**Functional Language Features: Iterators and Closures.**

Rust’s design has taken inspiration from many existing languages and techniques, and one significant influence is *functional programming*. Programming in a functional style often includes using functions as values by passing them in arguments, returning them from other functions, assigning them to variables for later execution, and so forth.

In this note, we will cover:

* *Closures*, a function-like construct you can store in a variable.
* *Iterators*, a way of processing a series of elements.
* How to use closures and iterators.
* The performance of closures and iterators.

**Closures: Anonymous Functions That Capture Their Environment.**

**What is an Anonymous Function?**

An anonymous function (also called a lambda or closure, depending on the language) is a function without a name. It is usually defined inline, often as an argument to another function. Anonymous functions are commonly used for short operations like filtering, mapping, or sorting data.

Rust’s closures are anonymous functions you can save in a variable or pass as arguments to other functions. You can create the closure in one place and then call the closure elsewhere to evaluate it in a different context. Unlike functions, closures can capture values from the scope in which they’re defined. We’ll demonstrate how these closure features allow for code reuse and behavior customization.

**Capturing the Environment with Closures**

Example of how we can use closures to capture values from the environment they’re defined in for later use:

A t-shirt company that gives away free shirts. A customer may optionally specify a favorite color. If a preference is provided, the customer receives that color; otherwise, the most stocked color in inventory is chosen. The inventory is represented by a struct Inventory containing a Vec<ShirtColor> (either Red or Blue, defined using an enum).

The method giveaway uses unwrap\_or\_else to either return the provided preference or calculate the most stocked color using a closure: || self.most\_stocked This is a closure that takes no parameters itself (if the closure had parameters, they would appear between the two vertical bars).

In main, two test cases are run: one where the user prefers red, and another with no preference. The output reflects that the user's preference is honored when provided, and otherwise the most available color is selected:

A computer screen shot of a program

AI-generated content may be incorrect.

Running this code prints:

A close-up of a computer code

AI-generated content may be incorrect.

An important feature of closures in Rust is their ability to capture values from their surrounding environment. In the example, the closure used in unwrap\_or\_else captures an immutable reference to self to call self.most\_stocked(). This allows logic specific to the Inventory type to be embedded inside a generic standard library method.

Unlike closures, regular functions cannot capture external variables like self, so they require all necessary context to be passed explicitly as arguments.

**Closure Type Inference and Optional Annotation**

There are more differences between closure and functions. Closures don’t usually require developer to annotate the types of parameters or the return value like fn functions do.

Type annotations are required on functions because the types are part of explicit interface exposed to your users, ensuring that everyone agrees on what types of values a function uses and returns.

Closures, on the other hand, aren’t used in an exposed interface like this: they’re stored in variables and used without naming them and exposing them to users of our library, closure types are usually inferred by the compiler. This makes closures concise and ideal for short-lived tasks.

Example of defining a closure and storing it in variable:

A computer screen shot of a computer code

AI-generated content may be incorrect.

With type annotations added, the syntax of closures looks more similar to the syntax of functions, Example  illustrates how closure syntax is similar to function syntax except for the use of pipes and the amount of syntax that is optional:

A close-up of a number

AI-generated content may be incorrect.

The first line shows a function definition.

The second line shows a fully annotated closure definition.

The third line is removed the type annotations from the closure definition.

The fourth line, we remove the brackets, which are optional because the closure body has only one expression.

The add\_one\_v3 and add\_one\_v4 lines require the closures to be evaluated to be able to compile because the types will be inferred from their usage:

A white background with black text

AI-generated content may be incorrect.

For closure definitions, the compiler will infer one concrete type for each of their parameters and for their return value. That means closures are **monomorphic**, when once a closure is used with one type, that type is locked in. For example:

A black text on a white background

AI-generated content may be incorrect.

The compiler gives us this error:

A computer screen shot of a computer code

AI-generated content may be incorrect.

**Capturing by Reference, Mutable Reference, or Ownership**

Closures determine how to capture variables automatically. If the body only reads a variable, it borrows immutably:

A screen shot of a computer code

AI-generated content may be incorrect.

If the body modifies a variable, it captures it mutably:

A white background with colorful text

AI-generated content may be incorrect.

Note that there’s no longer a println! between the definition and the call of the borrows\_mutably closure: when borrows\_mutably is defined, it captures a mutable reference to list. We don’t use the closure again after the closure is called, so the mutable borrow ends.

A screen shot of a computer program

AI-generated content may be incorrect.

