moved heap number: 72
Original stack number: 42
Moved stack number: 42
```

Explanation of Ownership Transfer

- When we assign `heap_num` to `moved_heap_num`, ownership moves, and `heap_num` becomes invalid.
  - Attempting to use `heap_num` afterward results in a compile-time error.
- For `stack_num`, which is an `i32`, the `Copy` trait is implemented, so the value is copied, and both `stack_num` and `moved_stack_num` can be used.

Full Code:

```rust
fn create_stack_number() -> i32 {
    let x = 42; // Stored on the stack
    x
}

fn create_heap_number() -> Box<i32> {
    let x = Box::new(72); // Allocated on the heap
    x
}

fn main() {
    let stack_num = create_stack_number();
    let heap_num = create_heap_number();

    println!("Stack number: {}", stack_num);
    println!("Heap number: {}", heap_num);

    // Move ownership of heap number
    let moved_heap_num = heap_num; // heap_num is now invalid
    // println!("Heap number: {}", heap_num); // Uncommenting this line will cause an error

    println!("Moved heap number: {}", moved_heap_num);

    // Move stack number
    let moved_stack_num = stack_num; // Copies the value
    println!("Original stack number: {}", stack_num);
    println!("Moved stack number: {}", moved_stack_num);
}
```

Explanation:

- Stack allocation is used for values with a known, fixed size at compile time.
- Heap allocation is used for data that can change size or needs to outlive the current scope.
- `Box<T>` is a smart pointer that allocates memory on the heap and provides ownership of that memory.
- Moving a `Box<T>` transfers ownership of the heap data.
- Simple types like `i32` are `Copy`, so assigning them creates a copy rather than moving ownership.

Chapter 4: Structs and Enums

Challenge 1 Solution: Shape Area Calculator with Enums

Step 1: Define the `Shape` Enum

```
enum Shape {
    Circle { radius: f64 },
    Rectangle { width: f64, height: f64 },
    Triangle { base: f64, height: f64 },
}
```

The enum `Shape` has variants for `Circle`, `Rectangle`, and `Triangle`, each with associated data.

Step 2: Implement the `calculate_area` Function

```
fn calculate_area(shape: &Shape) -> f64 {
    match shape {
        Shape::Circle { radius } => std::f64::consts::PI * radius * radius,
        Shape::Rectangle { width, height } => width * height,
        Shape::Triangle { base, height } => 0.5 * base * height,
    }
}
```

The function uses pattern matching to compute the area based on the shape variant.

Step 3: Demonstrate Usage

```
fn main() {
    let circle = Shape::Circle { radius: 5.0 };
    let rectangle = Shape::Rectangle { width: 4.0, height: 6.0 };
    let triangle = Shape::Triangle { base: 3.0, height: 7.0 };

    let circle_area = calculate_area(&circle);
    let rectangle_area = calculate_area(&rectangle);
    let triangle_area = calculate_area(&triangle);

    println!("Area of the circle: {:.2}", circle_area);
    println!("Area of the rectangle: {:.2}", rectangle_area);
    println!("Area of the triangle: {:.2}", triangle_area);
}
```

Instances of each shape are created, and their areas are calculated and printed.

Complete Code

```
enum Shape {
```

```rust
    Circle { radius: f64 },
    Rectangle { width: f64, height: f64 },
    Triangle { base: f64, height: f64 },
}

fn calculate_area(shape: &Shape) -> f64 {
    match shape {
        Shape::Circle { radius } => std::f64::consts::PI * radius * radius,
        Shape::Rectangle { width, height } => width * height,
        Shape::Triangle { base, height } => 0.5 * base * height,
    }
}

fn main() {
    let circle = Shape::Circle { radius: 5.0 };
    let rectangle = Shape::Rectangle { width: 4.0, height: 6.0 };
    let triangle = Shape::Triangle { base: 3.0, height: 7.0 };

    let circle_area = calculate_area(&circle);
    let rectangle_area = calculate_area(&rectangle);
    let triangle_area = calculate_area(&triangle);

    println!("Area of the circle: {:.2}", circle_area);
    println!("Area of the rectangle: {:.2}", rectangle_area);
    println!("Area of the triangle: {:.2}", triangle_area);
}
```

Sample Output:

