**Functional Programming using Rust**

[[](https://medium.com/@technocrat21?source=post_page-----3776c10cfc6--------------------------------)](https://medium.com/@technocrat21?source=post_page-----3776c10cfc6--------------------------------)

[[CoderHack.com](https://medium.com/coderhack-com?source=post_page-----3776c10cfc6--------------------------------)](https://medium.com/coderhack-com?source=post_page-----3776c10cfc6--------------------------------)

[Amay B](https://medium.com/@technocrat21?source=post_page-----3776c10cfc6--------------------------------)

·

Follow

Published in

[CoderHack.com](https://medium.com/coderhack-com?source=post_page-----3776c10cfc6--------------------------------)

·

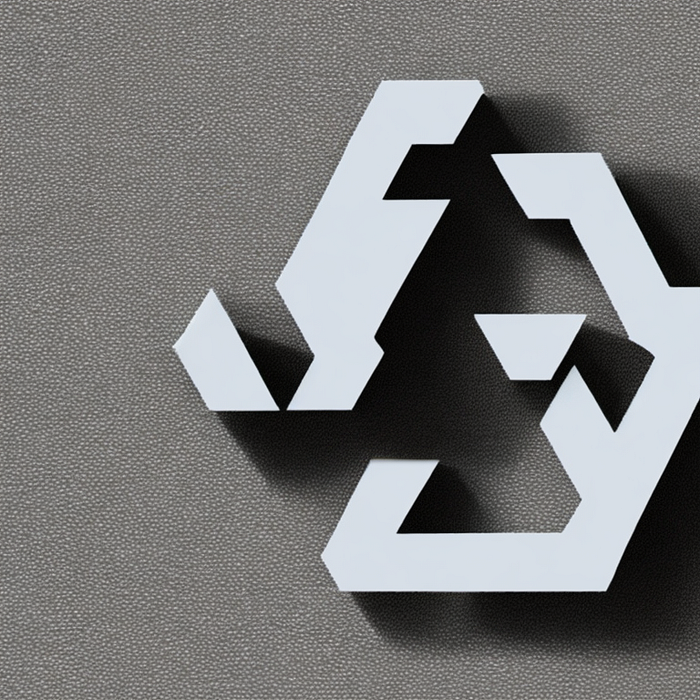
11 min read

·

Sep 21

177

1



Rust is a multi-paradigm programming language focused on safety, speed, and concurrency. It supports functional programming patterns in addition to procedural and object-oriented ones.

Functional programming (FP) is a programming paradigm where programs are constructed by applying and composing functions. It emphasizes pure functions, immutability, and recursion over mutable data and iteration.

Some of the main benefits of FP are:

* Easy to reason about and test pure functions — Easy to Maintain
* No side effects so functions are deterministic — **Easy to TEST**
* Concurrency is easier due to immutability — Possibly FASTEST
* Code is often more concise
* Can leverage powerful functional concepts like higher-order functions

Rust has many features that enable functional programming:

* Functions are first-class values
* Support for closures
* Tuples and enums to represent algebraic data types
* Pattern matching for easy function abstraction
* Traits for ad-hoc polymorphism
* Iterator adapter methods (map, filter, fold, etc)
* Minimal runtime so overhead of recursion and closures is low

Here’s an example of a simple recursive function in Rust that calculates factorials:

fn factorial(n: u64) -> u64 {  
 if n == 0 {  
 1  
 } else {  
 n \* factorial(n - 1)  
 }   
}  
  
fn main() {  
 println!("Factorial of 3 is: {}", factorial(3));  
}

This prints:

Factorial of 3 is: 6

In this article we’ll explore core functional programming concepts in Rust through examples. Let’s begin!

**Functions as First Class Values**

In Rust, functions are first class values, meaning they can be treated like any other value. This enables many powerful functional patterns.

**Storing functions in variables**

We can store functions in variables, just like any other value:

fn add(x: i32, y: i32) -> i32 {  
 x + y   
}  
  
let mut sum = add;  
  
sum(1, 2); // Returns 3

Here we store the add function in the sum variable, and then invoke it using sum(1, 2).

**Passing functions as arguments to other functions**

We can pass functions as arguments to other functions:

fn call\_with\_two(func: fn(i32, i32) -> i32, x: i32) -> i32 {  
 func(x, 2)  
}  
  
call\_with\_two(add, 1); // Returns 3

Here we define a call\_with\_two function that takes another function func and an integer x as arguments. It then invokes func with x and the value 2, and returns the result. We call it by passing the add function, and it returns 3.

