Closures

What are Closures

- Similar to Anonymous Classes and Functions in other languages, closures are anonymous functions
- You basically assign a variable to a function body and can treat the variable as a function
- The Closure is only defined within the scope that it is declared
- It helps to make helper functions or functional programming easier when making small scope functions
- A great benefit of a closure is that it has access to all variables defined within the same scope
- However this is memory expensive as it captures the environment
- Also note that using Closures to its full potential lies in combining generics, structs, and Hashmaps together with closures

When is it good to use closures

- Usually, it is useful if you have multiple function calls of the same result within a scope
    - You can solve this by storing the result of the function call at the beginning of the scope
    - Ok, maybe its not needed in this case
- What if there's an if statement that makes the function call conditional on certain cases, you don't want to unnecessarily call and store the result of the function at the beginning of scope
    - You can store the result of the function call at after any check is made to determine whether the function is needed
- Ok, so these are the scenarios that is presented when showing that Closures are necessary... but lets be honest they aren't
- Closures are not necessary, although you can argue neither is generics for that matter
- HOWEVER, closures can help with the readability of your code and organization
- You can argue that it makes it more complicated especially when using it a Struct Closure, that's more of a "dealer's choice"
- In any case, it is a tool under the Rust belt and worth learning especially if you come across it later as the syntax is a little weird

Let's make a problem

- Let's assume we have a very expensive function call (to be honest any redundant function call is bad coding, but lets take it to the extreme)

- Example:

```
fn expensive_function(num: u32) -> u32 {
    println!("calculating slowly...");
    thread::sleep(Duration::from_secs(2));
    num
}
```

- Now we have an expensive function call that takes too long,
- Now lets make the problem in a complicated function
- Example:

```
fn problem_function(min: u32, tired: u32){

    if min < 25 as u32 {
        println!("It's too early: you should do {} situps",
expensive_function(tired));
        println!("It's too early: you should do {} pushups",
expensive_function(tired));
    }
    else {
        if tired == 0 as u32 {
            println!("your too tired, take a break");
        }
        else{
            println!("You got some energy left, cooldown time {}",
expensive_function(tired));
        }
    }
}
```

- There are many easy ways to fix the problem we just created and closures is not needed as we know
- And I know that the double call in the first `if` statement is redundant
- But let's refactor this function with closure

Defining a Closure

- A Closure syntax mimics that of a function with some extra features that make it easier and faster to write
- Example:

```
let expensive_closure = |num| {
    println!("calculating slowly...");
    thread::sleep(Duration::from_secs(2));
    num
};
```

  - We see that we use a `let` statement, because remember we are defining a variable as a function body
  - The `|<param>|` syntax is how we define parameters to the closure function

- We can notice 2 things that are very different from a normal function and that is the `explicit type declaration`
  - In rust, since closures are only defined within a certain scope, after it's first use case, the compiler will be able to determine the type of parameter and return
  - However you can explicitly define it if need be
  - Example: `rust let expensive_closure = |num: u32| -> u32 { println! ("calculating slowly..."); thread::sleep(Duration::from_secs(2)); num };`
- Rust will only infer the type of parameter and return type using the first call
- This means I cannot use the closure for any other types (unless we use generics)
- Simple example:

```
let example_closure = |x| x;

let s = example_closure(String::from("hello"));
let n = example_closure(5);
```

  - The `let n` line is an error because that uses an `i32` type when after setting `let s` makes the closure `String`

- Now lets first throw the closure into the function body
- Example:

```
fn problem_function(min: u32, tired: u32){
    let expensive_closure = |num| {
        println!("calculating slowly...");
        thread::sleep(Duration::from_secs(2));
        num
    };

    if min < 25 as u32 {
        println!("It's too early: you should do {} situps",
expensive_closure(tired));
        println!("It's too early: you should do {} pushups",
expensive_closure(tired));
    }
    else {
        if tired == 0 as u32 {
            println!("your too tired, take a break");
        }
        else{
            println!("You got some energy left, cooldown time {}",
expensive_closure(tired));
        }
    }
}
```

- Note we didn't fix anything yet, we are just seeing how the closure looks into the function

Using a Closure/Struct to solve the problem

- First let's make a struct that takes a closure as a field
- To do this we need to use a generic with a trait bound
- Example:

```
struct Closures<T, U> where T: Fn(U) -> U, U: Copy{
    calculation: T,
    value: Option<U>,
}
```

  - Let's digest this slowly, we are making a struct called `Closures`
  - It has two generics, `<T,U>`
  - One of these Generics, `<T>` inherits from the `Fn` trait
    - This `Fn()->` trait is one of 3 traits that implement `closure` functions
    - There is also `FnOnce` and `FnMut` traits which we will explain later
  - The `Fn()` trait takes a generic type `U` as a parameter and returns the same generic type back
  - The generic `U` takes in a `Copy` trait, this just ensures we do not have to worry about types that cannot be `copied` and needs to be `move`
    - Gets around the ownership issues
  - Then we have two fields
    - `calculation` is of type `T` that implements the trait `Fn` meaning its a field of **any** `closure type`
    - `value` is of type `Option<U>` which is the same type that the trait `Fn` takes in and returns
      - Value is going to be the field that stores the result
  - In short we have made a wrapper around the closure such that we can store the result of the closure instead of rerunning the function.
- We need to now create some methods for the closure and to unwrap the value
- Example:

```
impl <T,U> Closures<T, U>  where T: Fn(U) -> U, U: Copy {
    fn new(calculation: T) -> Closures<T, U> {
        Closures{
            calculation: calculation,
            value: None,
        }
    }

    fn get_result(&mut self, arg: U) -> U {
        match self.value {
            Some(val) => val,
            None => {
                let val = (self.calculation)(arg);
                self.value = Some(val);
                val
```

```
            }
        }
    }
}
```

