CMSC 330: Organization of Programming Languages

Structs and 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)
  - Enums (like Ocaml Data Types)

Structs: Definitions & Construction

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

fn main() {
  // construction
  let rect1 = Rectangle { width: 30, height: 50 };
  // accessing fields
  println!("rect1's width is {}", rect1.width);
}
```

Field with int type

Construction

Field accessing

> rect1's width is 30

Structs: Printing

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

fn main() {
  letrect1 = Rectangle { width:30, height:50};
  println!("rect1 is {}", rect1);
}
```

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

Structs: Printing with Derived Traits

```
#[derive(Debug)]

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

fn main() {
  letrect1 = Rectangle { width:30, height:50};
  println!("rect1 is {:?}", rect1);
}
```

Derive printing format

Use printing format

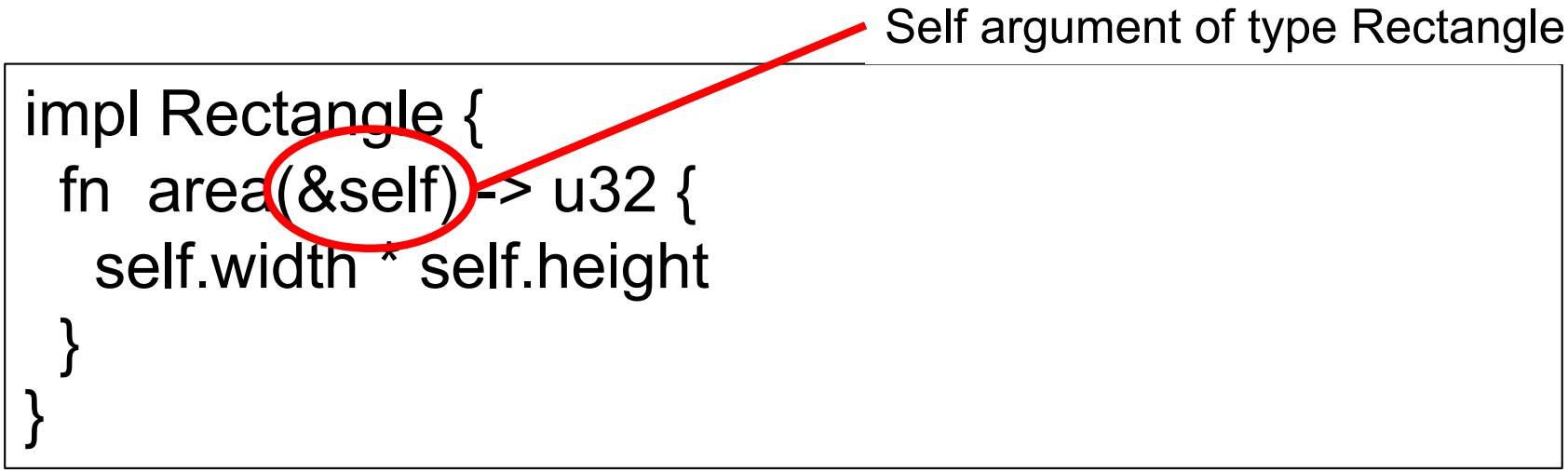
> rect1 is Rectangle { width: 30, height: 50 }

Structs

- Syntax
  - `struct T [<T>] {n1:t1, …, ni:ti,}`
  - the *ni* are called fields, begin with a lowercase letter
  - `[<T>]` optionally for generics (see later)
- Evaluation
  - **Construction:**
    T {n1:v1, ni:vi} is a value if **vi** are values.
  - **Destruction:**

    t.ni returns the **ni** field of **t**
- Type Checking
  - *T {n1:v1, ni:vi}* : *T* [if *vi* has type *ti*]

Methods: Definitions on Structs

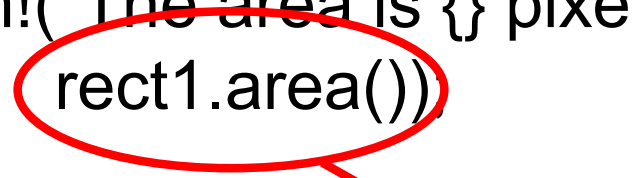


impl Rectangle defines an implementation block

- self arg has type Rectangle (reference)
- same ownership rules:
  - &self for borrowing,
  - self to take ownership,
  - &mut self to allow mutation

Methods: Calls

```
fn main() {
  let rect1 = Rectangle { width:30, height: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 Arguments

```
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 }
 }
}
```

square is called an Associated Method

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

Generic Lifetimes

```
struct ImportantExcerpt<‘a> {
  part: & ‘a str,
}

fn main() {
  let novel =String::from("Generic Lifetime");
  let i = ImportantExcerpt { part: &novel; }
}
```

When structs hold **references**, we need to add a lifetime annotation on **every** reference in the struct’s definition.

Lifetimes in Methods

```
struct ImportantExcerpt<‘a> {
  part: & ‘a str,
}

impl<‘a> ImportantExcerpt<‘a> {
  fn level(&self) -> i32 {
    3
  }
}
```

Implementation needs lifetime annotation.

Lifetime is inferred in function (using **elision**[*]).

[*] https://doc.rust-lang.org/book/second-edition/ch10-03-lifetime-syntax.html#lifetime-elision

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

Enums

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

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

definition

construction

Like Variants in Ocaml

```
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
}

enumIpAddr{
  V4(Ipv4Addr),
  V6(Ipv6Addr),
}
```

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” ;;
val absent_number :: int option ;;
let absent_number = 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),

  }
}
```

Error at compile time!

error[E0004]: non-exhaustive patterns:

`None` not covered

Enums

- 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
  - Destructing 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`***]

Quiz 2: 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 2: 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

If-let, for non exhaustive matches

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

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

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 variant types
   - Use match or if-let to deconstruct