```
Area of the circle: 78.54
Area of the rectangle: 24.00
Area of the triangle: 10.50
```

Explanation:

This solution demonstrates the use of enums to represent different shapes and pattern matching to compute their areas in a clean and straightforward manner.

Challenge 2 Solution: Temperature Converter with Structs and Methods

Step 1: Define the `Temperature` Struct

```rust
struct Temperature {
    value: f64,
    is_celsius: bool,
}
```

The `Temperature` struct holds the temperature value and a flag indicating whether it's in Celsius.

Step 2: Implement Methods for Conversion

```rust
impl Temperature {
    fn new_celsius(value: f64) -> Self {
        Self {
            value,
            is_celsius: true,
        }
    }

    fn new_fahrenheit(value: f64) -> Self {
        Self {
            value,
            is_celsius: false,
        }
    }

    fn to_celsius(&self) -> Temperature {
        if self.is_celsius {
            Temperature {
                value: self.value,
                is_celsius: true,
            }
        } else {
            Temperature {
                value: (self.value - 32.0) * 5.0 / 9.0,
                is_celsius: true,
            }
        }
    }

    fn to_fahrenheit(&self) -> Temperature {
        if self.is_celsius {
            Temperature {
                value: self.value * 9.0 / 5.0 + 32.0,
                is_celsius: false,
            }
        } else {
            Temperature {
                value: self.value,
                is_celsius: false,
            }
        }
    }
}
```

- `new_celsius` and `new_fahrenheit` are constructors for creating temperatures in specific units.
- `to_celsius` and `to_fahrenheit` convert the temperature to the desired unit.

Step 3: Demonstrate Usage

```
fn main() {
    let temp_c = Temperature::new_celsius(25.0);
    let temp_f = temp_c.to_fahrenheit();

    println!(
        "{}°C is equal to {:.2}°F",
        temp_c.value, temp_f.value
    );

    let temp_f2 = Temperature::new_fahrenheit(77.0);
    let temp_c2 = temp_f2.to_celsius();

    println!(
        "{}°F is equal to {:.2}°C",
        temp_f2.value, temp_c2.value
    );
}
```

The program converts temperatures from Celsius to Fahrenheit and vice versa, printing the results.

Complete Code

```
struct Temperature {
    value: f64,
    is_celsius: bool,
}

impl Temperature {
    fn new_celsius(value: f64) -> Self {
        Self {
            value,
            is_celsius: true,
        }
    }

    fn new_fahrenheit(value: f64) -> Self {
        Self {
            value,
            is_celsius: false,
        }
    }

    fn to_celsius(&self) -> Temperature {
        if self.is_celsius {
            Temperature {
                value: self.value,
                is_celsius: true,
            }
        } else {
            Temperature {
```

```
                value: (self.value - 32.0) * 5.0 / 9.0,
                is_celsius: true,
            }
        }
    }

    fn to_fahrenheit(&self) -> Temperature {
        if self.is_celsius {
            Temperature {
                value: self.value * 9.0 / 5.0 + 32.0,
                is_celsius: false,
            }
        } else {
            Temperature {
                value: self.value,
                is_celsius: false,
            }
        }
    }
}

fn main() {
    let temp_c = Temperature::new_celsius(25.0);
    let temp_f = temp_c.to_fahrenheit();

    println!(
        "{}°C is equal to {:.2}°F",
        temp_c.value, temp_f.value
    );

    let temp_f2 = Temperature::new_fahrenheit(77.0);
    let temp_c2 = temp_f2.to_celsius();

    println!(
        "{}°F is equal to {:.2}°C",
        temp_f2.value, temp_c2.value
    );
}
```

Output:

```
25°C is equal to 77.00°F
77°F is equal to 25.00°C
```

Explanation:

This solution uses a struct to encapsulate temperature values and methods to perform conversions, demonstrating the use of structs and methods in a simple context.

Challenge 3 Solution: Simple Calculator with Error Handling

Step 1: Handle User Input

```rust
use std::io;

fn main() {
    let mut input1 = String::new();
    let mut input2 = String::new();
    let mut operator = String::new();

    println!("Enter the first number:");
    io::stdin()
        .read_line(&mut input1)
        .expect("Failed to read input");

    println!("Enter an operator (+, -, *, /):");
    io::stdin()
        .read_line(&mut operator)
        .expect("Failed to read input");

    println!("Enter the second number:");
    io::stdin()
        .read_line(&mut input2)
        .expect("Failed to read input");

    let num1: f64 = match input1.trim().parse() {
        Ok(n) => n,
        Err(_) => {
            println!("Invalid input for the first number.");
            return;
        }
    };

    let num2: f64 = match input2.trim().parse() {
        Ok(n) => n,
        Err(_) => {
            println!("Invalid input for the second number.");
            return;
        }
    };

    let operator = operator.trim().chars().next().unwrap_or(' ');

