Functional Programming Features

Mustakimur R. Khandaker

Introduction

Rust's design has taken inspiration from many existing languages and techniques.

- One significant influence is functional programming.

Rust adopted function language features (some, not all):

- **Enums** (Already covered).
- **Pattern Matching** (Basics covered).
- **Closures**, a function-like construct you can store in a variable (Basics covered).
- **Iterators**, a way of processing a series of elements.

Closures (Memorization)

Memorization

Creating a struct that will hold the closure and the resulting value of the closure is a design approach to avoid computing the same value for each (same) closure call.

- The struct will execute the closure only if the resulting value is unavailable (first time), and it will cache the resulting value to avoid re-computation.
- This pattern is known as memoization or lazy evaluation.

```
struct Cacher<T>
where
    T: Fn(u32) -> u32,
{
    calculation: T,
    value: Option<u32>,
}
```

```
impl<T> Cacher<T>
where
    T: Fn(u32) -> u32,
{
    fn new(calculation: T) -> Cacher<T> {
        Cacher {
            calculation,
            value: None,
        }
    }
```

```
fn value(&mut self, arg: u32) -> u32 {
        match self.value {
            Some(v) => v,
            None => {
                let v =
(self.calculation)(arg);
                self.value = Some(v);
                v
            }
        }
    }
}
```

Continue ...

```
fn generate_workout(intensity: u32, random_number: u32) {
    let mut expensive_result = Cacher::new(|num| {
        println!("calculating slowly...");
        thread::sleep(Duration::from_secs(2));
        num
    });

    if intensity < 25 {
        println!("..", expensive_result.value(intensity));
        println!("..", expensive_result.value(random_number));
        println!("..", expensive_result.value(intensity));
    } else {
    }
}
```

All will return same value.

Let's Fix It!

```
struct CacherV2<T>
where
    T: Fn(u32) -> u32,
{
    calculation: T,
    value: HashMap<u32, u32>,
}
```

```
fn generate_workout_v2(intensity: u32, random_number: u32) {
    let mut expensive_result = CacherV2::new(|num| {
        println!("calculating slowly...");
        thread::sleep(Duration::from_secs(2));
        num
    });

    if intensity < 25 {
        println!("..", expensive_result.value(intensity));
        println!("..", expensive_result.value(random_number));
        println!("..", expensive_result.value(intensity));
    } else {
    }
}
```

```
impl<T> CacherV2<T>
where
    T: Fn(u32) -> u32,
{
    fn new(calculation: T) -> CacherV2<T> {
        CacherV2 {
            calculation,
            value: HashMap::new(),
        }
    }

    fn value(&mut self, arg: u32) -> u32 {
        match self.value.get(&arg) {
            Some(v) => v.clone(),
            None => {
                let v = (self.calculation)(arg);
                self.value.insert(arg, v);
                v
            }
        }
    }
}
```

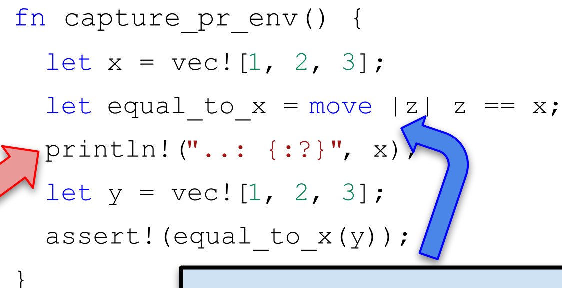
Capturing the Environment (Function vs Closures)

Capturing the environment means capability to access the variables of parent function.

- We could have nested function but would not be able to capture the environment.
- Closures, on the other hand, naturally can capture the environment (advantage).

```
fn capture_pr_env() {
  let x = 4;
  let equal_to_x = |z| z == x;
  let y = 4;
  assert!(equal_to_x(y));
}
```

```
fn capture_pr_env() {
  let x = 4;
  fn equal_to_x(z: i32) -> bool {
     z == x
  }
  let y = 4;
  assert!(equal_to_x(y));
}
```




```
$ cargo run
   Compiling equal-to-x v0.1.0 (file:///projects/equal-to-x)
error[E0434]: can't capture dynamic environment in a fn item
 --> src/main.rs:5:14
  |
5 |         z == x
  |              ^
  |
  = help: use the `|| { ... }` closure form instead
```

Function Traits

All closures implement at least one of the three Function traits: Fn, FnMut, or FnOnce.

