Rust Refresher for C/C++ Developers

This guide focuses on Rust's idiomatic patterns that differ from C/C++. Each section demonstrates key concepts with complete working examples.

1. Functions, Arrays, and Tuples

Key Differences from C/C++

- Functions are expressions (return values without `return`)
- Arrays have compile-time known sizes `[T; N]`
- Tuples provide heterogeneous data grouping
- Pattern matching for destructuring

Function Basics

```rust
fn main() {
    println!("{}", add_numbers(5, 3));
    println!("{}", format_name("John", "Doe"));

    let coords = create_point(10, 20);
    println!("Point: ({}, {})", coords.0, coords.1);

    // Destructuring tuple
    let (x, y) = coords;
    println!("x: {}, y: {}", x, y);
}

// Expression-based function (no semicolon on last line)
fn add_numbers(a: i32, b: i32) -> i32 {
    a + b
}

// String formatting
fn format_name(first: &str, last: &str) -> String {
    format!("{}, {}", last, first)
}

// Tuple return type
fn create_point(x: i32, y: i32) -> (i32, i32) {
    (x, y)
}
```

Array Operations

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

```rust
    println!("Sum: {}", sum_array(&numbers));
    println!("Max: {}", find_max(&numbers));

    let matrix = [[1, 2], [3, 4], [5, 6]];
    println!("Matrix sum: {}", sum_matrix(&matrix));
}

fn sum_array(arr: &[i32; 5]) -> i32 {
    arr.iter().sum()
}

fn find_max(arr: &[i32; 5]) -> i32 {
    *arr.iter().max().unwrap()
}

fn sum_matrix(matrix: &[[i32; 2]; 3]) -> i32 {
    matrix.iter().flatten().sum()
}
```

2. Borrow Checker and Ownership

Key Concepts for C/C++ Developers

- Move semantics by default (unlike C++ explicit std::move)
- Borrowing prevents data races at compile time
- No manual memory management
- RAII enforced by compiler

Ownership Transfer

```rust
fn main() {
    let s1 = String::from("hello");

    // s1 is moved into take_ownership
    take_ownership(s1);
    // println!("{}", s1); // Won't compile - s1 was moved

    let s2 = String::from("world");
    let s3 = clone_string(&s2);  // Borrowing s2, not moving
    println!("Original: {}, Clone: {}", s2, s3);

    // Alternative: use clone to avoid move
    let s4 = String::from("test");
    let s5 = s4.clone();
    take_ownership(s4);  // s4 moved
    println!("Still have: {}", s5);  // s5 still valid
}

fn take_ownership(s: String) {
    println!("Took ownership of: {}", s);
```

```
    // s is automatically dropped here
}

fn clone_string(s: &String) -> String {
    s.clone()
}
```

Borrowing Rules

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

    // Immutable borrow first
    let len = calculate_length(&data);

    // Then mutable borrow (no overlap with immutable)
    modify_vector(&mut data);

    println!("Length: {}, Modified: {:?}", len, data);

    // Multiple immutable borrows are OK
    let len1 = calculate_length(&data);
    let len2 = calculate_length(&data);
    println!("Lengths: {}, {}", len1, len2);
}

fn calculate_length(v: &Vec<i32>) -> usize {
    v.len()
}

fn modify_vector(v: &mut Vec<i32>) {
    for item in v.iter_mut() {
        *item += 10;
    }
}
```

3. References and Slices

Key Differences from C/C++

- Slices are fat pointers (pointer + length) vs simple pointers
- Compile-time lifetime checking prevents dangling references
- Automatic dereferencing in method calls
- No need for const& vs & distinction - mutability is explicit

Working with Slices

```
```

```rust
fn main() {
    let text = "The quick brown fox jumps";
    let numbers = [10, 20, 30, 40, 50, 60];

    println!("First word: '{}'", first_word(text));
    println!("Middle slice: {:?}", middle_slice(&numbers));
    println!("Largest in slice: {}", largest_in_slice(&numbers[1..4]));

    // String slicing
    let s = String::from("hello world");
    println!("First 5 chars: '{}'", &s[0..5]);
    println!("From index 6: '{}'", &s[6..]);
}

fn first_word(s: &str) -> &str {
    let bytes = s.as_bytes();

    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return &s[0..i];
        }
    }