**Returning functions from functions**

We can also return functions from other functions:

fn make\_adder(x: i32) -> fn(i32) -> i32 {  
 fn add(y: i32) -> i32 {  
 x + y  
 }  
 add   
}  
  
let add\_10 = make\_adder(10);  
  
add\_10(3); // Returns 13

Here make\_adder returns a closure (anonymous function) that adds its argument x to the parameter it receives. We call make\_adder with 10 and it returns a closure that adds 10 to its argument. We store that in add\_10 and invoke it with 3 to get 13.

**Closures**

Closures are anonymous functions that can capture their environment. We’ll cover closures in more detail in the next section.

**Iterators and Closures**

Iterators are a way to perform some action on a sequence of values in a collection. They are a key part of Rust’s standard library and are used in many functions.

A closure is an anonymous function that can capture its environment. Closures are used often with iterators to perform some logic on each element of a collection.

Here’s an example of an iterator that uses a closure:

let vec = vec![1, 2, 3];  
  
let doubled = vec.iter().map(|x| x \* 2);

The iter() method returns an iterator over the vector. The map() method takes a closure as an argument which will be called on each element. Here we multiply each element by 2. The |x| x \* 2 syntax defines an anonymous closure.

We can collect the resulting iterator into a new vector:

let doubled = vec.iter().map(|x| x \* 2).collect();   
// doubled is [2, 4, 6]

Here’s another example of using an iterator with a closure to filter elements:

let vec = vec![1, 2, 3, 4, 5];  
  
let even = vec.iter().filter(|x| x % 2 == 0).collect();  
// even is [2, 4]

The filter() method will call the closure on each element, and only keep elements where the closure returns true.

Closures capture their environment by borrowing values from the scope they’re defined in. For example:

let x = 10;  
  
let closure = |y| {   
 x + y   
};  
  
let answer = closure(2); // answer is 12

The closure uses the x variable from the outer scope.

Iterators and closures are essential functional concepts in Rust and enable a lot of useful functionality. Let me know if you have any other questions!

**Purity and Immutability**

Pure functions are essential to functional programming. A pure function has two important properties:

1. It returns the same output for the same set of inputs every time.
2. It has no side effects — it does not mutate any external state.

Let’s look at some examples of pure and impure functions in Rust:

// Pure function   
fn add(x: i32, y: i32) -> i32 {  
 x + y   
}  
  
// Impure function - depends on external mutable state  
static mut COUNTER: i32 = 0;  
  
fn increment\_counter() {  
 unsafe {  
 COUNTER += 1;  
 }  
}

The add() function is pure because:

1. For a given x and y, it will always return the same result.
2. It has no side effects — it does not manipulate any external state.

The increment\_counter() function is impure because:

1. It manipulates the external COUNTER variable, mutating its state.

To write pure functions in Rust, we need to follow some guidelines:

1. Never mutate external state (no use of mut)
2. Never read mutable static variables (no unsafe blocks)
3. Only transform inputs to outputs

However, we can still use mutation in pure functions as long as we only mutate locally declared variables. For example:

fn add\_one(mut x: i32) -> i32 {  
 x += 1;  
 x   
}

This function is pure because even though it mutates x, x is a locally declared variable. The mutation does not affect anything external to the function.

In summary, by adhering to purity and immutability in Rust, we get code that is:

1. Easy to reason about
2. Concurrency-friendly
3. Cacheable
4. Testable

Writing pure functions leads to a clean functional style in Rust. Let me know if you have any other questions!

**Recursion**

Recursion is a technique in programming where a function calls itself. A recursive function keeps calling itself until a stopping condition is met.

**How recursion works**

A recursive function must have two parts:

1. Base case: A stopping condition that doesn’t call the function recursively.
2. Recursive case: The function calls itself.

If a recursive function lacks either of these parts, it will continue endlessly in an infinite loop.

Here’s a basic example of recursion in Rust:

fn count\_down(n: i32) {  
 if n == 0 { // base case  
 println!("Done!");  
 } else { // recursive case  
 println!("{}", n);  
 count\_down(n - 1); // function calls itself  
 }  
}  
  
fn main() {  
 count\_down(5);  
}

This will print:

5   
4  
3  
2  
1   
Done!

