

CMSC 330: Organization of Programming Languages

Structs, Enums in Rust

Rust Data

- So far, we've seen the following kinds of data
 - Scalar types (int, float, char, string, bool)
 - Tuples, Arrays, and Collections
- How can we build other data structures?
 - **Structs** (like Objects: support for methods)
 - <https://doc.rust-lang.org/book/ch05-00-structs.html>
 - **Enums** (like OCaml Datatypes)
 - <https://doc.rust-lang.org/book/ch06-00-enums.html>
 - **Traits** (like Java Interfaces)

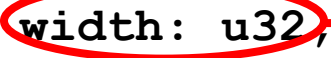

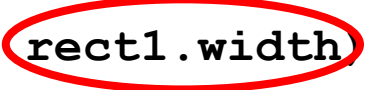
Primitive Data Conversion with `as`

```
fn main() {  
    let decimal = 65.4321_f32; //floating point number  
  
    let integer: u8 = decimal; //error: no auto-convert  
  
    // Explicit conversion  
    let integer = decimal as u8; // explicit conversion  
    let character = integer as char;  
    println!("Casting: {} -> {} -> {}",  
            decimal, integer, character);  
}
```

Casting: 65.4321 -> 65 -> A

Examples and rules at <https://doc.rust-lang.org/rust-by-example/types/cast.html>

Structs: Definitions & Construction

```
struct Rectangle {  
    width: u32,  Field with unsigned int type  
    height: u32,  
}  
  
fn main() {  
    // construction  
    let rect1 = Rectangle { width: 30, height: 50 };  Construction  
    // accessing fields  
    println!("rect1's width is {}", rect1.width);  Field accessing  
}
```

> rect1's width is 30

Aside: Construction by **Method** (more later)

```
struct Rectangle {  
    width: u32,  
    height: u32,  
}  
  
impl Rectangle { // associated methods  
    fn new(width: u32, height: u32) -> Rectangle {  
        return Rectangle{width,height}; //name match  
    }  
}  
  
fn main() {  
    let rect1 = Rectangle::new(30,50);  
    println!("rect1's width is {}", rect1.width);  
}
```

Structs: Printing

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

fn main() {
    let rect1 = Rectangle::new(30,50);
    println!("rect1 is {}", rect1);
}
```

error[E0277]: the trait bound `Rectangle: std::fmt::Display` is not satisfied

Structs: Printing via Derived Traits

```
#[derive(Debug)]
```

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

Derive **Debug** trait to support printing

```
fn main() {  
    let rect1 = Rectangle::new(30,50);  
    println!("rect1 is {:?}", rect1);  
}
```

Use printing format

```
> rect1 is Rectangle { width: 30, height: 50 }
```

Structs: Summary

- Syntax

- **struct** T [$\langle T \rangle$] { $n1:t1$, ..., $ni:ti$, }
- the ni are called **fields**, begin with a lowercase letter
- [$\langle T \rangle$] optionally for generics (see later)

- Evaluation

- **Construction:** T { $n1:v1$, ..., $ni:vi$ } is a value if vi are values
- **Access:** $t.ni$ returns the ni field of t

- Type Checking

- T { $n1:v1$, ..., $ni:vi$ } : T [if vi has type ti]

Quiz 1: `point` is immutable at *HERE*

```
struct Point {  
    x: i32,  
    y: i32,  
}  
let mut point = Point { x: 0, y: 0 };  
point.x = 5;  
let point = point;  
// HERE
```

- A. True
- B. False

Quiz 1: `point` is immutable at *HERE*

```
struct Point {  
    x: i32,  
    y: i32,  
}  
let mut point = Point { x: 0, y: 0 };  
point.x = 5;  
let point = point;  
// HERE
```

- A. True
- B. False

Mutability is a property of the binding;
the old `point`'s contents are copied to
the new one

A Note on Mutability

- A failed attempt to make a `Point` that is always mutable:

```
struct MutablePoint {  
    x: mut i32,  
    y: mut i32,  
}
```

`error: expected type, found keyword `mut``

- Mutability is a **property of the variable** that holds the `MutablePoint`, **not a property of the type itself**

Methods: Definitions on Structs