    // Proceed to calculation
}
```

User inputs are read and validated.

Step 2: Implement the `calculate` Function

```
fn calculate(a: f64, b: f64, operator: char) -> Result<f64, String> {
    match operator {
        '+' => Ok(a + b),
        '-' => Ok(a - b),
        '*' => Ok(a * b),
        '/' => {
            if b == 0.0 {
                Err("Error: Division by zero".to_string())
            } else {
                Ok(a / b)
            }
        }
        _ => Err("Error: Invalid operator".to_string()),
    }
}
```

The function matches the operator and performs the calculation or returns an error.

Step 3: Perform the Calculation and Handle the Result

```
match calculate(num1, num2, operator) {
    Ok(result) => println!("Result: {}", result),
    Err(e) => println!("{}", e),
}
```

The `calculate` function is called, and the result is handled using pattern matching.

Complete Code

```
use std::io;

fn calculate(a: f64, b: f64, operator: char) -> Result<f64, String> {
    match operator {
        '+' => Ok(a + b),
        '-' => Ok(a - b),
        '*' => Ok(a * b),
        '/' => {
            if b == 0.0 {
                Err("Error: Division by zero".to_string())
            } else {
                Ok(a / b)
            }
        }
        _ => Err("Error: Invalid operator".to_string()),
    }
}

fn main() {
```

```rust
    let mut input1 = String::new();
    let mut input2 = String::new();
    let mut operator = String::new();

    println!("Enter the first number:");
    io::stdin()
        .read_line(&mut input1)
        .expect("Failed to read input");

    println!("Enter an operator (+, -, *, /):");
    io::stdin()
        .read_line(&mut operator)
        .expect("Failed to read input");

    println!("Enter the second number:");
    io::stdin()
        .read_line(&mut input2)
        .expect("Failed to read input");

    let num1: f64 = match input1.trim().parse() {
        Ok(n) => n,
        Err(_) => {
            println!("Invalid input for the first number.");
            return;
        }
    };

    let num2: f64 = match input2.trim().parse() {
        Ok(n) => n,
        Err(_) => {
            println!("Invalid input for the second number.");
            return;
        }
    };

    let operator = operator.trim().chars().next().unwrap_or(' ');

    match calculate(num1, num2, operator) {
        Ok(result) => println!("Result: {}", result),
        Err(e) => println!("{}", e),
    }
}
```

This solution provides a simple calculator that demonstrates error handling using the `Result` type, ensuring the program handles errors gracefully without panicking.

Chapter 5: Collections

Challenge 1 Solution: The Great Vector Shuffle

Let's tackle the challenge step by step.

Step 1: Create a Deck of Cards

```
let mut deck: Vec<u8> = (1..=52).collect();
```

We use a range to collect numbers from 1 to 52 into a vector.

Step 2: Remove All Even-Numbered Cards

```
deck.retain(|&card| card % 2 != 0);
```

`retain` is used to keep only the odd-numbered cards.

Step 3: Insert the Joker Card at the Beginning and End

```
deck.insert(0, 0); // Insert Joker at the beginning
deck.push(0);      // Insert Joker at the end
```

Step 4: Shuffle the Deck Randomly

```
use rand::seq::SliceRandom;
use rand::thread_rng;

deck.shuffle(&mut thread_rng());
```

We use the `rand` crate for shuffling.

Remember to add `rand = "0.X"` (the latest version) to your `Cargo.toml` dependencies.

Step 5: Print the First Five Cards

```
println!("First five cards: {:?}", &deck[..5]);
```

Complete Solution:

```
use rand::seq::SliceRandom;
use rand::thread_rng;

fn main() {
    // Step 1: Create a deck of cards numbered from 1 to 52
    let mut deck: Vec<u8> = (1..=52).collect();

    // Step 2: Remove all even-numbered cards
    deck.retain(|&card| card % 2 != 0);
```

```
    // Step 3: Insert the Joker card (0) at the beginning and end
    deck.insert(0, 0); // Joker at the beginning
    deck.push(0);      // Joker at the end

    // Step 4: Shuffle the deck
    deck.shuffle(&mut thread_rng());

    // Step 5: Print the first five cards
    println!("First five cards: {:?}", &deck[..5]);
}
```

Output:

First five cards: [7, 0, 31, 45, 21]

Challenge 2 Solution: Secret Message with Strings

Step 1: Define the Garbled Text

```
let garbled = "ThiiXs iasXa tseXcrt mXessaXge!";
```

Step 2: Extract Every Third Character

We'll iterate over the characters and collect every third one.