**Recursive functions in Rust**

Here’s another example of a recursive function in Rust that calculates factorials:

fn factorial(n: i32) -> i32 {  
 if n == 0 { // base case  
 1  
 } else { // recursive case  
 n \* factorial(n - 1) // function calls itself  
 }  
}  
  
fn main() {  
 println!("5 factorial is {}", factorial(5));  
}

This will print:

5 factorial is 120

We satisfy the two conditions for recursion by having a base case of n == 0 which returns 1, and a recursive case that calls factorial(n — 1) until n reaches 0.

**When to use recursion**

Some good use cases for recursion are:

* Mathematical calculations (like factorials)
* Solving mazes or pathfinding
* Tree/graph traversals
* Divide and conquer algorithms

However, recursion should be used carefully. Recursive functions can often be rewritten using iterative loops, which may perform better and avoid stack overflow errors.

**Stack overflow and avoiding it**

Since each recursive call adds a stack frame, a recursive function can exceed the stack size limit and cause a stack overflow. This is a runtime error and will crash your program.

Some tips to avoid stack overflow with recursion:

1. Make sure there is a base case, and that it is reached.
2. Pass a decreasing argument to recursive calls.
3. Try rewriting the function iteratively.
4. Increase the stack size limit, though this is not recommended.

Using tail call optimization (TCO) can also help avoid stack overflow. This is where the recursive call is the last expression in the function. TCO allows the stack frame to be reused, preventing overflow.

Rust does not guarantee TCO, so iterative solutions are preferable when performance and stack space are a concern.

**Higher-Order Functions**

Higher-order functions are functions that operate on other functions, either by taking them as arguments or by returning them. They allow us to abstract over actions and compose functions in interesting ways.

**A. What are higher-order functions?**

A higher-order function is simply a function that takes one or more functions as arguments or returns a function as its result. Some examples in Rust would be:

* fn apply<F>(f: F) -> F - Takes a function and returns it
* fn compose<A, B, C>(f: fn(A) -> B, g: fn(B) -> C) -> fn(A) -> C - Takes two functions and returns a composed function
* iter.map(f) - Takes a function f and applies it to each element in the iterator

These functions operate on other functions, hence the name “higher-order functions”.

**B. Examples of higher-order functions**

Here are a few useful higher-order functions in Rust:

fn apply<F>(f: F) -> F {  
 f   
}  
  
fn compose<A, B, C>(f: fn(A) -> B, g: fn(B) -> C) -> fn(A) -> C {  
 move |x| g(f(x))  
}  
  
fn twice<F>(f: F) -> F   
where   
 F: FnOnce() -> ()   
{  
 move || {  
 f();  
 f();  
 }  
}

We can use these like so:

fn add\_5(x: i32) -> i32 {  
 x + 5   
}  
  
fn multiply(x: i32, y: i32) -> i32 {  
 x \* y  
}  
  
fn apply\_example() {  
 let f = apply(add\_5);  
 let result = f(10); // Uses the add\_5 function, result is 15  
}  
  
fn compose\_example() {  
 let h = compose(add\_5, multiply);   
 let result = h(2, 3); // Uses compose to do add\_5(multiply(2, 3)) = 17  
}   
  
fn twice\_example() {  
 let f = twice(|| println!("Hello!"));  
 f(); // Prints "Hello!" twice  
}

**C. Using iterators with higher-order functions**

We can use higher-order functions with iterators to perform powerful operations. For example:

let nums = vec![1, 2, 3, 4, 5];  
  
// Map   
let doubles = nums.iter().map(|x| x \* 2);   
// [2, 4, 6, 8, 10]  
  
// Filter   
let evens = nums.iter().filter(|&x| x % 2 == 0);  
// [2, 4]  
  
// Fold  
let sum = nums.iter().fold(0, |a, b| a + b);  
// 15

These are some useful techniques for manipulating collections in a declarative style.

**Functional Patterns**

Functional programming languages like Rust have some common patterns that we can use to write efficient programs. Let’s look at a few of the popular ones.

**Function pipelines**

Function pipelines let us compose functions by chaining them together. Each function takes the output of the previous function as its input. This allows us to transform data in complex ways by combining simple functions.

For example, say we have a list of numbers and we want to:

1. Get only the even numbers
2. Double each number
3. Get the sum of all numbers

We can use a pipeline to do this succinctly:

let nums = [1, 2, 3, 4, 5, 6];  
  
let sum = nums   
 .iter()  
 .filter(|&n| n % 2 == 0)  
 .map(|n| n \* 2)  
 .sum();  
  
assert\_eq!(sum, 20);

This is cleaner than writing three separate function calls and temporary variables to hold intermediate results.