- Now let's digest this slowly
- First we need a `new` method that creates a new struct based on a closure
  - Here the `value` field is just `None` because we don't know the values that will be passed into the closure
- Next we see the `get_result` function
  - This function is meant to return the result of the closure depending on the `arg`
  - We can see that if `value` field is None, meaning that the `struct` was first initialized
  - Only then do we run the calculation and return its result
  - If there is already **any** value then we assume the arg is the same as before and just return the binded value from `Some()`
- We can see there is a couple of problems with this
  - For one, this assumes that the the `value` for any individual `Closrues` struct is the same
  - In other words, we cannot set any new results to the same closure struct.
  - This can be solved by rearranging the Struct using HashMaps where the args are the keys and the results of the closure function are the values
  - To make this the most complete and powerful solution

```
struct Closures<T, U> where T: Fn(U) -> U, U: Copy + PartialEq + Eq +
Hash{
    calculation: T,
    value: HashMap<U, U>,
}

impl <T,U> Closures<T, U>  where T: Fn(U) -> U, U: Copy + PartialEq +
Eq + Hash{
    fn new(calculation: T) -> Closures<T, U> {
        Closures{
            calculation: calculation,
            value: HashMap::new(),
        }
    }

    fn get_result(&mut self, arg: U) -> U {
        match self.value.get(&arg) {
            Some(val) => *val,
            None => {
                let val = (self.calculation)(arg);
                self.value.insert(arg, val);
                val
            }
        }
    }
}
```

        - Now we have implemented a HashMap system so that the parameter and return type are the same
        - Also note that to use a HashMap with generics, you need specific traits for the generic
- Since in our specific example, we pass the same argument to the same closure, we do not need to worry about the super complete form, but it won't make a difference in how we use it right now
- Putting everything together
- Example:

```
fn problem_function(min: u32, tired: u32){

    let expensive_closure = |num| {
        println!("calculating slowly...");
        thread::sleep(Duration::from_secs(2));
        num
    };

    let mut struct_closure = Closures::new(expensive_closure);

    if min < 25 as u32 {
        println!("It's too early: you should do {} situps",
struct_closure.get_result(tired));
        println!("It's too early: you should do {} pushups",
struct_closure.get_result(tired));
    }
    else {
        if tired == 0 as u32 {
            println!("your too tired, take a break");
        }
        else{
            println!("You got some energy left, cooldown time {}",
struct_closure.get_result(tired));
        }
    }
}
```

    - First we will break it down and then explain the benefits
        - The expensive_closure is used as a calculation field for the `struct_closure`
        - Then we see that we substitute the `expensive_closure` function calls with the Struct `get_result` function calls to get the result
    - Surface layer it all looks the same and is just as readable as making multiple functions calls but its more efficient here.
        - First we make it very clear the number of times we get a result from the `expensive_closure` function
        - But we don't run the `expensive_closure` function each time. We only run it once and that's because the `get_result` only calls the `expensive_closure` if the `arg` is completely new
        - Otherwise it will just return the value from the hashmap

- Now we maintain the same level of readability from the terrible bad coding at the beginning of the problem, without the negative issues that comes with it

Environment Vairables

- One topic of the closures that we do not discuss directly is the environment variables
- Closures are seen as memory intensive functions, besides the HashMap and struct we used to make this code
- This is because when making a closure it captures all the environment variables in that scope so it can be used within the closure body, this means a little more memory is allocated to the closure than typical functions
- Also the way it interacts with the environment variables and parameters is a little more complicated as well
- As we saw earlier there were serveral traits of closures: `Fn(), FnMut(), and FnOnce` each one maps to the three ways a function can take a parameter: borrowing immutably, borrowing mutably, and taking ownership of the parameter
  - `Fn` looks something like this:

```
let x = 5;

let func = || x;
```

    - This borrows a reference to the value of `x` and returns `x`
    - But this does not take ownership of the parameter
    - Also note that `x` is not an explicit parameter but is an **environment vairable**
    -
  - `FnMut` looks something like this:

```
let mut x = 5;

let mut func = || x = 7;
```

    - By making the function `FnMut` we can alter mutable environment variables
    - Note that we don't have to explicitly make the closure `mut`, as long as the environment vairable is `mut` and we attempt to change it in the closue, the compiler can determine whether or not it is `FnMut`
  - `FnOnce` looks something like this:

```
/*FnOnce */
let x = String::from("hi");

let func = move || x == String::from("5");
```

```
//println!("x: {}", x); error
```

- The rust compiler will always assume `Fn` or immutable borrowing by default, if you want to do an `FnOnce` where the closure takes ownership of the environment variable, then you need to use the key word `move`
- Remember this only applies to data types that do not `copy` by default, and so we use a `String` in this case and printing `x` later becomes an error