- FnOnce consumes the variables it captures from its enclosing scope, known as the closure's environment.
  - To consume the captured variables, the closure must take ownership of these variables and move them into the closure when it is defined.
- FnMut can change the environment because it mutably borrows values.
- Fn borrows values from the environment immutably.

When you create a closure, Rust infers which trait to use based on how the closure uses the values from the environment.

Iterators

Introduction

An iterator is responsible for the logic of iterating over each item and determining when the sequence has finished.

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

```
let v1 = vec![1, 2, 3];

let v1_iter = v1.iter();

for val in v1_iter {
    println!("Got: {}", val);
}
```

```
Because, the next()
will modify the
environment of the
iterator e.g.
current index
pointer.
```

```
let v1 = vec![1, 2, 3];

let mut v1_iter = v1.iter();

assert_eq!(v1_iter.next(), Some(&1));
assert_eq!(v1_iter.next(), Some(&2));
assert_eq!(v1_iter.next(), Some(&3));
assert_eq!(v1_iter.next(), None);
```

The iter method produces an iterator over immutable references.
The iter_mut method produces an iterator over mutable references.
The into_iter method produces an iterator of own value (also moves the collection).

Iterator: Standard Method

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 we're required to implement the next method when implementing the Iterator trait.

```
let v1_iter = v1.iter();
let total: i32 = v1_iter.sum();

assert_eq!(total, 6);
```

Methods that call next are called consuming adaptors, because calling them uses up the iterator.

```
$ cargo run
   Compiling iterators v0.1.0
(file:///projects/iterators)
warning: unused `Map` that must be used
 --> src/main.rs:4:5
  |
4 |     v1.iter().map(|x| x + 1);
  |     ^^^^^^^^^^^^^^^^^^^^^^^^^
  |
  = note: `#[warn(unused_must_use)]` on by
default
  = note: iterators are lazy and do
nothing unless consumed
```

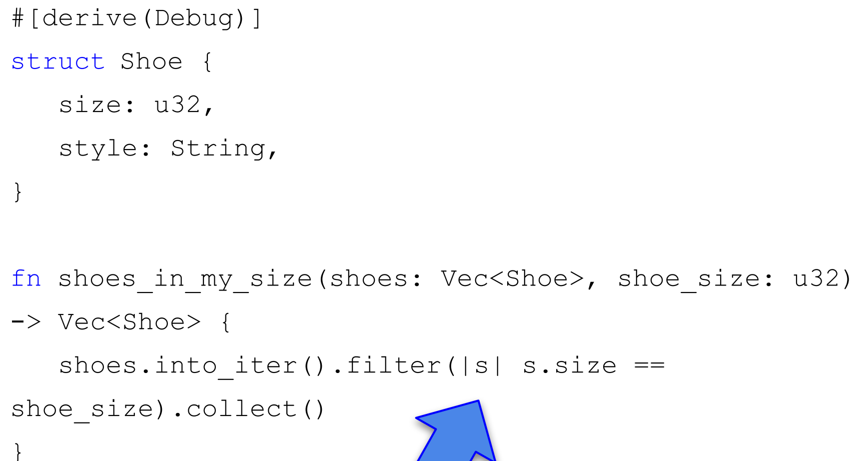
```
let v1: Vec<i32> = vec![1, 2, 3];

//v1.iter().map(|x| x + 1);
let v2: Vec<_> = v1.iter().map(|x| x +1).collect();

assert_eq!(v2, vec![2, 3, 4]);
```

iterator adaptors, allow us to change iterators into different kinds of iterators.

Closures within Iterator



```
#[derive(Debug)]
struct Shoe {
    size: u32,
    style: String,
}

fn shoes_in_my_size(shoes: Vec<Shoe>, shoe_size: u32)
-> Vec<Shoe> {
    shoes.into_iter().filter(|s| s.size ==
shoe_size).collect()
}
```

Here, filter with a closure that captures the shoe_size variable from its environment to iterate over a collection of Shoe struct instances. It will return only shoes that are the specified size.

```
fn iterator_filter() {
    let shoes = vec![
        Shoe {
            size: 10,
            style: String::from("sneaker"),
        },
        Shoe {
            size: 13,
            style: String::from("sandal"),
        },
        Shoe {
            size: 10,
            style: String::from("boot"),
        },
    ];

    let in_my_size = shoes_in_my_size(shoes,10);

    for item in in_my_size.iter() {
        println!("Items: {:?}", item);
    }
}
```

Implement Iterator

We can create iterators that do anything we want by implementing the Iterator trait on our own types.