```
let secret_chars: Vec<char> = garbled.chars()
    .enumerate()
    .filter(|&(i, _)| i % 3 == 0)
    .map(|(_, c)| c)
    .collect();
```

Step 3: Build the Secret Message String

```
let secret_message: String = secret_chars.into_iter().collect();
```

Step 4: Print the Secret Message

```
println!("Secret message: {}", secret_message);
```

Complete Solution:

```
fn main() {
    let garbled = "ThiiXs iasXa tseXcrt mXessaXge!";

    // Step 2: Extract every third character
    let secret_chars: Vec<char> = garbled.chars()
```

```rust
        .enumerate()
        .filter(|&(i, _)| i % 3 == 0)
        .map(|(_, c)| c)
        .collect();

    // Step 3: Build the secret message
    let secret_message: String = secret_chars.into_iter().collect();

    // Step 4: Print the secret message
    println!("Secret message: {}", secret_message);
}
```

Output:

Secret message: This is a secret message!

Challenge 3 Solution: Inventory Management with Hash Maps

Step 1: Initialize the Inventory Hash Map

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

fn main() {
    let mut inventory = HashMap::new();

    // ... rest of the code
}
```

Step 2: Add Initial Items to the Inventory

```rust
inventory.insert("Apple".to_string(), 10);
inventory.insert("Banana".to_string(), 5);
inventory.insert("Orange".to_string(), 8);
```

Step 3: Update Inventory with Shipment

Use `entry().and_modify().or_insert()` to update quantities.

```rust
let shipment = vec![
    ("Apple".to_string(), 5),
    ("Banana".to_string(), 2),
    ("Grapes".to_string(), 15),
];

for (item, qty) in shipment {
    inventory.entry(item)
        .and_modify(|e| *e += qty)
        .or_insert(qty);
```

```
}
```

Step 4: Customer Purchase

We need to handle cases where the stock is insufficient.

```
let purchase = vec![
    ("Apple".to_string(), 4),
    ("Banana".to_string(), 2),
    ("Grapes".to_string(), 10),
];

for (item, qty) in purchase {
    match inventory.get_mut(&item) {
        Some(stock) => {
            if *stock >= qty {
                *stock -= qty;
                println!("Customer bought {} {}", qty, item);
            } else {
                println!("Insufficient stock for {}", item);
            }
        }
        None => println!("Item {} does not exist in inventory", item),
    }
}
```

Step 5: Print the Updated Inventory

```
println!("\nUpdated Inventory:");
for (item, qty) in &inventory {
    println!("- {}: {}", item, qty);
}
```

Complete Solution:

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

fn main() {
    let mut inventory = HashMap::new();

    // Step 2: Add initial items
    inventory.insert("Apple".to_string(), 10);
    inventory.insert("Banana".to_string(), 5);
    inventory.insert("Orange".to_string(), 8);

    // Step 3: Update inventory with shipment
    let shipment = vec![
        ("Apple".to_string(), 5),
        ("Banana".to_string(), 2),
        ("Grapes".to_string(), 15),
```

```
    ];

    for (item, qty) in shipment {
        inventory.entry(item)
            .and_modify(|e| *e += qty)
            .or_insert(qty);
    }

    // Step 4: Customer purchase
    let purchase = vec![
        ("Apple".to_string(), 4),
        ("Banana".to_string(), 2),
        ("Grapes".to_string(), 10),
    ];

    for (item, qty) in purchase {
        match inventory.get_mut(&item) {
            Some(stock) => {
                if *stock >= qty {
                    *stock -= qty;
                    println!("Customer bought {} {}", qty, item);
                } else {
                    println!("Insufficient stock for {}", item);
                }
            }
            None => println!("Item {} does not exist in inventory", item),
        }
    }

