Rust Structs and Enums

CPSC 310 - Programming Languages

Rust Data

- So far, we have seen the following kinds of data:
  - Scalar types (primitives)
  - Tuples, Arrays, and Collections
- Other data structures:
  - Structs – like Objects with support for methods
  - Enums – like OCaml variant types
  - Traits – like Java interfaces

Primitive Data Conversion with as

- Example: explicit type coercion

```
fn main() {
    let decimal = 65.23_f32; // floating point number
    let integer: u8 = decimal; // error: no implicit
                               // coercion
    let integer = decimal as u8;
    let character = integer as char;
    println!("{} -> {} -> {}",
             decimal, integer, character);
}
```

Structs

- Structs are Rust's record type

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

fn main() {
    // construction
    let r1 = Rectangle { width: 30, height: 50 };
    // accessing fields
    println!("width: {}", r1.width);
}
```

Aside: Construction by Method

- Example: add a method for construction (more on this 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 r1 = Rectangle::new(30, 50);
    println!("width: {}", r1.width);
}
```

Structs: Printing

- Rust does not know how to print arbitrary structs; we can get a string representation of a struct by deriving the `Debug` trait (more on this later), or implementing the `Display` trait

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

fn main() {
    let r1 = Rectangle::new(30, 50);
    println!("r1: {:?}", r1);
}
```

Methods: Definitions on Structs

- The `impl` keyword defines an implementation block

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

- The`self` argument is a borrowed reference to the object
- Ownership rules
  - `&self` for read-only borrowed reference
  - `&mut self` for read/write borrowed reference
  - `self` for full ownership

Methods: Calls

- Example

```
fn main() {
    let r1 = Rectangle::new(30, 50);
    println!("The area is {}" r1.area());
}
```

Methods: Many Args, Associated Methods

- Example

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

    fn square(size: u32) -> Rectangle {
        Rectangle { width: size, height: size }
    }
}
```

- `square` is called an associative method
  - no `self` argument
  - operates on `Rectangles`
  - called with `let sq = Rectangle::square(3)`

Generic Lifetimes

- Example

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

- When `structs` are defined to hold references, we need to add a lifetime annotation on the reference
- The lifetime is inferred for `i`, by the compilero

Lifetimes in Implementation Methods

- Example

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

// here the <'a> after impl is the parameter
// for a lifetime annotation
impl<'a> ImportantExcerpt<'a> {
    fn level(&self) -> i32 {
        3
    }

}
```

Enums

- Rust enums are similar to OCaml variant types

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

let home = IpAddr::V4(String::from("127.0.01"));
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

- Example

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

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

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

Enums with Structs

- Rust enums can contain any type, for example structs

```
struct Ipv4Addr {
    // details elided
}

struct Ipv6Addr {
    // details elided
}

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

The Option Enum

- The Option enum is defined in the standard library

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

let some_num = Some(5);
let some_string = Some("string");
let nothing: Option<i32> = None;
// can instantiate None with any type
```

- OCaml equivalent

```
type 'a Option = Some of 'a | None;;
let some_num = Some 5;;
let some_string = Some "string";;
let nothing : int option = None;;
```

Generics in Structs and 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`

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

Matching

- Example

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

- If the match is non-exhaustive, then the compilation fails with an error.

Non-Exhaustive Matches

- Example: if-let

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

- Example: wilcard

```
match x {
    1 => 1
    2 => 2
    _ => 0
}
```

Summary: Structs and Enums

1. Structs define data structures with fields
   - implementation blocks collect methods to specify the behavior of structs
2. Enums define a set of possible data types
   - Similar to OCaml variant types
   - Use match of if-let to deconstruct