Closures

CIS 198 Lecture 4

Closures

- A closure, anonymous function, or lambda function is a common paradigm in functional languages.
- In Rust, they're fairly robust, and match up well with the rest of Rust's ownership model.

```rust
let square = |x: i32| -> i32 { x * x };
println!("{}", square(3));
// => 6
```

Closure Syntax

```
let foo_v1 = |x: i32| { x * x };
let foo_v2 = |x: i32, y: i32| x * y;
let foo_v3 = |x: i32| {
    // Very Important Arithmetic
    let y = x * 2;
    let z = 4 + y;
    x + y + z
};
let foo_v4 = |x: i32| if x == 0 { 0 } else { 1 };
```

- These look pretty similar to function definitions.
- Specify arguments in `||`, followed by the return expression.
    - The return expression can be a series of expressions in `{}`.

Type Inference

```
let square_v4 = |x: u32| { (x * x) as i32 };

let square_v4 = |x| -> i32 { x * x }; // ← unable to infer enough
let square_v4 = |x|        { x * x }; // ← type information!
```

- Unlike functions, we don't *need* to specify the return type or argument types of a closure.
  - In this case, the compiler can't infer the type of the argument `x` from the return expression `x * x`.

Closure Environment

- Closures *close* over (contain) their environment.

```rust
let magic_num = 5;
let magic_johnson = 32;
let plus_magic = |x: i32| x + magic_num;
```

- The closure `plus_magic` is able to reference `magic_num` even though it's not passed as an argument.
  - `magic_num` is in the "environment" of the closure.
  - `magic_johnson` is not borrowed!

Closure Environment

- If we try to borrow `magic_num` in a conflicting way after the closure is bound, we'll get an error from the compiler:

```
let mut magic_num = 5;
let magic_johnson = 32;
let plus_magic = |x: i32| x + magic_num;

let more_magic = &mut magic_num; // Err!
println!("{}", magic_johnson); // Ok!
```

```
error: cannot borrow `magic_num` as mutable because it is
already borrowed as immutable

[...] the immutable borrow prevents subsequent moves or mutable
borrows of `magic_num` until the borrow ends
```

- Why? `plus_magic` borrows `magic_num` when it closes over it!
- However, `magic_johnson` is not used in the closure, and its ownership is not affected.

Closure Environment

- We can fix this kind of problem by making the closure go out of scope:

```
let mut magic_num = 5;
{
    let plus_magic = |x: i32| x + magic_num;
} // the borrow of magic_num ends here

let more_magic = &mut magic_num; // Ok!
println!("magic_num: {}", more_magic);
```

Move Closures

- As usual, closures are choose-your-own-~~adventure~~ ownership.
- Sometimes it's not okay to have a closure borrow *anything*.
- You can force a closure to *take ownership* of all environment variables by using the `move` keyword.
    - "Taking ownership" can mean taking a copy, not just moving.

```rust
let mut magic_num = 5;
let own_the_magic = move |x: i32| x + magic_num;
let more_magic = &mut magic_num;
```

Move Closures

- `move` closures are necessary when the closure `f` needs to outlive the scope in which it was created.
  - e.g. when you pass `f` into a thread, or return `f` from a function.
  - `move` essentially *disallows* bringing references into the closure.

```
fn make_closure(x: i32) -> Box<Fn(i32) -> i32> {
    let f = move |y| x + y; // ^ more on this in 15 seconds
    Box::new(f)
}

let f = make_closure(2);
println!("{}", f(3));
```

Closure Ownership

- Sometimes, a closure *must* take ownership of an environment variable to be valid. This happens automatically (without `move`):
    - If the value is moved into the return value.

    ```
    let lottery_numbers = vec![11, 39, 51, 57, 75];
    {
        let ticket = || { lottery_numbers };
    }
    // The braces do no good here.
    println!("{:?}", lottery_numbers); // use of moved value
    ```

    - Or moved anywhere else.

    ```
    let numbers = vec![2, 5, 32768];
    let alphabet_soup = || { numbers; vec!['a', 'b'] };
                          // ^ throw away unneeded ingredients
    println!("{:?}", numbers); // use of moved value
    ```

- If the type is not `Copy`, the original variable is invalidated.