    // Step 5: Print the updated inventory
    println!("\nUpdated Inventory:");
    for (item, qty) in &inventory {
        println!("- {}: {}", item, qty);
    }
}
```

Output:

Customer bought 4 Apple

Customer bought 2 Banana

Customer bought 10 Grapes

Updated Inventory:

- Apple: 11

- Banana: 5

- Orange: 8

- Grapes: 5

Chapter 6: Generic Types, Traits, and Lifetimes

Challenge 1 Solution: Generic Statistics Calculator

Step 1: Define Trait Bounds

We need to ensure that our functions can work with numeric types that support necessary operations like addition, division, and ordering. We'll use standard traits like `Add`, `Div`, `Ord`, and `Copy`.

Step 2: Implement `calculate_mean` Function

```
use std::ops::Add;
use std::ops::Div;
use std::fmt::Display;

fn calculate_mean<T>(data: &[T]) -> f64
where
    T: Add<Output = T> + Div<Output = T> + Copy + Into<f64> + From<u8>,
{
    let sum: T = data.iter().cloned().fold(T::from(0u8), |a, b| a + b);
    let count = T::from(data.len() as u8);
    let mean = sum / count;
    mean.into()
}
```

Explanation:

- **Trait Bounds:**
  - `Add<Output = T>`: Allows addition.
  - `Div<Output = T>`: Allows division.
  - `Copy`: Allows copying values.
  - `Into<f64>`: Allows conversion into `f64` for returning a floating-point mean.
  - `From<u8>`: Allows creating a `T` from `u8` (used for initializing sum and count).
- **Logic:**
  - Sum all the elements.
  - Divide the sum by the count.
  - Convert the result into `f64`.

Step 3: Implement `calculate_median` Function

```
fn calculate_median<T>(data: &[T]) -> f64
where
    T: Copy + Ord + Into<f64>,
{
    let mut sorted_data = data.to_vec();
    sorted_data.sort();
```

```rust
    let mid = sorted_data.len() / 2;

    if sorted_data.len() % 2 == 0 {
        let a = sorted_data[mid - 1];
        let b = sorted_data[mid];
        ((a.into() + b.into()) / 2.0)
    } else {
        sorted_data[mid].into()
    }
}
```

Explanation:

- **Trait Bounds:**
  - `Copy`: Allows copying values.
  - `Ord`: Allows ordering for sorting.
  - `Into<f64>`: Allows conversion into `f64`.
- **Logic:**
  - Sort the data.
  - If the length is even, average the two middle values.
  - If the length is odd, return the middle value.

Step 4: Implement `calculate_mode` Function

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

fn calculate_mode<T>(data: &[T]) -> Vec<T>
where
    T: Copy + Eq + std::hash::Hash,
{
    let mut occurrences = HashMap::new();
    for &value in data {
        *occurrences.entry(value).or_insert(0) += 1;
    }

    let max_count = occurrences.values().cloned().max().unwrap_or(0);

    occurrences
        .into_iter()
        .filter(|&(_k, v)| v == max_count)
        .map(|(k, _v)| k)
        .collect()
}
```

Explanation:

- **Trait Bounds:**
  - `Copy`: Allows copying values.
  - `Eq` and `Hash`: Required for keys in `HashMap`.
- **Logic:**
  - Count the occurrences of each value.

- Find the maximum occurrence count.
- Collect all values that have the maximum count.

