Structs, Enums, Collections

Rust, in Practice and in Theory
Lecture 4

CAS CS 392 (M1)

Outline

Recap **ownership** and **borrowing**

Discuss **structures**, **enumerations**, and **collections**

Look at issues of **ownership** and **borrowing** with regards to structures and enumerations

Workshop: Assignment 2

Recap: Ownership

Ownership

There are two rules:

1. Every value has exactly one **owner**

2. When the owner of a value **goes out of scope,** any memory associated with the value is **freed**

The Big Question

If we're not explicitly allocating/deallocating memory, when should it happen?

1. When it's owner goes out of scope

2. When it no longer has an owner

Who can be an owner?

```
fn foo(x : String) -> String {
    x.clone() + &x
}

fn main() {
    let y = foo(String::from("bar"));
    println!("{}", y);
}
```

Mostly **variables** and **parameters,** and **return values**

Drop

```
fn main() {
    let x = String::from("x");
}
```

When a variable goes out of scope, Rust calls a function called **drop** *on its value* to return the memory

*(It's kind of like adding **free(x)** at the end of the block)*

Drop

```
fn main() {
    let mut x = String::from("x");
    x = String::from("y");
    println!("{x}");
}
```

drop is also call when a value is replaced

The original string no longer has an owner

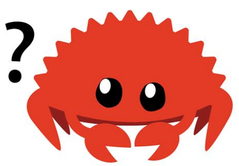
Drop

```
fn main() {
    let mut x = String::from("x");
    x = String::from("y") + &x;
    println!("{x}");
}
```

What about this case? Should we drop the String **"x"**?

Should we drop before or after evaluating the RHS of the assignment?

Move




```
fn main() {
    let x = String::from("x")
    let y = x;
    println!("{x}");
    println!("{y}");
}
```

Data on the heap must be **moved** on assignment (really, the pointer must be given up)

y owns the one copy of the string that **x** originally owned

Move

```
fn foo(mut x : String) -> String {
    x.push_str("y");
    x
}

fn main() {
    let x = String::from("x");
    let y = foo(x);
    println!("{0}", y);
}
```

Moves also happen at return values

Ownership is transferred to the parameter of **foo**, and then given to **y** from return value of **foo**

Copy

```
fn main() {
    let x = 5;
    let y = x;
    println!("{x}");
    println!("{y}");
}
```

For data on the stack, there is no memory to return. Data on stack can be **copied** on assignment

x and **y** both own a copy of the value **5**

Clone

```
fn main() {
    let x = String::from("x")
    let y = x.clone();
    println!("{x}");
    println!("{y}");
}
```

It is possible to copy data on the heap but we must explicitly call the function **clone**

y owns a *deep* copy of the string that **x** *still* owns

What's copied and what's moved?

Short answer: Stack data is copied, heap data is moved

Long answer: Everything is moved except for those types which implement the **Copy** trait

(we'll talk about traits later)

Borrowing

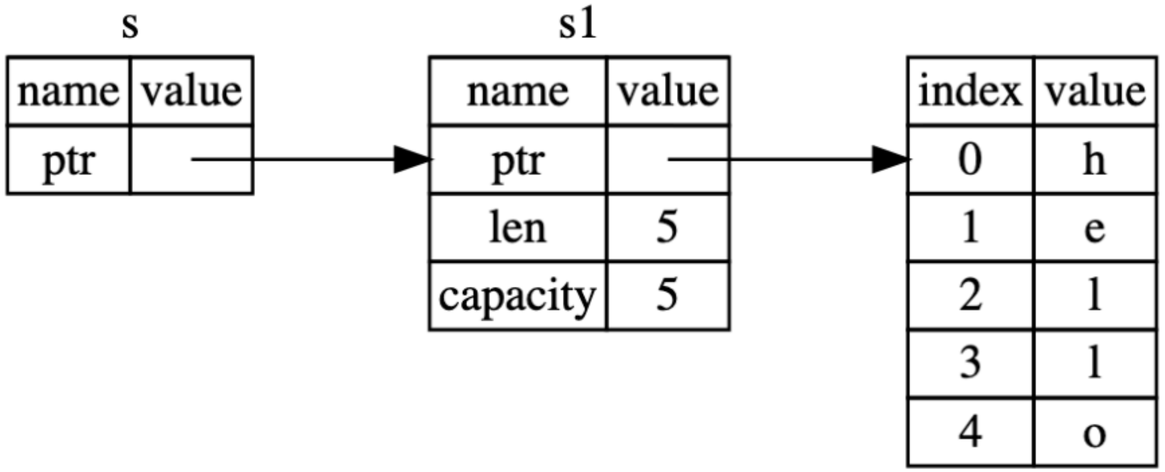
Immutable References

```
fn length(x : &String) -> i32 {
    let mut count = 0;
    for _ in x.chars() { count += 1; }
    count
}

fn main() {
    let x : String = String::from("xyz");
    let y = length(&x);
    println!("{}", y);
}
```