    &s[..]  // Return entire string if no space found
}

fn middle_slice(arr: &[i32]) -> &[i32] {
    let len = arr.len();
    let start = len / 4;
    let end = start + len / 2;
    &arr[start..end]
}

fn largest_in_slice(slice: &[i32]) -> i32 {
    *slice.iter().max().unwrap()
}
```

Mutable vs Immutable References

```rust
fn main() {
    let mut text = String::from("hello world");

    // Must sequence borrows properly
    let word_count = count_words(&text);
    capitalize_first_letter(&mut text);

    println!("Words: {}, Text: '{}'", word_count, text);

    // Demonstrate borrowing rules
    let mut data = vec![1, 2, 3];
```

```rust
    {
        let r1 = &data;
        let r2 = &data;  // Multiple immutable refs OK
        println!("r1: {:?}, r2: {:?}", r1, r2);
    }  // r1 and r2 go out of scope

    let r3 = &mut data;  // Now mutable ref is OK
    r3.push(4);
    println!("After mutation: {:?}", r3);
}

fn count_words(s: &String) -> usize {
    s.split_whitespace().count()
}

fn capitalize_first_letter(s: &mut String) {
    if let Some(first_char) = s.chars().next() {
        let upper_char = first_char.to_uppercase().to_string();
        let rest = &s[first_char.len_utf8()..];
        *s = format!("{}{}", upper_char, rest);
    }
}
```

4. Structs and Methods

Key Differences from C/C++

- No inheritance, composition instead
- Methods defined in `impl` blocks
- Associated functions (like static methods)
- Automatic field initialization shorthand

Basic Structs

```rust
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

impl Rectangle {
    // Associated function (constructor)
    fn new(width: u32, height: u32) -> Rectangle {
        Rectangle { width, height }  // Field shorthand
    }

    // Instance method
    fn area(&self) -> u32 {
        self.width * self.height
    }
```

```rust
    fn perimeter(&self) -> u32 {
        2 * (self.width + self.height)
    }

    fn can_hold(&self, other: &Rectangle) -> bool {
        self.width >= other.width && self.height >= other.height
    }

    // Associated function to create square
    fn square(size: u32) -> Rectangle {
        Rectangle::new(size, size)
    }

    // Mutable method
    fn scale(&mut self, factor: u32) {
        self.width *= factor;
        self.height *= factor;
    }
}

fn main() {
    let mut rect1 = Rectangle::new(30, 50);
    let rect2 = Rectangle::square(25);

    println!("rect1: {:?}", rect1);
    println!("Area: {}", rect1.area());
    println!("Perimeter: {}", rect1.perimeter());
    println!("Can hold rect2: {}", rect1.can_hold(&rect2));

    rect1.scale(2);
    println!("After scaling: {:?}", rect1);
}
```

Complex Structs

```rust
#[derive(Debug, Clone)]
struct Person {
    name: String,
    age: u32,
    email: String,
}

impl Person {
    fn new(name: String, age: u32, email: String) -> Person {
        Person { name, age, email }
    }

    // Alternative constructor with validation
    fn try_new(name: String, age: u32, email: String) -> Result<Person, String> {
        if age > 150 {
            return Err("Age too high".to_string());
```

```rust
        }
        if !email.contains('@') {
            return Err("Invalid email".to_string());
        }
        Ok(Person::new(name, age, email))
    }

    fn is_adult(&self) -> bool {
        self.age >= 18
    }

    fn update_email(&mut self, new_email: String) {
        self.email = new_email;
    }

    fn greeting(&self) -> String {
        format!("Hello, I'm {} and I'm {} years old", self.name, self.age)
    }

    // Method that consumes self
    fn into_name(self) -> String {
        self.name
    }
}

fn main() {
    let mut person = Person::new(
        "Alice".to_string(),
        25,
        "alice@example.com".to_string()
    );

    println!("{}", person.greeting());
    println!("Is adult: {}", person.is_adult());

    person.update_email("alice.new@example.com".to_string());
    println!("Updated person: {:?}", person);

    // Try constructor with validation
    match Person::try_new("Bob".to_string(), 200, "bob@example.com".to_string()) {
        Ok(p) => println!("Created: {:?}", p),
        Err(e) => println!("Error: {}", e),
    }
}
```

5. Enums and Pattern Matching

Key Differences from C/C++

- Enums can hold data (algebraic data types)
- Pattern matching with `match` is exhaustive
- `Option<T>` replaces null pointers

- `Result<T, E>` for error handling

Basic Enums