Complete Code

```rust
use std::ops::{Add, Div};
use std::fmt::Display;
use std::collections::HashMap;

fn calculate_mean<T>(data: &[T]) -> f64
where
    T: Add<Output = T> + Div<Output = T> + Copy + Into<f64> + From<u8>,
{
    let sum: T = data.iter().cloned().fold(T::from(0u8), |a, b| a + b);
    let count = T::from(data.len() as u8);
    let mean = sum / count;
    mean.into()
}

fn calculate_median<T>(data: &[T]) -> f64
where
    T: Copy + Ord + Into<f64>,
{
    let mut sorted_data = data.to_vec();
    sorted_data.sort();

    let mid = sorted_data.len() / 2;

    if sorted_data.len() % 2 == 0 {
        let a = sorted_data[mid - 1];
        let b = sorted_data[mid];
        ((a.into() + b.into()) / 2.0)
    } else {
        sorted_data[mid].into()
    }
}

fn calculate_mode<T>(data: &[T]) -> Vec<T>
where
    T: Copy + Eq + std::hash::Hash,
{
    let mut occurrences = HashMap::new();
    for &value in data {
        *occurrences.entry(value).or_insert(0) += 1;
    }

    let max_count = occurrences.values().cloned().max().unwrap_or(0);

    occurrences
        .into_iter()
```

```
        .filter(|&(_k, v)| v == max_count)
        .map(|(k, _v)| k)
        .collect()
}

fn main() {
    let data_int = vec![1, 2, 2, 3, 4];
    let data_float = vec![1.5, 2.5, 3.5, 4.5];

    let mean_int = calculate_mean(&data_int);
    let median_int = calculate_median(&data_int);
    let mode_int = calculate_mode(&data_int);

    println!("Integer Data: {:?}", data_int);
    println!("Mean: {}", mean_int);
    println!("Median: {}", median_int);
    println!("Mode: {:?}", mode_int);

    let mean_float = calculate_mean(&data_float);
    let median_float = calculate_median(&data_float);

    println!("\nFloat Data: {:?}", data_float);
    println!("Mean: {}", mean_float);
    println!("Median: {}", median_float);
}
```

Output

```
Integer Data: [1, 2, 2, 3, 4]
Mean: 2.4
Median: 2.0
Mode: [2]

Float Data: [1.5, 2.5, 3.5, 4.5]
Mean: 3.0
Median: 3.0
```

Explanation

- **Trait Bounds and Generics:** We used trait bounds to ensure the functions work with different numeric types.
- **Ownership and Borrowing:** The functions take slices (`&[T]`), borrowing the data without taking ownership.
- **Using Lifetimes:** Lifetimes are not explicitly required here since we're working with borrowed data and returning owned data (`f64`).

Challenge 2 Solution: Custom String Formatter Trait

Step 1: Define the `StringFormatter` Trait

```
trait StringFormatter {
    fn to_uppercase(&self) -> String;
```

```rust
    fn to_lowercase(&self) -> String;
    fn to_title_case(&self) -> String;
}
```

Step 2: Implement `StringFormatter` for `&str`

```rust
impl StringFormatter for &str {
    fn to_uppercase(&self) -> String {
        self.to_uppercase()
    }

    fn to_lowercase(&self) -> String {
        self.to_lowercase()
    }

    fn to_title_case(&self) -> String {
        self.split_whitespace()
            .map(|word| {
                let mut chars = word.chars();
                match chars.next() {
                    Some(first) => first.to_uppercase().collect::<String>() + chars.as_str(),
                    None => String::new(),
                }
            })
            .collect::<Vec<_>>()
            .join(" ")
    }
}
```

Step 3: Implement `StringFormatter` for `String`

```rust
impl StringFormatter for String {
    fn to_uppercase(&self) -> String {
        self.as_str().to_uppercase()
    }

    fn to_lowercase(&self) -> String {
        self.as_str().to_lowercase()
    }

    fn to_title_case(&self) -> String {
        self.as_str().to_title_case()
    }
}
```

Step 4: Write the Generic Function `format_and_print`

```rust
fn format_and_print<T>(text: T)
```

```rust
where
    T: StringFormatter + Display,
{
    println!("Original: {}", text);
    println!("Uppercase: {}", text.to_uppercase());
    println!("Lowercase: {}", text.to_lowercase());
    println!("Title Case: {}", text.to_title_case());
}
```

Complete Code

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

trait StringFormatter {
    fn to_uppercase(&self) -> String;
    fn to_lowercase(&self) -> String;
    fn to_title_case(&self) -> String;
}

impl StringFormatter for &str {
    fn to_uppercase(&self) -> String {
        self.to_uppercase()
    }

    fn to_lowercase(&self) -> String {
        self.to_lowercase()
    }

    fn to_title_case(&self) -> String {
        self.split_whitespace()
            .map(|word| {
                let mut chars = word.chars();
                match chars.next() {
                    Some(first) => first.to_uppercase().collect::<String>() + chars.as_str(),
                    None => String::new(),
                }
            })
            .collect::<Vec<_>>()
            .join(" ")
    }
}

impl StringFormatter for String {
    fn to_uppercase(&self) -> String {
        self.as_str().to_uppercase()
    }

    fn to_lowercase(&self) -> String {
        self.as_str().to_lowercase()
    }
```

