Structures and Enumerations

Rust, in Practice and in Theory
Lecture 5

CAS CS 392 (M1)

Outline

Discuss **structures** and **enumerations**

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

Workshop: Assignment 2

Slices

```rust
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

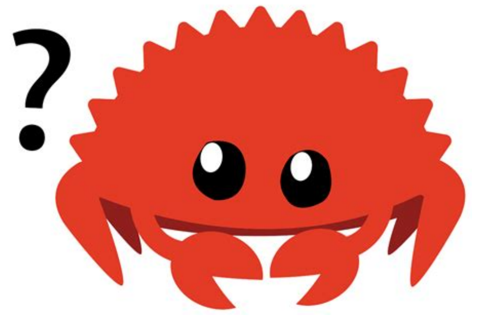
They're just a special kind of reference, and they follow similar rules as references

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 b : &mut str = &mut s[3..8];
    b.make_ascii_lowercase();
    println!("{}", &s) // prints: LONg string
}
```

You can have multiple overlapping mutable slices...

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)
}
```

But a slice still counts a reference...

(Rationale: Slices cannot move data)

Common Pattern

```
fn grab(s: &str) -> &str {
    &s[1..4]
}


fn main() {
    let s = String::from("long string");
    let a : &str = grab(&s); // a refers to s
    // cannot drop(s) because a is borrowing from s
    println!("{}", &a)
}
```

We can pass references of strings as slices, and we can return slices (there's an issue with lifetimes here, we won't get into yet)

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

This allows us to make new types for **type-driven development**

Defining Structures (Syntax)

```
<stmts>       ::= <stmt-no-sc> <stmts>
<stmt>        ::= struct <struct-ident>
<stmt-no-sc>  ::= struct <struct-ident> { <ft-pairs> }
<ft-pairs>    ::= ϵ | <ft-pair> | <ft-pair> , <ft-pairs>
<ft-pair>     ::= <var-ident> : <ty>
```

Structure identifiers must be capitalized

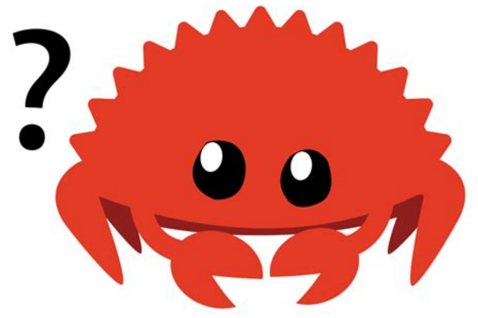
Note that we can define "unit-like" structure types with without fields (distinct syntactically from *no* fields)

Instantiating Structures (Syntax)

```
<expr>      ::= <struct-ident>
              | <struct-ident> { <fv-pairs> }
<fv-pairs> ::= ϵ | <fv-pair> | <fv-pair> , <fv-pairs>
<fv-pairs> ::= <var-ident> : <expr>
```

(just for the heck of it, I'll probably stop formalizing the syntax)

Field Access/Update



```rust
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

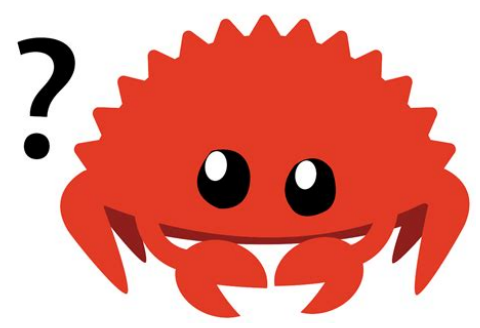
Borrowing Structure Fields

```
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



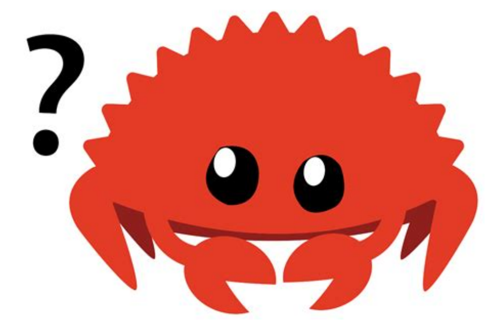

```rust
struct User {
    a: String,
    b: String,
}

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})
}
```

Borrowing a structure means borrowing *every* field

Borrowing a Struct



And this includes deep values, not just fields

(the error here is not very useful)

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

fn main() {
    let mut a = A {b: B {i:10}};
    let n : &i32 = &a.b.i;
    let a_ref : &mut A = &mut a;
    println!("{}", n);
}
```

Again, this works

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

```
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);
}
```

No Partial Mutability

We can't selectively choose fields to be mutable

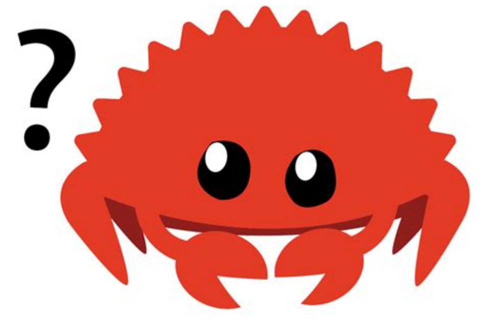
If we borrow a structure, we can mutate any part of it

```
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




```rust
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 are a meta-programming technique in which "obvious" traits can 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);
}
```

Methods and Ownership

```
                                    x = method(&x, ...)
x.method(...)        ≈                      or
                                    x = method(&mut x, ...)
```

In terms of ownership, we should think of calling a method as calling a function with a (mutable) reference

This means that methods can return references to x within itself

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

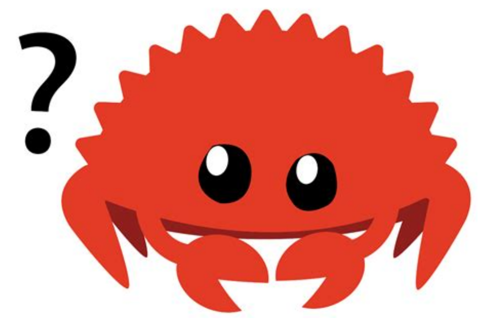
```
fn supported(o : OS) -> bool {
    match o {
        OS::BSD => false,
        OS::MacOS(major, minor) => major >= 10 && minor >= 3,
        OS::Linux {major, .. }=> major >= 33,
    }
}
```

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

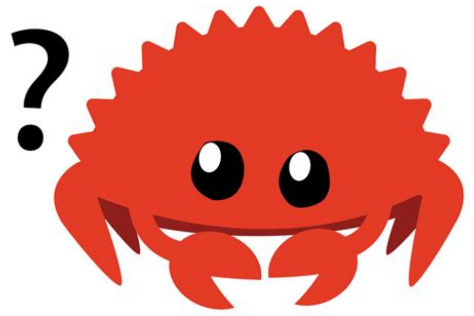


```
enum A {
    X(String)
}

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

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

```rust
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)

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)