- the only method we're required to provide a definition for is the next method.

```
struct Counter {
    count: u32,
}

impl Counter {
    fn new() -> Counter {
        Counter { count: 0 }
    }
}
```

```
impl Iterator for Counter {
    type Item = u32;

    fn next(&mut self) -> Option<Self::Item> {
        if self.count < 5 {
            self.count += 1;
            Some(self.count)
        } else {
            None
        }
    }
}
```

```
fn customize_iterator() {
    let mut counter =
Counter::new();

    let counter = Counter::new();

    for cnt in counter {
        println!("cnt: {}", cnt);
    }
}
```

Pattern Matching (Everything is Pattern)

Introduction

Patterns are a special syntax in Rust for matching against the structure of types, both complex and simple.

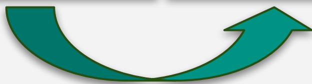
- A pattern consists of some combination of the following:
  - Literals.
  - Destructured arrays, enums, structs, or tuples.
  - Variables.
  - Wildcards.
  - Placeholders.

Patterns come in two forms: refutable and irrefutable.

- Patterns that will match for any possible value passed are irrefutable.
  - For example: in the statement `let x = 5;` because *x* matches anything and therefore cannot fail to match.
- Patterns that can fail to match for some possible value are refutable.
  - For example: Some(x) in the expression `if let Some(x) = a_value` because if the value in the `a_value` variable is None, the Some(x) pattern will not match.

```
let mut hash = HashMap::new();
hash.insert(String::from("key"), 10);
let Some(x) = hash.get(&String::from("none"));
```

```
if let Some(x) = hash.get(&String::from("none")) {
        println!("{}", x);
}
```



Conditional let (with Mixed)

```
fn cond_let() {
    let favorite_color: Option<&str> = None;
    let is_tuesday = false;
    let age: Result<u8, _> = "34".parse();
    //let age: Result<u8, _> = "".parse();

    if let Some(color) = favorite_color {
        println!("..", color);
    } else if is_tuesday {
        println!("...");
    } else if let Ok(age) = age {
        if age > 30 {
            println!("....");
        } else {
            println!(".....");
        }
    } else {
        println!("......");
    }
}
```

Okay to mix and match if let, else if, and else if let expressions.

```
fn while_let() {
    let mut stack = Vec::new();

    stack.push(1);
    stack.push(2);
    stack.push(3);

    while let Some(top) = stack.pop() {
        println!("{}", top);
    }
}
```

Pattern Syntax

```
let x = 1;

match x {
    1 => println!("one"),
    2 => println!("two"),
    3 => println!("three"),
    _ => println!("anything"),
}
```

```
let x = 1;

match x {
  1 | 2 => println!("one or two"),
  3 => println!("three"),
  _ => println!("anything"),
}
```

```
let x = 4;
match x {
    1..=5 => println!("one to
five"),
    _ => println!("something
else"),
}
```

```
    let x = Some(5);
    //let x = None;
    let y = 10;

    match x {
        Some(50) => println!("Got 50"),
        Some(y) => println!("Matched, y = {:?}", y), // new score of y variable
        _ => println!("Default case, x = {:?}", x),
    }

    println!("at the end: x = {:?}, y = {:?}", x, y);
```

Destructuring

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

fn destructure() {
    let p = Point { x: 0, y: 7 };

    let Point { x: a, y: b } = p;


    match p {
        Point { x, y: 0 } => println!(".. {}", x),
        Point { x: 0, y } => println!(".. {}", y),
        Point { x, y } => println!(".. ({}, {})", x, y),
    }
}
```

```
enum Color {
    Rgb(i32, i32, i32),
    Hsv(i32, i32, i32),
}
```

```
enum Message {
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(Color),
}
```

```
fn nested_destructure() {
    let msg = Message::ChangeColor(Color::Hsv(0, 160, 255));
    match msg {
        Message::ChangeColor(Color::Rgb(r, g, b)) => {
            println!("Change the color to red {}, green {},
and blue {}", r, g, b)
        }
        Message::ChangeColor(Color::Hsv(h, s, v)) =>
println!(
            "Change the color to hue {}, saturation {}, and
value {}",
            h, s, v
        ),
        _ => (),
    }
}
```

Ignore Items