A **reference** is like a pointer, guaranteed to point at a valid value

The Picture



In the above picture **s** has access without taking ownership

We can have as many immutable references we want

A Note on Dereferencing

```
fn foo(x : &String) {
    let _ : &String = x;
    let _ : String = *x;
    let _ : str = **x;
}
```

It is also possible to dereference, and this looks a bit more like a pointer, but the behavior can be a bit unclear

Deref is a trait (like **Copy**) and the behavior of dereferencing can include implicit coercions

Mutable References

```
fn main() {
    let mut s = String::from("hello");

    change(&mut s);
}

fn change(some_string: &mut String) {
    some_string.push_str(", world");
}
```

Mutable references are the same, except that we're allowed to update the associated value

We can only have one mutable reference at a time

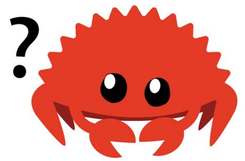
Slices

```
fn main() {
    let s = String::from("long string");
    println!("{}", &s[2..8]) // prints: ng str
}
```

Slices let you refer to a contiguous chunk of a collection like a string

They're a special kind of reference, and they follow similar rules as references

Slices and Borrowing




```
fn main() {
    let mut s = String::from("long string");
    let a : &mut str = &mut s[1..4];
    a.make_ascii_uppercase();
    let _c : &mut String = &mut s;
    println!("{}", &a)
}
```

A slice still counts a reference. We can't mutably borrow a string if someone else is borrowing a slice

Structures and Enumerations

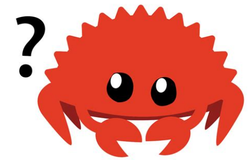
Structures

```
struct Player {
    name: String,
    score: i32,
}
```

```
let p = Player {
    name: String::from("Ash"),
    score: 0,
}
```

Structures are unordered, named, fixed-size groups of data

Field Access/Update




```
struct User {
    a: String,
    b: String,
}

fn main() {
    let mut u = User {a: "test".to_string(), b: "ing".to_string()};
    let x : String = u.a;
    u.b = String::from("er");
    println!("{}", u.a)
}
```

We can use **dot notation** to access and update fields of a structure

Accessing can move values

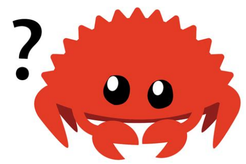
Borrowing Structure Fields

```
struct User {
    a: String,
    b: String,
}

fn main() {
    let mut u = User {a: "test".to_string(), b: "ing".to_string()};
    let x : &String = &u.a;
    let y : &mut String = &mut u.b;
    *y = String::from("er");
    println!("{}", {x})
}
```

We can have both mutable and immutable references to fields in a structure

Borrowing a Struct




```
struct User {
    a: String,
    b: String,
}

fn update(u : &mut User) {
    u.b = String::from("er")
}

fn main() {
    let mut u = User {a: "test".to_string(), b: "ing".to_string()};
    let x : &String = &u.a;
    update(&mut u);
    println!("{}", {x})
}
```

But we can't borrow overlapping parts of a structure

This works

```
struct A { b : B, i : i32}
struct B { i : i32 }

fn main() {
    let mut a = A {i: 20, b: B {i:10}};
    let n : &mut i32 = &mut a.b.i;
    let m : &mut i32 = &mut a.i;
    *n += 1;
    *m += 2;
    println!("{} {}", a.i, a.b.i);
}
```

We can have multiple mutable references to non-intersecting parts of a structure

No Partial Mutability

We can't selectively choose fields to be mutable

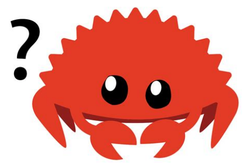
If we borrow a structure, we can mutate any part of it

```
struct U { a: i32, b: i32 }

fn update (u : &mut U) {
    u.a += 1;
    u.b -= 1;
}

fn main() {
    let mut u = U {a:0, b:0};
    update(&mut u);
    println!("{}, {}", u.a, u.b);
}
```

Structures and the Stack




```
struct List {
    head: i32,
    tail: Option<List>,
}
```

what is the size of a **List**?