```
    fn to_title_case(&self) -> String {
        self.as_str().to_title_case()
    }
}

fn format_and_print<T>(text: T)
where
    T: StringFormatter + Display,
{
    println!("Original: {}", text);
    println!("Uppercase: {}", text.to_uppercase());
    println!("Lowercase: {}", text.to_lowercase());
    println!("Title Case: {}", text.to_title_case());
}

fn main() {
    let text = "rust programming language";

    format_and_print(text);

    let text_string = String::from("another example STRING");

    format_and_print(text_string);
}
```

Output

```
Original: rust programming language
Uppercase: RUST PROGRAMMING LANGUAGE
Lowercase: rust programming language
Title Case: Rust Programming Language
Original: another example STRING
Uppercase: ANOTHER EXAMPLE STRING
Lowercase: another example string
Title Case: Another Example STRING
```

Explanation

- **Custom Trait:** `StringFormatter` defines methods for string formatting.
- **Trait Implementation:** Implemented the trait for both `&str` and `String`.
- **Generic Function:** `format_and_print` works with any type implementing `StringFormatter` and `Display`.
- **Borrowing and References:** Used `as_str()` when necessary to convert `String` to `&str`.

Challenge 3 Solution: Lifetime Management in Text Processing

Step 1: Define the Struct `WordExtractor<'a>`

```
struct WordExtractor<'a> {
    text: &'a str,
}
```

Step 2: Implement Methods for `WordExtractor<'a>`

```
impl<'a> WordExtractor<'a> {
    fn new(text: &'a str) -> Self {
        WordExtractor { text }
    }

    fn extract_words(&self, start_char: char) -> Vec<&'a str> {
        self.text
            .split_whitespace()
            .filter(|word| word.starts_with(start_char))
            .collect()
    }
}
```

Complete Code

```
struct WordExtractor<'a> {
    text: &'a str,
}

impl<'a> WordExtractor<'a> {
    fn new(text: &'a str) -> Self {
        WordExtractor { text }
    }

    fn extract_words(&self, start_char: char) -> Vec<&'a str> {
        self.text
            .split_whitespace()
            .filter(|word| word.starts_with(start_char))
            .collect()
    }
}

fn main() {
    let text = String::from("Rust is remarkable and reliable for robust systems");

    let extractor = WordExtractor::new(&text);
    let words = extractor.extract_words('r');

    println!("Words starting with 'r': {:?}", words);
}
```

Output

```
Words starting with 'r': ["Rust", "remarkable", "reliable", "robust"]
```

Explanation

- **Lifetimes and References:** The lifetime `'a` ensures that the references to words are valid as long as `text` is.

- **Ownership and Borrowing:** `WordExtractor` borrows the text; it does not take ownership.
- **Method Logic:** `extract_words` filters words starting with the specified character.
- **No Need for Additional Lifetime Annotations:** The compiler can infer lifetimes in the `extract_words` method due to lifetime elision rules.

Chapter 7: Smart Pointers

Challenge 1 Solution: Implementing a Recursive Data Structure with Box<T>

Step 1: Define the BinaryTree Enum

```
enum BinaryTree {
    Leaf,
    Node(i32, Box<BinaryTree>, Box<BinaryTree>),
}
```

Step 2: Implement Methods

Insert Method

```
impl BinaryTree {
    fn insert(self, value: i32) -> Self {
        match self {
            BinaryTree::Leaf => {
                BinaryTree::Node(value, Box::new(BinaryTree::Leaf),
Box::new(BinaryTree::Leaf))
            }
            BinaryTree::Node(current_value, left, right) => {
                if value < current_value {
                    BinaryTree::Node(
                        current_value,
                        Box::new(left.insert(value)),
                        right,
                    )
                } else {
                    BinaryTree::Node(
                        current_value,
                        left,
                        Box::new(right.insert(value)),
                    )
                }
            }
        }
    }
}
```

Search Method

```
impl BinaryTree {
```

```
    fn contains(&self, value: i32) -> bool {
        match self {
            BinaryTree::Leaf => false,
            BinaryTree::Node(current_value, left, right) => {
                if *current_value == value {
                    true
                } else if value < *current_value {
                    left.contains(value)
                } else {
                    right.contains(value)
                }
            }
        }
    }
}
```

Step 3: Demonstrate Usage