```
impl Rectangle {  
    fn area(&self) -> u32 {  
        self.width * self.height  
    }  
}
```

Self argument has type Rectangle

Self argument is a borrowed **reference** to the object

`impl Rectangle` defines an implementation block

- `self` arg has type `Rectangle` (or reference thereto)
- **Ownership rules:**
 - `&self` for read-only borrowed reference (preferred)
 - `&mut self` for read/write borrowed reference (if needed)
 - `self` for full ownership (least preferred, most powerful)

Methods: Calls

```
fn main() {  
    let rect1 = Rectangle::new(30,50);  
    println!("The area is {} pixels.", rect1.area());  
}
```

dot syntax to call methods

If method had arguments, use function call e.g.,
rect1.area(3)

Methods: Many Args, Associated Methods

```
impl Rectangle {  
  fn can_hold(&self, other &Rectangle) -> bool {  
    self.width > other.width && self.height > other.height  
  }  
  
  fn square(size:u32) -> Rectangle {  
    Rectangle { width: size, height: size }  
  }  
}
```

A reference to the Rectangle; most flexible

square is called an **associated method**

- no **self** argument
- operates on **Rectangles**
- called with **let sq = Rectangle::square(3);**

Quiz 2: What is the output

```
#[derive(Debug)]
struct Point {
    x: i32,
    y: i32,
}
impl Point {
    fn m(&mut self) {
        self.x += 1;
        self.y += 1;
    }
}
fn main() {
    let mut p = Point{ x: 0, y: 0 };
    p.m();
    println!("{:?}", p);
}
```

- A. Point { x: 1, y: 1 }
- B. Point { x: 0, y: 0 }
- C. Point { 0,0 }
- D. Point {1,1 }

Quiz 2: What is the output

```
#[derive(Debug)]
struct Point {
    x: i32,
    y: i32,
}
impl Point {
    fn m(&mut self) {
        self.x += 1;
        self.y += 1;
    }
}
fn main() {
    let mut p = Point{ x: 0, y: 0 };
    p.m();
    println!("{:?}", p);
}
```

- A. Point { x: 1, y: 1 }
- B. Point { x: 0, y: 0 }
- C. Point { 0,0 }
- D. Point {1,1 }

Generic Lifetimes

```
struct ImportantExcerpt<'a> {  
    part: &'a str,  
}  
fn main() {  
    let novel = String::from("Generic Lifetime");  
    let i = ImportantExcerpt { part: &novel; }  
}
```

- When **structs** defined to hold **references**, we need to add a **lifetime annotation** on the reference (here, **'a**)
- Lifetime is inferred for **i**, by the compiler (no need to fill it in manually); called “elision”

Lifetimes in Implementation Methods

```
struct ImportantExcerpt<'a> {  
    part: &'a str,  
}  
impl<'a> ImportantExcerpt<'a> {  
    fn level(&self) -> i32 {  
        3  
    }  
}
```

- Parameter for lifetime annotation
(would need the same for a generic
Implementation of a generic interface in Java)
- Often can be inferred

Enums: Like OCaml Datatypes

```
enum IpAddr{  
  V4(String),  
  V6(String),  
}
```

definition

construction

```
let home = IpAddr::V4(String::from("127.0.0.1"));  
let loopback = IpAddr::V6(String::from("::1"));
```

OCaml equivalent

```
type IpAddr = V4 of string | V6 of string ;;  
let home = V4 "127.0.0.1";;  
let loopback = V6 "1";;
```

Enums with Blocks

```
enum IpAddr{
    V4 (String) ,
    V6 (String) ,
}

impl IpAddr {
    fn call(&self) {
        // method body would be defined here
    }
}

let m = IpAddr::V6(String::from("::1")) ;
m.call() ;
```

Enums with Structs

Like in OCaml, enums might contain any type, e.g., structs, references, ...