```rust
#[derive(Debug)]
enum Shape {
    Circle(f64),
    Rectangle(f64, f64),
    Triangle(f64, f64, f64),
}

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) => {
                // Heron's formula
                let s = (a + b + c) / 2.0;
                (s * (s - a) * (s - b) * (s - c)).sqrt()
            }
        }
    }

    fn perimeter(&self) -> f64 {
        match self {
            Shape::Circle(radius) => 2.0 * std::f64::consts::PI * radius,
            Shape::Rectangle(width, height) => 2.0 * (width + height),
            Shape::Triangle(a, b, c) => a + b + c,
        }
    }

    // Pattern matching with guards
    fn classify(&self) -> &'static str {
        match self {
            Shape::Circle(r) if *r > 10.0 => "Large circle",
            Shape::Circle(_) => "Small circle",
            Shape::Rectangle(w, h) if w == h => "Square",
            Shape::Rectangle(_, _) => "Rectangle",
            Shape::Triangle(a, b, c) if a == b && b == c => "Equilateral 
triangle",
            Shape::Triangle(_, _, _) => "Triangle",
        }
    }
}

fn main() {
    let shapes = vec![
        Shape::Circle(5.0),
        Shape::Rectangle(10.0, 20.0),
        Shape::Triangle(3.0, 4.0, 5.0),
        Shape::Rectangle(15.0, 15.0),
```

```
    ];

    for shape in shapes {
        println!("{:?}", shape);
        println!("  Classification: {}", shape.classify());
        println!("  Area: {:.2}, Perimeter: {:.2}",
                shape.area(), shape.perimeter());
        println!();
    }
}
```

Option and Result

```
fn main() {
    let numbers = vec!["42", "not_a_number", "17", ""];

    for num_str in numbers {
        // Using Option
        match parse_number(num_str) {
            Some(n) => println!("Parsed '{}' as {}", num_str, n),
            None => println!("Failed to parse '{}'", num_str),
        }

        // Alternative: using if let
        if let Some(n) = parse_number(num_str) {
            println!("  Successfully parsed: {}", n);
        }
    }

    // Using Result
    let operations = vec![(10.0, 2.0), (10.0, 0.0), (15.0, 3.0)];

    for (a, b) in operations {
        match safe_divide(a, b) {
            Ok(result) => println!("{} / {} = {}", a, b, result),
            Err(e) => println!("Error: {}", e),
        }
    }

    // Chaining operations with ? operator
    match calculate_complex() {
        Ok(result) => println!("Complex calculation result: {}", result),
        Err(e) => println!("Calculation failed: {}", e),
    }
}

fn parse_number(s: &str) -> Option<i32> {
    s.parse().ok()  // Convert Result to Option
}

fn safe_divide(a: f64, b: f64) -> Result<f64, String> {
```

```rust
    if b == 0.0 {
        Err("Division by zero".to_string())
    } else {
        Ok(a / b)
    }
}

// Demonstrating ? operator for error propagation
fn calculate_complex() -> Result<f64, String> {
    let a = safe_divide(10.0, 2.0)?;  // ? propagates error
    let b = safe_divide(20.0, 4.0)?;
    let result = safe_divide(a + b, 3.0)?;
    Ok(result)
}

// Custom enum with methods
#[derive(Debug)]
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(i32, i32, i32),
}

