Structured Data

CIS 198 Lecture 2

Structured Data

- Rust has two simple ways of creating structured data types:
  - Structs: C-like structs to hold data.
  - Enums: OCaml-like; data that can be one of several types.
- Structs and enums may have one or more implementation blocks (`impl`s) which define methods for the data type.

Structs

- A struct declaration:
  - Fields are declared with `name: type`.

```
struct Point {
    x: i32,
    y: i32,
}
```

- By convention, structs have `CamelCase` names, and their fields have `snake_case` names.
- Structs may be instantiated with fields assigned in braces.

```
let origin = Point { x: 0, y: 0 };
```

Structs

- Struct fields may be accessed with dot notation.
- Structs may not be partially-initialized.
  - You must assign all fields upon creation, or declare an uninitialized struct that you initialize later.

```
let mut p = Point { x: 19, y: 8 };
p.x += 1;
p.y -= 1;
```

Structs

- Structs do not have field-level mutability.
- Mutability is a property of the **variable binding**, not the type.
- Field-level mutability (interior mutability) can be achieved via `Cell` types.
  - More on these very soon.

```
struct Point {
    x: i32,
    mut y: i32, // Illegal!
}
```

Structs

- Structs are namespaced with their module name.
  - The fully qualified name of `Point` is `foo::Point`.
- Struct fields are private by default.
  - They may be made public with the `pub` keyword.
- Private fields may only be accessed from within the module where the struct is declared.

```
mod foo {
    pub struct Point {
        pub x: i32,
        y: i32,
    }
}

fn main() {
    let b = foo::Point { x: 12, y: 12 };
    //      ^~~~~~~~~~~~~~~~~~~~~~~~~~~~
    // error: field `y` of struct `foo::Point` is private
}
```

Structs

```
mod foo {
    pub struct Point {
        pub x: i32,
        y: i32,
    }

    // Creates and returns a new point
    pub fn new(x: i32, y: i32) -> Point {
        Point { x: x, y: y }
    }
}
```

- `new` is inside the same module as `Point`, so accessing private fields is allowed.

Struct `match`ing

- Destructure structs with `match` statements.

```
pub struct Point {
    x: i32,
    y: i32,
}

match p {
    Point { x, y } => println!("({}, {})", x, y)
}
```

Struct `match`ing

- Some other tricks for struct `match`es:

```rust
match p {
    Point { y: y1, x: x1 } => println!("({}, {})", x1, y1)
}

match p {
    Point { y, .. } => println!("{}", y)
}
```

- Fields do not need to be in order.
- List fields inside braces to bind struct members to those variable names.
    - Use `struct_field: new_var_binding` to change the variable it's bound to.
- Omit fields: use `..` to ignore all unnamed fields.

Struct Update Syntax

- A struct initializer can contain `.. s` to copy some or all fields from `s`.
- Any fields you don't specify in the initializer get copied over from the target struct.
- The struct used must be of the same type as the target struct.
    - No copying same-type fields from different-type structs!

```
struct Foo { a: i32, b: i32, c: i32, d: i32, e: i32 }

let mut x = Foo { a: 1, b: 1, c: 2, d: 2, e: 3 };
let x2 = Foo { e: 4, .. x };

// Useful to update multiple fields of the same struct:
x = Foo { a: 2, b: 2, e: 2, .. x };
```

Tuple Structs

- Variant on structs that has a name, but no named fields.
- Have numbered field accessors, like tuples (e.g. `x.0`, `x.1`, etc).
- Can also `match` these.

```
struct Color(i32, i32, i32);

let mut c = Color(0, 255, 255);
c.0 = 255;
match c {
    Color(r, g, b) => println!("({}, {}, {})", r, g, b)
}
```

Tuple Structs

- Helpful if you want to create a new type that's not just an alias.
  - This is often referred to as the "newtype" pattern.
- These two types are structurally identical, but not equatable.

```
// Not equatable
struct Meters(i32);
struct Yards(i32);

// May be compared using `==`, added with `+`, etc.
type MetersAlias = i32;
type YardsAlias  = i32;
```

Unit Structs (Zero-Sized Types)

- Structs can be declared to have zero size.
  - This struct has no fields!
- We can still instantiate it.
- It can be used as a "marker" type on other data structures.
  - Useful to indicate, e.g., the type of data a container is storing.

```
struct Unit;

let u = Unit;
```

Enums

- An enum, or "sum type", is a way to express some data that may be one of several things.
- Much more powerful than in Java, C, C++, C#...
- Each enum variant can have:
  - no data (unit variant)
  - named data (struct variant)
  - unnamed ordered data (tuple variant)

```rust
enum Resultish {
    Ok,
    Warning { code: i32, message: String },
    Err(String)
}
```

Enums

- Enum variants are namespaced by their enum type: `Resultish::Ok`.
    - You can import all variants with `use Resultish::*`.
- Enums, much as you'd expect, can be matched on like any other data type.

```
match make_request() {
    Resultish::Ok =>
        println!("Success!"),
    Resultish::Warning { code, message } =>
        println!("Warning: {}!", message),
    Resultish::Err(s) =>
        println!("Failed with error: {}", s),
}
```

Enums

- Enum constructors like `Resultish::Ok` and the like can be used as functions.
- This is not currently very useful, but will become so when we cover closures & iterators.