Closure Ownership

```
let numbers = vec![2, 5, 32768];
let alphabet_soup = || { numbers; vec!['a', 'b'] };
                      // ^ throw away unneeded ingredients
alphabet_soup();
alphabet_soup(); // use of moved value
```

- Closures which own data and then move it can only be called once.
  - `move` behavior is implicit because `alphabet_soup` must own `numbers` to move it.

```
let numbers = vec![2, 5, 32768];
let alphabet_soup = move || { println!("{:?}", numbers) };
alphabet_soup();
alphabet_soup(); // Delicious soup
```

- Closures which own data but don't move it can be called multiple times.

Closure Ownership

- The same closure can take some values by reference and others by moving ownership (or Copying values), determined by behavior.

Closure Traits

- Closures are actually based on a set of traits under the hood!
    - `Fn`, `FnMut`, `FnOnce` - method calls are overloadable operators.

```
pub trait Fn<Args> : FnMut<Args> {
    extern "rust-call"
      fn call(&self, args: Args) -> Self::Output;
}

pub trait FnMut<Args> : FnOnce<Args> {
    extern "rust-call"
      fn call_mut(&mut self, args: Args) -> Self::Output;
}

pub trait FnOnce<Args> {
    type Output;

    extern "rust-call"
      fn call_once(self, args: Args) -> Self::Output;
}
```

Closure Traits

- These traits all look pretty similar, but differ in the way they take `self`:
  - `Fn` borrows `self` as `&self`
  - `FnMut` borrows `self` mutably as `&mut self`
  - `FnOnce` takes ownership of `self`
- `Fn` is a superset of `FnMut`, which is a superset of `FnOnce`.
- Functions also implement these traits.

"The `|| {}` syntax for closures is sugar for these three traits. Rust will generate a struct for the environment, impl the appropriate trait, and then use it."[1]

[1]Taken from the Rust Book

Closures As Arguments

- Passing closures works like function pointers.
- Let's take a (simplified) look at Rust's definition for `map`[1].

```
// self = Vec<A>
fn map<A, B, F>(self, f: F) -> Vec<B>
    where F: FnMut(A) -> B;
```

- `map` takes an argument `f: F`, where `F` is an `FnMut` trait object.
- You can pass regular functions in, since the traits line up!

[1]Real `map` coming in next lecture.

Returning Closures

- You may find it necessary to return a closure from a function.
- Unfortunately, since closures are implicitly trait objects, they're unsized!

```
fn i_need_some_closure() -> (Fn(i32) -> i32) {
    let local = 2;
    |x| x * local
}
```

```
error: the trait `core::marker::Sized` is not implemented
    for the type `core::ops::Fn(i32) -> i32 + 'static`
```

- An `Fn` object is not of constant size at compile time.
    - The compiler cannot properly reason about how much space to allocate for the `Fn`.

Returning Closures

- Okay, we can fix this! Just wrap the `Fn` in a layer of indirection and return a reference!

```
fn i_need_some_closure_by_reference() -> &(Fn(i32) -> i32) {
    let local = 2;
    |x| x * local
}
```

```
error: missing lifetime specifier
```

- Now what? We haven't given this closure a lifetime specifier...
  - The reference we're returning must outlive this function.
  - But it can't, since that would create a dangling pointer.

Returning Closures

- What's the right way to fix this? Use a `Box`!

```
fn box_me_up_that_closure() -> Box<Fn(i32) -> i32> {
    let local = 2;
    Box::new(|x| x * local)
}
```

```
error: closure may outlive the current function, but it
borrows `local`, which is owned by the current function [E0373]
```

- Augh! We were so close!
- The closure we're returning is still holding on to its environment.
  - That's bad, since once `box_me_up_that_closure` returns, `local` will be destroyed.

Returning Closures

- The good news? We already know how to fix this:

```
fn box_up_your_closure_and_move_out() -> Box<Fn(i32) -> i32> {
    let local = 2;
    Box::new(move |x| x * local)
}
```

- And you're done. It's elementary!