impl Message {
    fn process(&self) {
        match self {
            Message::Quit => println!("Quitting application"),
            Message::Move { x, y } => println!("Moving to ({}, {})", x, y),
            Message::Write(text) => println!("Writing: {}", text),
            Message::ChangeColor(r, g, b) => println!("Changing color to RGB({},
{}, {})", r, g, b),
        }
    }
}
```

6. Generics and Traits

Key Differences from C/C++

- Traits similar to interfaces but more powerful
- Zero-cost abstractions through monomorphization
- Associated types and generic associated types
- Trait bounds replace template constraints

Generic Functions

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

fn main() {
    let numbers = vec![1, 5, 2, 9, 3];
```

```
    let words = vec!["hello", "world", "rust"];

    println!("Largest number: {}", largest(&numbers));
    println!("Largest word: {}", largest(&words));

    let point1 = Point { x: 3, y: 4 };
    let point2 = Point { x: 3.0, y: 4.0 };

    println!("Integer point distance: {}", point1.distance_from_origin());
    println!("Float point distance: {}", point2.distance_from_origin());

    // Generic function with multiple type parameters
    let pair1 = Pair::new(5, 10);
    let pair2 = Pair::new("hello", "world");
    pair1.cmp_display();
    pair2.cmp_display();
}

#[derive(Debug)]
struct Point<T> {
    x: T,
    y: T,
}

impl<T> Point<T>
where
    T: Copy + std::ops::Mul<Output = T> + std::ops::Add<Output = T>,
    f64: From<T>,
{
    fn distance_from_origin(&self) -> f64 {
        let x_squared = self.x * self.x;
        let y_squared = self.y * self.y;
        let sum = x_squared + y_squared;
        f64::from(sum).sqrt()
    }
}

fn largest<T>(list: &[T]) -> &T
where
    T: PartialOrd,
{
    let mut largest = &list[0];

    for item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}

// Generic struct with multiple type parameters
struct Pair<T, U> {
```

```rust
    x: T,
    y: U,
}

impl<T, U> Pair<T, U> {
    fn new(x: T, y: U) -> Self {
        Self { x, y }
    }
}

// Conditional method implementation
impl<T> Pair<T, T>
where
    T: Display + PartialOrd,
{
    fn cmp_display(&self) {
        if self.x >= self.y {
            println!("The largest member is x = {}", self.x);
        } else {
            println!("The largest member is y = {}", self.y);
        }
    }
}
```

Custom Traits

```rust
use std::f64::consts::PI;

trait Drawable {
    fn draw(&self);
    fn area(&self) -> f64;

    // Default implementation
    fn describe(&self) {
        println!("This shape has an area of {:.2}", self.area());
    }
}

trait Colorable {
    fn set_color(&mut self, color: String);
    fn get_color(&self) -> &str;

    // Default method with body
    fn reset_color(&mut self) {
        self.set_color("white".to_string());
    }
}

// Trait with associated types
trait Iterator {
    type Item;
```

```
    fn next(&mut self) -> Option<Self::Item>;

    // Default method using associated type
    fn collect_all(mut self) -> Vec<Self::Item>
    where
        Self: Sized,
    {
        let mut result = Vec::new();
        while let Some(item) = self.next() {
            result.push(item);
        }
        result
    }
}

#[derive(Debug)]
struct Circle {
    radius: f64,
    color: String,
}

#[derive(Debug)]
struct Square {
    side: f64,
    color: String,
}

impl Drawable for Circle {
    fn draw(&self) {
        println!("Drawing a {} circle with radius {}", self.color, self.radius);
    }

    fn area(&self) -> f64 {
        PI * self.radius * self.radius
    }
}

impl Colorable for Circle {
    fn set_color(&mut self, color: String) {
        self.color = color;
    }

    fn get_color(&self) -> &str {
        &self.color
    }
}

impl Drawable for Square {
    fn draw(&self) {
        println!("Drawing a {} square with side {}", self.color, self.side);
    }

    fn area(&self) -> f64 {
```

```
        self.side * self.side
    }
}

impl Colorable for Square {
    fn set_color(&mut self, color: String) {
        self.color = color;
    }

    fn get_color(&self) -> &str {
        &self.color
    }
}

// Function using trait bounds
fn draw_and_color<T>(mut shape: T, new_color: String)
where
    T: Drawable + Colorable,
{
    shape.draw();
    shape.set_color(new_color);
    println!("Changed color to: {}", shape.get_color());
    shape.describe();
}

// Function using trait objects
fn process_shapes(shapes: &mut [Box<dyn Drawable + Colorable>]) {
    for shape in shapes {
        shape.draw();
        println!("Current color: {}, Area: {:.2}", shape.get_color(),
shape.area());
        shape.reset_color();
    }
}

fn main() {
    let mut circle = Circle {
        radius: 5.0,
        color: "red".to_string()
    };
    let mut square = Square {
        side: 10.0,
        color: "blue".to_string()
    };

    // Using trait bounds
    draw_and_color(circle, "green".to_string());
    draw_and_color(square, "yellow".to_string());

    // Using trait objects
    let mut shapes: Vec<Box<dyn Drawable + Colorable>> = vec![
        Box::new(Circle { radius: 3.0, color: "purple".to_string() }),
        Box::new(Square { side: 8.0, color: "orange".to_string() }),
    ];
```

```
    process_shapes(&mut shapes);
}
```