If you want to force the closure to take ownership of the values it uses in the environment even though the body of the closure doesn’t strictly need ownership, you can use the move keyword before the parameter list.

A computer code with colorful text

AI-generated content may be incorrect.

This technique is mostly useful when passing a closure to a new thread to move the data so that it’s owned by the new thread.

So why we must use move keyword when when spawn new thread”

In Previous example, the closure only captured list using an immutable reference because that’s the least amount of access to list needed to print it. In this example, even though the closure body still only needs an immutable reference, we need to specify that list should be moved into the closure by putting the move keyword at the beginning of the closure definition due to this reason:

This closure uses the variable list, which was defined in the main thread. Although the closure only reads from list (an immutable reference), there’s a risk:

* The main thread might finish first and drop list.
* The new thread would then be left holding a dangling reference to freed memory — something Rust strictly forbids.

To avoid unsafe behavior, Rust enforces that anything the new thread depends on must be owned by that thread. By adding move, you are moving ownership of list into the closure, and thus into the thread.

**Moving Captured Values Out of Closures and the Fn Traits**

Once a closure has captured a reference or captured ownership of a value from the environment where the closure is defined, the code in the body of the closure defines what happens to the references or values when the closure is evaluated later.

 A closure body can do any of the following: move a captured value out of the closure, mutate the captured value, neither move nor mutate the value, or capture nothing from the environment to begin with.

What a closure captures, and how it interacts with those captured values, directly impacts which traits the closure automatically implements. These traits are Fn, FnMut, and FnOnce, and they define how and how often a closure can be called, which is critical for APIs that accept closures as parameters:

* **FnOnce** is implemented by all closures. It allows a closure to be called **once**. A closure that **moves a captured value** (takes ownership) can only be called once and therefore only implements this trait.
* **FnMut** is implemented by closures that **do not move values out** but **may mutate** captured variables. These closures can be called **multiple times**, and they require **mutable access** to their environment.
* **Fn** is implemented by closures that **neither mutate** nor move values from their environment. These closures are the most flexible, can be called **multiple times**, and can be used concurrently.

**Example: Trait Bound in unwrap\_or\_else:**

The standard library method unwrap\_or\_else on Option<T> accepts a closure argument of type F: FnOnce() -> T. This means that the closure can be invoked once to produce a value of type T. Since all closures implement FnOnce, this method can work with any kind of closure:

A computer code with text

AI-generated content may be incorrect.

This trait bound ensures that if the Option is None, the closure will be invoked exactly once to produce a fallback value. If the Option is Some, the closure is never evaluated.

Note: If what we want to do doesn’t require capturing a value from the environment, we can use the name of a function rather than a closure. For example, we could call unwrap\_or\_else(Vec::new) on an Option<Vec<T>> value to get a new, empty vector if the value is None. The compiler automatically implements whichever of the Fn traits is applicable for a function definition.

A close-up of a code

AI-generated content may be incorrect.

This works because function pointers can also satisfy Fn, FnMut, and FnOnce traits depending on how they are used.

**Example: FnMut in sort\_by\_key**

Consider the sort\_by\_key method for slices, which requires a closure that implements FnMut. This is because the closure is invoked multiple times during sorting:

A screen shot of a computer code

AI-generated content may be incorrect.

The reason sort\_by\_key is defined to take an FnMut closure is that it calls the closure multiple times: once for each item in the slice. The closure |r| r.width doesn’t capture, mutate, or move out anything from its environment, so it meets the trait bound requirements.

**Improper Use of FnOnce: Compilation Error**

When a closure moves a captured value (i.e., takes ownership of it), it no longer satisfies the requirements of FnMut or Fn. In the following example, the closure attempts to push a captured String into a vector, which moves the value:

A screen shot of a computer code

AI-generated content may be incorrect.

This results in a compiler error because sort\_by\_key requires a closure that can be called multiple times, but value is only available once(The closure captures value and then moves value out of the closure by transferring ownership of value to the sort\_operations vector). The solution is to avoid moving the value or to clone it if necessary.

A screenshot of a computer program

AI-generated content may be incorrect.

We fix it by using a mutable value, num\_sort\_operations , and only capturing a mutable reference to the num\_sort\_operations counter and can therefore be called more than once.

The Fn traits are important when defining or using functions or types that make use of closures.

**Processing a Series of Items with Iterators**

An **iterator** in Rust is an object that allows you to **step through a sequence of values**, one at a time. It provides a way to **process elements** from a collection like a vector, array, or even a custom data structure, without exposing the underlying details of how the data is stored.