Remember, unless otherwise specified, everything is put on the stack. This means structures as well

This means we can't create **recursive** structures (yet)

Aside: Derived Traits and Debug

Traits allow us to abstract behaviors of given types

Derived traits allow "obvious" traits to be implemented without any work

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let scale = 2;
    let rect1 = Rectangle {
        width: dbg!(30 * scale),
        height: 50,
    };

    dbg!(&rect1);
}
```

Methods

We can define methods and associated functions on structures

```
struct Rectangle {width: u32,height: u32}

impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
    fn square(size: u32) -> Self {
        Self {
            width: size,
            height: size,
        }
    }
}

fn main() {
    let rect1 = Rectangle {width: 30, height: 50};
    let _a = rect1.area();
    let _s = Rectangle::square(5);
}
```

Enumerations

Enumerates describe possible "shapes" (i.e., constructors) of the data

Constructors can hold (named) data

```
enum OS {
    BSD,
    MacOS(u32, u32),
    Linux {
        major: u32,
        minor: u32,
    }
}
```

Pattern Matching

```
fn supported(o : OS) -> bool {
    match o {
        OS::BSD => false,
        OS::MacOS(major, minor) => major >= 10 && minor >= 3,
        OS::Linux {major, .. }=> major >= 33,
    }
}
```

We use **match expressions** to match on enumerations

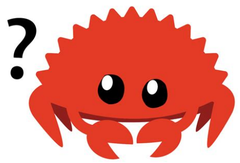
Matches must be *exhaustive*

(There are a lot of fancy pattern matching tools, use them if you want)

Enumerations and Ownership

```
enum A {
    X(String)
}

fn main() {
    let a = A::X(String::from("inner string"));
    let s = match a { A::X(s) => s };
    println!("{}", s);
    match a { A::X(s) => println!("{}", s) };
}
```

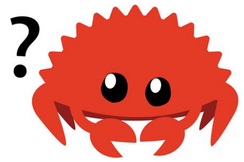


Values *can* be moved out of constructors

References and Pattern Matching

```
enum A {
    X(String, String)
}

fn main() {
    let il = String::from("left inner string");
    let ir = String::from("right inner string");
    let mut a = A::X(il, ir);
    let s : &String = match a { A::X(ref il, _) => il };
    let a_ref : &mut A = &mut a;
    println!("{}", s);
}
```



We can bind by reference during pattern matching

Options and Results

```
enum Option<T> {
    None,
    Some(T),
}
```

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

We have the usual types for dealing with errors

(along with some nice operators like **?** for working in the monad)

Collections

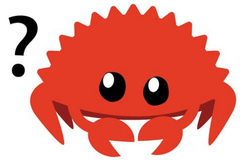
Vectors

```
let v: Vec<i32> = Vec::new();    // creating a new vector
let  mut v = vec![1, 2, 3];      // from array shorthand
v.push(5);                       // append to end
let x: Option<i32> = v.pop();    // removing from end
let x: &i32 = &v[2];             // unsafe indexing
let x: Option<&i32> = v.get(2); // safe indexing
```

A **vector** is a contiguous collection of data in memory

They have the usual methods (check the docs)

Vectors and Borrowing




```
let first = &v[0];
v.push(6);
let x = first;
```

A reference to an element in a vector counts as a borrow of the *entire* vector

(Apologies again for mixing this up in the case of slices)

Iteration

```
let mut x = 0;
for i in &v {
    x += i
}
for i in &mut v {
    *i += 10
}
```

We can iterate over vectors in the usual way

(note the dereference operator *)

Question

Can we iterate over a vector that might be updated intermittently?

Strings

```
let hello = String::from("السلام عليكم");
let hello = String::from("Dobrý den");
let hello = String::from("Hello");
let hello = String::from("שלום");
let hello = String::from("नमस्ते");
let hello = String::from("こんにちは");
let hello = String::from("안녕하세요");
let hello = String::from("你好");
let hello = String::from("Olá");
let hello = String::from("Здравствуйте");
let hello = String::from("Hola");
```

Strings are complicated...

We're not going to worry about it too much...

Hash Maps

```
use std::collections::HashMap;
let mut h : HashMap<String,i32> = HashMap::new(); // create
h.insert(String::from("ten"), 10);                // insert (moves values into h)
let x : Option<&i32> = h.get("ten");              // access (does not consume key)
```

The standard library also has hash maps with the usual interface

Note that insertion moves values whereas accessing does not

(See the docs for more examples)

Workshop: Assignment 2

Workshop

If you haven't gotten started on assignment 2, nows a good time. I'll walk around and see how everyone is doing on it.

(And take attendance)