7. Collections and Iterators

Key Differences from C/C++

- Iterator chains instead of manual loops
- Lazy evaluation with iterators
- Functional programming patterns
- No iterator invalidation issues

Working with Vec and HashMap

```rust
use std::collections::{HashMap, HashSet, VecDeque};

fn main() {
    let words = vec!["apple", "banana", "apple", "cherry", "banana", "apple"];

    let word_count = count_words(&words);
    println!("Word count: {:?}", word_count);

    let numbers = vec![1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
    let processed = process_numbers(&numbers);
    println!("Processed: {:?}", processed);

    // HashMap operations
    demonstrate_hashmap();

    // Other collections
    demonstrate_other_collections();
}

fn count_words(words: &[&str]) -> HashMap<String, usize> {
    let mut counts = HashMap::new();

    for word in words {
        *counts.entry(word.to_string()).or_insert(0) += 1;
    }

    counts
}

fn process_numbers(numbers: &[i32]) -> Vec<i32> {
    numbers
        .iter()
        .filter(|&&n| n % 2 == 0)  // Filter even numbers
        .map(|n| n * n)            // Square them
        .collect()                 // Collect into Vec
}
```

```rust
fn demonstrate_hashmap() {
    let mut scores = HashMap::new();

    // Insert values
    scores.insert("Alice".to_string(), 85);
    scores.insert("Bob".to_string(), 92);
    scores.insert("Charlie".to_string(), 78);

    // Access values
    match scores.get("Alice") {
        Some(score) => println!("Alice's score: {}", score),
        None => println!("Alice not found"),
    }

    // Update values
    scores.entry("Alice".to_string()).and_modify(|e| *e += 5);

    // Iterate over HashMap
    for (name, score) in &scores {
        println!("{}: {}", name, score);
    }
}

fn demonstrate_other_collections() {
    // HashSet for unique values
    let mut unique_numbers = HashSet::new();
    let numbers = vec![1, 2, 2, 3, 3, 3, 4];

    for num in numbers {
        unique_numbers.insert(num);
    }
    println!("Unique numbers: {:?}", unique_numbers);

    // VecDeque for efficient front/back operations
    let mut deque = VecDeque::new();
    deque.push_back(1);
    deque.push_back(2);
    deque.push_front(0);
    println!("Deque: {:?}", deque);

    if let Some(front) = deque.pop_front() {
        println!("Popped from front: {}", front);
    }
}
```

Iterator Chains

```rust
#[derive(Debug, Clone)]
struct Student {
    name: String,
```

```
    age: u32,
    grade: f64,
}

fn main() {
    let students = vec![
        Student { name: "Alice".to_string(), age: 20, grade: 85.5 },
        Student { name: "Bob".to_string(), age: 19, grade: 92.0 },
        Student { name: "Charlie".to_string(), age: 21, grade: 78.5 },
        Student { name: "Diana".to_string(), age: 20, grade: 88.0 },
        Student { name: "Eve".to_string(), age: 22, grade: 95.0 },
    ];

    let honor_students = find_honor_students(&students, 85.0);
    println!("Honor students: {:?}", honor_students);

    let average_age = calculate_average_age(&students);
    println!("Average age: {:.1}", average_age);

    let top_student = find_top_student(&students);
    println!("Top student: {:?}", top_student);

    // Complex iterator chain example
    let result: Vec<(String, f64)> = students
        .iter()
        .filter(|s| s.age >= 20)              // Adults only
        .filter(|s| s.grade >= 85.0)          // Honor roll
        .map(|s| (s.name.clone(), s.grade))   // Extract name and grade
        .collect();

    println!("Adult honor students: {:?}", result);

    // Demonstrate lazy evaluation
    demonstrate_lazy_evaluation();

    // Custom iterator
    demonstrate_custom_iterator();
}

fn find_honor_students(students: &[Student], min_grade: f64) -> Vec<String> {
    students
        .iter()
        .filter(|student| student.grade >= min_grade)
        .map(|student| student.name.clone())
        .collect()
}

fn calculate_average_age(students: &[Student]) -> f64 {
    let sum: u32 = students.iter().map(|s| s.age).sum();
    sum as f64 / students.len() as f64
}

fn find_top_student(students: &[Student]) -> Option<&Student> {
    students.iter().max_by(|a, b| a.grade.partial_cmp(&b.grade).unwrap())
```

```
}

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

    // This creates an iterator but doesn't execute yet
    let iter = numbers
        .iter()
        .map(|x| {
            println!("Processing: {}", x);  // This won't print yet
            x * 2
        });

    println!("Iterator created, but not consumed yet");

    // Now the iterator is consumed and processing happens
    let doubled: Vec<i32> = iter.collect();
    println!("Doubled: {:?}", doubled);
}

// Custom iterator implementation
struct Counter {
    current: usize,
    max: usize,
}

impl Counter {
    fn new(max: usize) -> Counter {
        Counter { current: 0, max }
    }
}

impl Iterator for Counter {
    type Item = usize;

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

fn demonstrate_custom_iterator() {
    let counter = Counter::new(5);

    let squared: Vec<usize> = counter
        .map(|x| x * x)
        .collect();

    println!("Squared numbers: {:?}", squared);
```

```
    // Using iterator adapters
    let sum: usize = Counter::new(10)
        .filter(|x| x % 2 == 0)  // Even numbers only
        .map(|x| x * x)          // Square them
        .sum();                  // Sum them up

    println!("Sum of squares of even numbers 0-9: {}", sum);
}
```

