Lecture 8

Closures

Function

```
fn double(x: i32) -> i32 {
        2 * x
}

fn main() {
    let v = vec!(1, 2, 3);
    let w = v.into_iter().map(double);
    println!("{:?}", w.collect::<Vec<i32>>());
}

//Output:
[2, 4, 6]
```

Closure

```
fn main() {
    let v = vec!(1, 2, 3);
    let w = v.into_iter().map(|x| 2 * x);
    println!("{:?}", w.collect::<Vec<i32>>());
}

//Output:
[2, 4, 6]
```

Named Closure

```
fn main() {
    let double = |x| 2 * x;
    let v = vec!(1, 2, 3);
    let w = v.into_iter().map(double);
    println!("{:?}", w.collect::<Vec<i32>>());
}

//Output:
[2, 4, 6]
```

Fn Closures

- Closures that access variables only for read access implement the Fn trait.
- Any value they access are as reference types (&T).
- This is the default mode of borrowing the closures assumes.

```
fn main() {
        let a = String::from("Hey!");
        let fn_closure = || {
                println!("Closure says: {}", a);
         };
        fn_closure();
        println!("Main says: {}", a);}
//Output:
Closure says: Hey!
Main says: Hey!
```

FnMut Closures

- When the compiler figures out that a closure mutates a value referenced from the environment, the closure implements the FnMut trait.
- The following closure adds the "Alice" string to a. fn_mut_closure mutates its environment.

```
fn main() {
        let mut a = String::from("Hey!");
        let mut fn_mut_closure = || {
                a.push_str(" Alice");
        };
        fn_mut_closure();
        println!("Main says: {}", a);
}

//Output:
Main says: Hey! Alice
```

FnOnce Closures

- Closures that take ownership of the data they read from their environment get implemented with FnOnce.
- The name signifies that this closure can only be called once and, because of that, the variables are available only once.

```
fn main() {
        let mut a = Box::new(23);
        let call_me = || {
                let c = a;
        };
        call_me();
        call_me();
}
```

FnOnce Closures

```
fn main() {
        let mut a = Box::new(23);
        let call_me = || {
                let c = a;
        };
        call_me();
        call_me();
}
```

- This fails with the following error:

```
error[E0382]: use of moved value: `call_me`
 --> src\main.rs:7:2  |
6 |      call_me();
  |      ------- value moved here
7 |      call_me();
  |      ^^^^^^^ value used here after move
note: closure cannot be invoked more than once because it
moves the variable `a` out of its environment
 --> src\main.rs:4:11  |
4 |          let c = a;  |                    ^
```

Closures in Rust

```
let k = my_vec.iter().filter(|n| **n > 0).count();
```

- Guarantees to be just as fast as writing out an explicit loop!
- *iter()* gives an iterator over references to the elements (say &i32).
- *filter* takes a closure which is passed a reference to the iterator type (&&i32).
- Need a double dereference to get the actual integer.
- Rust avoids overhead by inlining closures.

Careful!

```
let g = "Good Morning".to_string();
let f = || {
    let g2 = g;
    println!("we got {}", g2);
};
println!("g is {}", g);

//     let f = || {
//   |         -- value moved (into closure) here

//   |     println!(“g is {}", g);
//   |                          ^ value used here
after move
```

- g has to move into g2, and then gets dropped.

Closures and Structs

```
struct CustomPair {
    i: f64,
    j: f64
}

fn main(){
      let m = CustomPair{i: 1.0, j: 2.0};
      let f = |y:CustomPair, x:f64|->f64 {y.i*x
      + y.j};
      println!("{:?}",f(m,3.0));
}
```

Traits: Defining Shared Behavior

- Tells Rust compiler about functionality a particular type has and can share with other types.
- We can use traits to define shared behavior in an abstract way.
- Traits commonly take self as a parameter.

```
pub trait Summary {
    fn summarize(self) -> String;
}
```

Implementing a Trait on a Type

- Now that we've defined the Summary trait, we can implement it on types.

```
pub struct Tweet {
    pub username: String,
    pub content: String,
    pub reply: bool,
    pub retweet: bool,
}

impl Summary for Tweet {
    fn summarize(self) -> String {
        format!("{}: {}", self.username,
                self.content)
    }
}
```

Trait Bounds

- We can use trait bounds to constrain generic types to ensure the type will be limited to those that implement a particular trait and behavior.

```
pub fn notify<T: Summary>(item: T) {
    println!("Breaking news! {}", item.summarize());
}
```

Trait Bounds

- However, there are downsides to using too many trait bounds. (e.g., unreadable!)
- For this reason, Rust has alternate syntax for specifying trait bounds inside a *where clause* after the function signature. So instead of writing this:

```
fn some_function<T: Display + Clone, U: Clone +
Debug>(t: T, u: U) -> i32 {
```

- Can write something like:

```
fn some_function<T, U>(t: T, u: U) -> i32
    where T: Display + Clone,
          U: Clone + Debug
{
```

Trait Bounds for Closures

- The type bound for the f argument reads: f is any type that implements Fn(f64)->f64.

```
fn invoke<F>(f: F, x: f64) -> f64
where F: Fn(f64)->f64 {
    f(x)
}
```