```
struct Ipv4Addr {  
    // details elided  
}  
  
struct Ipv6Addr {  
    // details elided  
}  
  
enum IpAddr {  
    V4 (Ipv4Addr) ,  
    V6 (Ipv6Addr) ,  
}
```

Quiz 3: What is the output

```
#[derive(Debug)]  
enum Auth {  
    Enabled(i32),  
    Disabled(i32)  
}  
fn main() {  
    let yes = Auth::Enabled(1);  
    let no = Auth::Disabled(0);  
    println!("{:?}", yes);  
}
```

- A. Enabled(1)
- B. Disabled(0)
- C. 1
- D. 0

Quiz 3: What is the output

```
#[derive(Debug)]
enum Auth {
    Enabled(i32),
    Disabled(i32)
}
fn main() {
    let yes = Auth::Enabled(1);
    let no = Auth::Disabled(0);
    println!("{:?}", yes);
}
```

- A. Enabled(1)
- B. Disabled(0)
- C. 1
- D. 0

The Option Enum: Generic Types

Defined in standard lib

```
enum Option<T> { Some(T), None, }  
  
let some_number = Some(5);  
let some_string  = Some("a string");  
let absent_number:Option<&Rectangle>= None;
```

Instantiation with
any type!

Compare with OCaml

```
type 'a Option = Some of 'a | None ;;  
  
let some_number = Some 5 ;;  
let some_string  = Some "a string" ;;  
let absent_number : int option = None;;
```


Generics in Structs & Methods

Generic **T** in struct

```
struct Point<T> {  
    x: T,  
    y: T,  
}
```

Generic **T** in methods

```
impl<T> Point<T> {  
    fn x(&self) -> &T {  
        &self.x  
    }  
}
```

Instantiate **T** as **i32**

```
fn main() {  
    let p = Point { x:5, y:10};  
    println! ("p.x = {}", p.x());  
}
```

Matching

```
fn plus_one(x:Option<i32>) -> Option<i32> {  
  match x {  
    Some(i) => Some(i +1),  
    None => None,  
  }  
}
```

Matching should be exhaustive!

```
fn plus_one(x:Option<i32>) -> Option<i32> {  
  match x {  
    Some(i) => Some(i +1),  
    //missing None  
  }  
}
```

Error at compile time!

error[E0004]: non-exhaustive patterns:
`None` not covered

If-let, for non exhaustive matches

```
fn check(x: Option<i32>) {  
    if let Some(42) = x {  
        println!("Success!")    // only executed if the match succeeds  
    } else {  
        println!("Failure!")  
    }  
}
```

```
fn main () {  
    check(Some(3)) ;;    // prints "Failure!"  
    check(Some(42)) ;;   // prints "Success!"  
    check(None) ;;       // prints "Failure!"  
}
```

Quiz 4: Output of following code

```
enum Number {  
    Zero,  
    One,  
    Two,  
}  
use Number::Zero;  
let t = Number::One;  
match t {  
    Zero => println!("0"),  
    Number::One => println!("1"),  
}
```

- A. 0
- B. 1
- C. Compile Error

Quiz 4: Output of following code

```
enum Number {  
    Zero,  
    One,  
    Two,  
}  
use Number::Zero;  
let t = Number::One;  
match t {  
    Zero => println!("0"),  
    Number::One => println!("1"),  
}
```

- A. 0
- B. 1

C. Compile Error. Pattern `Two` not covered

Enums: Summary

- Syntax

- `enum T [<T>] { C1 [(t1)] , ... , Cn [(tn)] , }`
- the *Ci* are called constructors
 - Must begin with a capital letter; may include associated data notated with brackets `[]` to indicate it's optional

- Evaluation

- A constructor *Ci* is a value if it has no assoc. data
 - *Ci(vi)* is a value if it does
- Accessing a value of type *t* is by pattern matching
 - patterns are constructors *Ci* with data components, if any

- Type Checking

- `Ci [(vi)] : T [if vi has type ti]`

Recap: Structs and Enums

1. Structs define data structures with fields
 - And implementation blocks collect methods on to specify the behavior of structs (like objects)
2. Enums define a set of possible data types
 - Like OCaml datatypes (aka variant types)
 - Use match or if-let to deconstruct