8. Smart Pointers

Key Differences from C/C++

- `Box<T>` for heap allocation (like unique_ptr)
- `Rc<T>` for reference counting (like shared_ptr)
- `RefCell<T>` for interior mutability
- No manual new/delete

Box and Rc

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

#[derive(Debug)]
struct Node {
    value: i32,
    children: Vec<Rc<Node>>,
}

impl Node {
    fn new(value: i32) -> Rc<Node> {
        Rc::new(Node {
            value,
            children: Vec::new(),
        })
    }

    // Since Node is behind Rc, we can't get &mut self
    // We'll create a version that returns a new Node with added child
    fn with_child(self: Rc<Self>, child: Rc<Node>) -> Rc<Node> {
        let mut children = self.children.clone();
        children.push(child);

        Rc::new(Node {
            value: self.value,
            children,
        })
    }

    fn sum_values(&self) -> i32 {
        let mut sum = self.value;
```

```
        for child in &self.children {
            sum += child.sum_values();
        }
        sum
    }
}

// Demonstrating Box<T>
#[derive(Debug)]
enum List {
    Cons(i32, Box<List>),
    Nil,
}

impl List {
    fn new() -> List {
        List::Nil
    }

    fn prepend(self, elem: i32) -> List {
        List::Cons(elem, Box::new(self))
    }

    fn len(&self) -> usize {
        match self {
            List::Cons(_, tail) => 1 + tail.len(),
            List::Nil => 0,
        }
    }

    fn stringify(&self) -> String {
        match self {
            List::Cons(head, tail) => {
                format!("{}, {}", head, tail.stringify())
            }
            List::Nil => format!("Nil"),
        }
    }
}

fn main() {
    // Using Box for recursive data structures
    let list = List::new()
        .prepend(1)
        .prepend(2)
        .prepend(3);

    println!("Linked list: {}", list.stringify());
    println!("Length: {}", list.len());

    // Using Rc for shared ownership
    let leaf = Node::new(3);
    let branch1 = Node::new(1).with_child(Rc::clone(&leaf));
    let branch2 = Node::new(2).with_child(leaf);
```

```
    let root = Node::new(0)
        .with_child(branch1)
        .with_child(branch2);

    println!("Tree sum: {}", root.sum_values());

    // Box for heap allocation
    let boxed_value = Box::new(42);
    println!("Boxed value: {}", boxed_value);