The iterator pattern allows you to perform some tasks on a sequence of items in turn. An iterator is responsible for the logic of iterating over each item and determining when the sequence has finished. When you use iterators, you don’t have to reimplement that logic yourself.

In Rust, iterators are *lazy*, meaning they have no effect until you call methods that consume the iterator to use it up. For example:

A black and red text

AI-generated content may be incorrect.

Here, v1\_iter is an iterator created from the vector v1 using the .iter() method. However, unless the iterator is used—for example, in a for loop—nothing is actually processed. This is demonstrated in the following use case:

A screenshot of a computer code

AI-generated content may be incorrect.

**The Iterator Trait and the next Method**

All iterators implement a trait named Iterator that is defined in the standard library. The definition of the trait looks like this:

A computer code with text

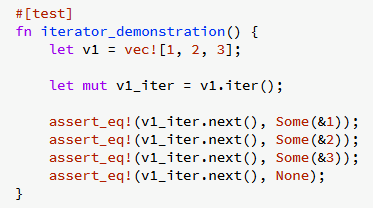
AI-generated content may be incorrect.

Iterator triat using new syntax: type Item and Self::Item; which are defining an *associated type* with this trait.

Iterator trait requires that you also define an Item type, and this Item type is used in the return type of the next method. In other words, the Item type will be the type returned from the iterator.

The Iterator trait only requires implementors to define one method: the next method, which returns one item of the iterator at a time, wrapped in Some and, when iteration is over, returns None.

We can call the next method on iterators directly:



In this test, v1\_iter must be mutable because calling next() changes the internal state of the iterator. Each call to next() consumes an element from the sequence.

We didn’t need to make v1\_iter mutable when we used a for loop because the loop took ownership of v1\_iter and made it mutable behind the scenes.

The values we get from the calls to next are immutable references to the values in the vector.

* The iter method produces an iterator over immutable references.
* Use into\_iter() to take ownership of v1 and returns owned values.
* Similarly, if we want to iterate over mutable references, we can call iter\_mut() instead of iter().

**Methods That Consume the Iterator**

The Iterator trait has a number of different methods with default implementations provided by the standard library.

Some of these methods call the next method in their definition, which is why you’re required to implement the next method when implementing the Iterator trait.

Methods that call next are called *consuming adapters* because calling them uses up the iterator. the sum method, which takes ownership of the iterator and iterates through the items by repeatedly calling next, thus consuming the iterator. A screenshot of a computer code

AI-generated content may be incorrect.

**Methods that Produce Other Iterators**

*Iterator adapters* are methods defined on the Iterator trait that don’t consume the iterator. Instead, they produce different iterators by changing some aspect of the original iterator. An example of calling the iterator adapter method map, which takes a closure to call on each item as the items are iterated through.

 The map method returns a new iterator that produces the modified items. The closure here creates a new iterator in which each item from the vector will be incremented by 1:

A number of mathematical equations

AI-generated content may be incorrect.

The code doesn’t do anything; the closure we’ve specified never gets called. The warning reminds us why: iterator adapters are lazy, and we need to consume the iterator here.

To fix this warning and consume the iterator, we’ll use the collect method, the method consumes the iterator and collects the resultant values into a collection data type.

A close-up of a computer code

AI-generated content may be incorrect.

We collect the results of iterating over the iterator that’s returned from the call to map into vector v2. This vector will end up containing each item from the original vector, incremented by 1.

Because map takes a closure, we can specify any operation we want to perform on each item. This is a great example of **how closures let you customize some behavior while reusing the iteration behavior that the Iterator trait provides**.

You can chain multiple calls to iterator adapters to perform complex actions in a readable way. But because **all iterators are lazy,** you have to **call one of the consuming adapter methods** to get results from calls to iterator adapters.

**Using Closures That Capture Their Environment**

Many iterator adapters **take closures as arguments**, and commonly the closures we’ll specify as arguments to iterator adapters will be closures that capture their environment.

For example:

A computer code with many colored text

AI-generated content may be incorrect.

We call into\_iter to create an iterator that takes ownership of the vector, filter to adapt that iterator into a new iterator that only contains elements for which the closure returns true.

The closure captures the shoe\_size parameter from the environment and compares the value with each shoe’s size, keeping only shoes of the size specified. Finally, calling collect gathers the values returned by the adapted iterator into a vector that’s returned by the function.