**Map**

The map function applies a function to each element in an iterator and returns a new collection with the results. It allows us to transform a collection in a simple way.

For example, to double each number in a list, we could write:

let nums = [1, 2, 3, 4];  
let doubled = nums.iter().map(|n| n \* 2).collect();  
  
assert\_eq!(doubled, [2, 4, 6, 8]);

The map call applies the closure (|n| n \* 2) to each element and collects the results into a new vector.

**Filter**

The filter function applies a predicate function to each element in an iterator and returns a new collection with only the elements that returned true. It allows us to easily filter out elements we don’t want.

For example, to get only even numbers from a list, we could write:

let nums = [1, 2, 3, 4, 5, 6];   
let even = nums.iter().filter(|&n| n % 2 == 0).collect();  
  
assert\_eq!(even, [2, 4, 6]);

The filter call applies the predicate (&n| n % 2 == 0) and only keeps elements where it returns true.

**Reduce**

The reduce function (also called fold) collapses a collection into a single value by combining elements. It uses a closure to determine how to combine elements.

For example, to sum all numbers in a list, we could write:

let nums = [1, 2, 3, 4];  
let sum = nums.iter().reduce(|a, b| a + b).unwrap();  
  
assert\_eq!(sum, 10);

The reduce call iterates over the list, starting with a = 1 and b = 2. It then combines them by calling the closure (|a, b| a + b) which sums them, giving a = 3. It continues this process until the entire list is summed.

**Functional Patterns**

Functional patterns are common ways of combining functions to solve problems in a declarative style. Some common patterns in Rust include:

**A. Function Pipelines**

Function pipelines, or function composition, is applying multiple functions in a sequence to arrive at the final result. Each function takes the result of the previous function as its input.

For example, say we want to get the length of a string after converting it to uppercase. We could do:

let uppercased = "hello".to\_uppercase();  
let length = uppercased.len();

But with function pipelines, we can compose the two functions:

let length = "hello".to\_uppercase().len();

The . operator composes the two function calls, giving us a concise way to apply multiple transformations.

**B. Map**

The map function applies a closure to each element of an iterator and returns a new collection with the results. For example:

let nums = vec![1, 2, 3];  
let doubled = nums.iter().map(|x| x \* 2);   
// doubled is [2, 4, 6]

We used the iter() method to get an iterator over the vector, and then map()ed the closure |x| x \* 2 over it to double each element.

**C. Filter**

The filter() method keeps only elements of an iterator that satisfy a closure condition. For example:

let nums = vec![1, 2, 3, 4, 5];  
let even = nums.iter().filter(|n| n % 2 == 0);   
// even is [2, 4]

Here we filtered the iterator to only keep even numbers.

**D. Reduce**

The reduce() method combines all elements of an iterator into a single value using a closure. For example:

let nums = vec![1, 2, 3];  
let sum = nums.iter().reduce(|a, b| a + b);  
// sum is 6

We reduced the iterator by summing all elements together.

This covers some of the most common functional patterns used in Rust. Let me know if you have any other questions!

Further there are patterns around managing the state, monads. Hope to cover in next article.

**Conclusion**

In this article, we explored the basics of functional programming in Rust. We learned about:

* Functions as first-class values: Storing functions in variables, passing them as arguments to other functions, and returning them from functions.
* Closures: Anonymous functions that can capture their environment.
* Iterators: Lazy sequences in Rust that we can use with closures.
* Purity and immutability: The importance of pure functions and avoiding mutation.
* Recursion: Writing recursive functions in Rust and avoiding stack overflow.
* Higher-order functions: Functions that take other functions as arguments or return functions.
* Functional patterns: Function pipelines, map, filter, and reduce.

We saw how these concepts enable a declarative and expression-oriented style of programming in Rust. However, functional programming is not always the right solution. We need to consider performance, mutation requirements, and state changes.

Rust’s functional features combine the best of imperative and functional styles. Its strict ownership rules allow mutation when needed while still enabling pure, functional approaches. Overall, the functional aspects of Rust make it a versatile, expressive language for any programmer.

I hope this guide helped you learn the basics of functional programming in Rust! Let me know if you have any other questions.

I hope this article has been helpful to you! If you found it helpful please support me with 1) click some claps and 2) share the story to your network. Let me know if you have any questions on the content covered.

Feel free to contact me at coderhack.com(at)xiv.in