    // Large data on heap
    let large_array = Box::new([0; 1000000]);
    println!("First element of large array: {}", large_array[0]);
}
```

RefCell for Interior Mutability

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

#[derive(Debug)]
struct Counter {
    value: RefCell<i32>,
}

impl Counter {
    fn new() -> Counter {
        Counter {
            value: RefCell::new(0),
        }
    }

    fn increment(&self) {
        *self.value.borrow_mut() += 1;
    }

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

    fn add(&self, n: i32) {
        *self.value.borrow_mut() += n;
    }
}

// More complex example: Mock object pattern
#[derive(Debug)]
struct MockObject {
    call_count: RefCell<usize>,
    last_args: RefCell<Vec<String>>,
}
```

```rust
impl MockObject {
    fn new() -> Self {
        MockObject {
            call_count: RefCell::new(0),
            last_args: RefCell::new(Vec::new()),
        }
    }

    fn method_call(&self, args: Vec<String>) -> String {
        *self.call_count.borrow_mut() += 1;
        *self.last_args.borrow_mut() = args.clone();

        format!("Called with: {:?}", args)
    }

    fn get_call_count(&self) -> usize {
        *self.call_count.borrow()
    }

    fn get_last_args(&self) -> Vec<String> {
        self.last_args.borrow().clone()
    }
}

fn main() {
    // Basic RefCell usage
    let counter = Rc::new(Counter::new());

    let counter1 = Rc::clone(&counter);
    let counter2 = Rc::clone(&counter);
    let counter3 = Rc::clone(&counter);

    // Simulate multiple owners incrementing
    counter1.increment();
    counter2.add(5);
    counter3.increment();

    println!("Final counter value: {}", counter.get_value());

    // Mock object example
    let mock = MockObject::new();

    let result1 = mock.method_call(vec!["arg1".to_string(), "arg2".to_string()]);
    let result2 = mock.method_call(vec!["different".to_string()]);

    println!("Result 1: {}", result1);
    println!("Result 2: {}", result2);
    println!("Total calls: {}", mock.get_call_count());
    println!("Last args: {:?}", mock.get_last_args());

    // Combining Rc and RefCell for shared mutable state
    demonstrate_shared_mutable_state();
}
```

```rust
fn demonstrate_shared_mutable_state() {
    use std::collections::HashMap;

    type SharedMap = Rc<RefCell<HashMap<String, i32>>>;

    let shared_map: SharedMap = Rc::new(RefCell::new(HashMap::new()));

    // Function that modifies the shared map
    let modify_map = |map: SharedMap, key: String, value: i32| {
        map.borrow_mut().insert(key, value);
    };

    // Multiple references to the same map
    let map1 = Rc::clone(&shared_map);
    let map2 = Rc::clone(&shared_map);
    let map3 = Rc::clone(&shared_map);

    modify_map(map1, "first".to_string(), 10);
    modify_map(map2, "second".to_string(), 20);
    modify_map(map3, "third".to_string(), 30);

    // Read from original reference
    println!("Shared map contents:");
    for (key, value) in shared_map.borrow().iter() {
        println!("  {}: {}", key, value);
    }
}

// Example: Weak references to avoid cycles
use std::rc::Weak;

#[derive(Debug)]
struct Parent {
    children: RefCell<Vec<Rc<Child>>>,
}

#[derive(Debug)]
struct Child {
    parent: RefCell<Weak<Parent>>,
    name: String,
}

impl Parent {
    fn new() -> Rc<Self> {
        Rc::new(Parent {
            children: RefCell::new(Vec::new()),
        })
    }

    fn add_child(self: &Rc<Self>, name: String) -> Rc<Child> {
        let child = Rc::new(Child {
            parent: RefCell::new(Rc::downgrade(self)),
            name,
```

```rust
        });

        self.children.borrow_mut().push(Rc::clone(&child));
        child
    }
}

impl Child {
    fn parent(&self) -> Option<Rc<Parent>> {
        self.parent.borrow().upgrade()
    }
}
```

Summary

This refresher covers the key idiomatic Rust patterns that differ significantly from C/C++:

1. **Memory Safety**: Compile-time guarantees eliminate entire classes of bugs
2. **Ownership**: Move semantics and borrowing prevent data races
3. **Pattern Matching**: Exhaustive matching ensures all cases are handled
4. **Zero-Cost Abstractions**: High-level constructs compile to efficient code
5. **Functional Patterns**: Iterator chains and closures for elegant data processing
6. **Smart Pointers**: Automatic memory management without garbage collection

Each section demonstrates practical, compilable examples that showcase Rust's philosophy of "zero-cost abstractions" and "memory safety without garbage collection." The patterns shown here form the foundation of idiomatic Rust programming for systems-level development.