```
let mut set_val = Some(5);
let n_set_val = Some(10);

match (set_val, n_set_val) {
   (Some(_), Some(_)) => {
      println!("...");
   }
   _ => {
      set_val = n_set_val;
   }
}

println!("... {:?}", set_val);
```

```
let numbers = (2, 4, 8, 16, 32);

match numbers {
   (first, .., last) => {
      println!("..: {}, {}", first, last);
   }
}
```

```
fn foo(_: i32, y: i32) {
   println!("This code only uses the y parameter: {}", y);
}
```

Match Guard

```
fn match_guard() {
    let num = Some(4);

    match num {
        Some(x) if x < 5 => println!("...: {}", x),
        Some(x) => println!("{}", x),
        None => (),
    }
}
```

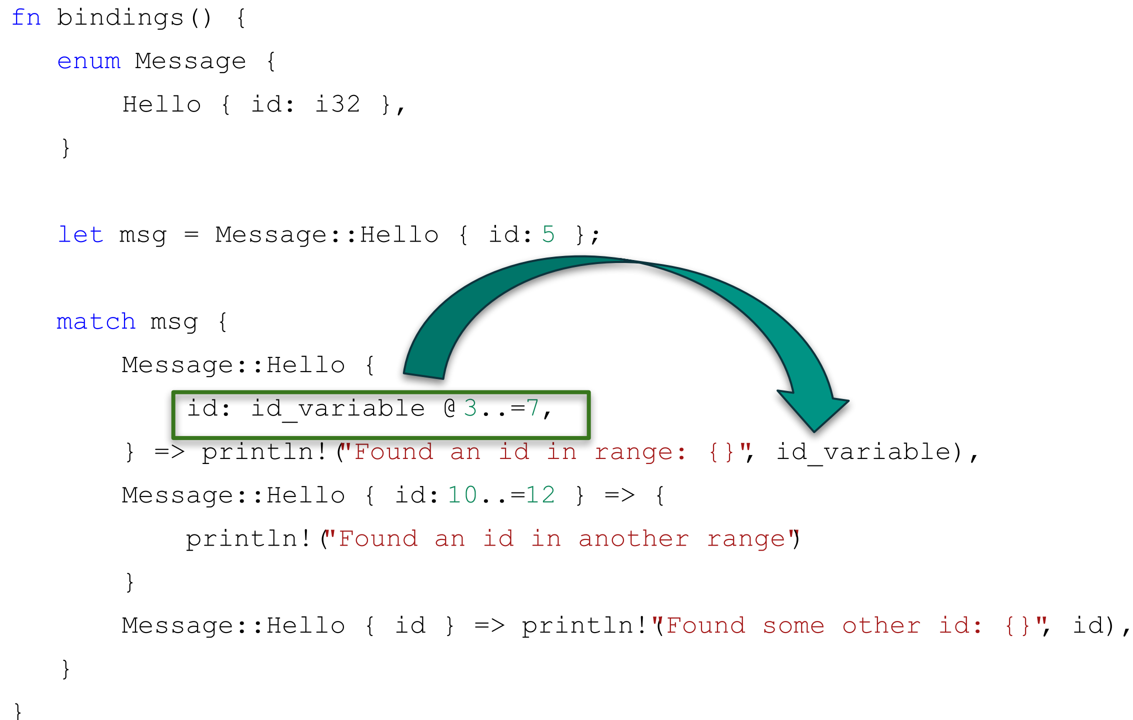
```
let x = Some(5);
let y = 10;
match x {
    Some(50) => println!("Got 50"),
    Some(n) if n == y => println!("..., n = {}", n),
    _ => println!("..., x = {:?}", x),
}
println!("...: x = {:?}, y = {}", x, y);
```

```
let x = 4;
let y = false;

match x {
  4 | 5 | 6 if y => println!("yes"),
  _ => println!("no"),
}
```

Bindings

The at operator (@) lets us create a variable that holds a value at the same time we're testing that value to see whether it matches a pattern.



```
fn bindings() {
    enum Message {
        Hello { id: i32 },
    }

    let msg = Message::Hello { id: 5 };

    match msg {
        Message::Hello {
            id: id_variable @ 3..=7,
        } => println!("Found an id in range: {}", id_variable),
        Message::Hello { id: 10..=12 } => {
            println!("Found an id in another range")
        }
        Message::Hello { id } => println!("Found some other id: {}", id),
    }
}
```

< Functional Programming Features />