```
fn main() {
    let tree = BinaryTree::Leaf;
    let tree = tree.insert(10);
    let tree = tree.insert(5);
    let tree = tree.insert(15);
    let tree = tree.insert(8);

    println!("Tree contains 8: {}", tree.contains(8)); // true
    println!("Tree contains 2: {}", tree.contains(2)); // false
}
```

Explanation

- **Recursive Data Type:** `BinaryTree` is recursively defined; each `Node` can contain left and right `BinaryTree` instances.
- **Heap Allocation:** Using `Box<T>` allows us to allocate nodes on the heap, enabling the recursive structure.
- **Ownership:** Each node owns its children through `Box<T>`, ensuring proper memory management.

Challenge 2 Solution: Managing Shared Ownership with Rc<T>

Step 1: Define the GraphNode Struct

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

struct GraphNode<T> {
    value: T,
    neighbors: Vec<Rc<GraphNode<T>>>,
}
```

Step 2: Implement Methods

Constructor

```rust
impl<T> GraphNode<T> {
    fn new(value: T) -> Rc<Self> {
        Rc::new(GraphNode {
            value,
            neighbors: Vec::new(),
        })
    }
}
```

Add Edge Method

```rust
impl<T> GraphNode<T> {
    fn add_edge(node: &Rc<Self>, neighbor: Rc<GraphNode<T>>) {
        node.neighbors.push(neighbor);
    }
}
```

Step 3: Demonstrate Usage

```rust
fn main() {
    let node1 = GraphNode::new(1);
    let node2 = GraphNode::new(2);
    let node3 = GraphNode::new(3);

    GraphNode::add_edge(&node1, node2.clone());
    GraphNode::add_edge(&node2, node3.clone());
    GraphNode::add_edge(&node3, node1.clone()); // Creating a cycle

    println!("Node 1 neighbors: {:?}", node1.neighbors.iter().map(|n|
n.value).collect::<Vec<_>>());
    println!("Node 2 neighbors: {:?}", node2.neighbors.iter().map(|n|
n.value).collect::<Vec<_>>());
    println!("Node 3 neighbors: {:?}", node3.neighbors.iter().map(|n|
n.value).collect::<Vec<_>>());
}
```

Note: **This simple implementation doesn't handle reference cycles which can lead to memory leaks.**

Explanation

- **Shared Ownership:** `Rc<T>` allows multiple nodes to share ownership of their neighbors.
- **Cloning Rc Pointers:** We clone `Rc<T>` pointers to add neighbors without transferring ownership.
- **Potential Memory Leaks:** Since we're creating cycles, we need to be cautious of memory leaks.

Challenge 3 Solution: Combining Rc<T> and RefCell<T> for Mutable Shared Data

Step 1: Define the Config Struct

```
use std::cell::RefCell;
use std::rc::Rc;

#[derive(Debug)]
struct Config {
    theme: String,
    volume: u8,
}
```

Step 2: Create Widgets that Share Config

```
struct Widget {
    config: Rc<RefCell<Config>>,
    name: String,
}

impl Widget {
    fn new(name: &str, config: Rc<RefCell<Config>>) -> Self {
        Widget {
            name: name.to_string(),
            config,
        }
    }

    fn update_theme(&self, new_theme: &str) {
        self.config.borrow_mut().theme = new_theme.to_string();
        println!("{} updated theme to {}", self.name, new_theme);
    }

    fn show_config(&self) {
        println!("{} config: {:?}", self.name, self.config.borrow());
    }
}
```

Step 3: Demonstrate Usage

```
fn main() {
    let config = Rc::new(RefCell::new(Config {
        theme: "Light".to_string(),
        volume: 50,
    }));
```

```
    let widget1 = Widget::new("Widget1", Rc::clone(&config));
    let widget2 = Widget::new("Widget2", Rc::clone(&config));

    widget1.show_config();
    widget2.show_config();

    widget1.update_theme("Dark");

    widget1.show_config();
    widget2.show_config();
}
```

Explanation

- **Rc<RefCell<T>>:** Allows multiple owners (`Rc<T>`) and interior mutability (`RefCell<T>`).
- **Borrow Checking at Runtime:** `RefCell<T>` enforces borrowing rules at runtime.
- **Shared Mutable Data:** Widgets can read and modify the shared `Config`, and changes are reflected across all instances.