Recursive Types

- You might think to create a nice functional-style `List` type:

```
enum List {
    Nil,
    Cons(i32, List),
}
```

Recursive Types

- Such a definition would have infinite size at compile time!
- Structs & enums are stored inline by default, so they may not be recursive.
  - i.e. elements are not stored by reference, unless explicitly specified.
- The compiler tells us how to fix this, but what's a `box`?

```
enum List {
    Nil,
    Cons(i32, List),
}
// error: invalid recursive enum type
// help: wrap the inner value in a box to make it representabl
```

Boxes, Briefly

- A `box` (lowercase) is a general term for one of Rust's ways of allocating data on the heap.
- A `Box<T>` (uppercase) is a heap pointer with exactly one owner.
  - A `Box` owns its data (the `T`) uniquely-- it can't be aliased.
- `Box`es are automatically destructed when they go out of scope.
- Create a `Box` with `Box::new()`:

```rust
let boxed_five = Box::new(5);

enum List {
    Nil,
    Cons(i32, Box<List>), // OK!
}
```

- We'll cover these in greater detail when we talk more about pointers.

Methods

```
impl Point {
    pub fn distance(&self, other: Point) -> f32 {
        let (dx, dy) = (self.x - other.x, self.y - other.y);
        ((dx.pow(2) + dy.pow(2)) as f32).sqrt()
    }
}

fn main() {
    let p = Point { x: 1, y: 2 };
    p.distance();
}
```

- Methods can be implemented for structs and enums in an `impl` block.
- Like fields, methods may be accessed via dot notation.
- Can be made public with `pub`.
    - `impl` blocks themselves don't need to be made `pub`.
- Work for enums in exactly the same way they do for structs.

Methods

- The first argument to a method, named `self`, determines what kind of ownership the method requires.
- `&self`: the method *borrows* the value.
    - Use this unless you need a different ownership model.
- `&mut self`: the method *mutably borrows* the value.
    - The function needs to modify the struct it's called on.
- `self`: the method takes ownership.
    - The function consumes the value and may return something else.

Methods

```rust
impl Point {
    fn distance(&self, other: Point) -> f32 {
        let (dx, dy) = (self.x - other.x, self.y - other.y);
        ((dx.pow(2) + dy.pow(2)) as f32).sqrt()
    }

    fn translate(&mut self, x: i32, y: i32) {
        self.x += x;
        self.y += y;
    }

    fn mirror_y(self) -> Point {
        Point { x: -self.x, y: self.y }
    }
}
```

- `distance` needs to access but not modify fields.
- `translate` modifies the struct fields.
- `mirror_y` returns an entirely new struct, consuming the old one.

Associated Functions

```rust
impl Point {
    fn new(x: i32, y: i32) -> Point {
        Point { x: x, y: y }
    }
}

fn main() {
    let p = Point::new(1, 2);
}
```

- Associated function: like a method, but does not take `self`.
    - This is called with namespacing syntax: `Point::new()`.
        - Not `Point.new()`.
    - Like a "static" method in Java.
- A constructor-like function is usually named `new`.
    - No inherent notion of constructors, no automatic construction.

Implementations

- Methods, associated functions, and functions in general may not be overloaded.
  - e.g. `Vec::new()` and `Vec::with_capacity(capacity: usize)` are both constructors for `Vec`
- Methods may not be inherited.
  - Rust structs & enums must be composed instead.
  - However, traits (coming soon) have basic inheritance.

Patterns

- Use `...` to specify a range of values. Useful for numerics and `char`s.
- Use `_` to bind against any value (like any variable binding) and discard the binding.

```
let x = 17;

match x {
    0 ... 5 => println!("zero through five (inclusive)"),
    _ => println!("You still lose the game."),
}
```

`match`: References

- Get a reference to a variable by asking for it with `ref`.

```
let x = 17;

match x {
    ref r => println!("Of type &i32: {}", r),
}
```

- And get a mutable reference with `ref mut`.
    - Only if the variable was declared `mut`.

```
let mut x = 17;

match x {
    ref r if x == 5 => println!("{}", r),
    ref mut r => *r = 5
}
```

- Similar to `let ref`.

`if-let` Statements

- If you only need a single match arm, it often makes more sense to use Rust's `if-let` construct.
- For example, given the `Resultish` type we defined earlier:

```
enum Resultish {
    Ok,
    Warning { code: i32, message: String },
    Err(String),
}
```

`if-let` Statements

- Suppose we want to report an error but do nothing on `Warning`s and `Ok`s.

```
match make_request() {
    Resultish::Err(_) => println!("Total and utter failure."),
    _ => println!("ok."),
}
```

- We can simplify this statement with an `if-let` binding:

```
let result = make_request();

if let Resultish::Err(s) = result {
    println!("Total and utter failure: {}", s);
} else {
    println!("ok.");
}
```

`while-let` Statement

- There's also a similar `while-let` statement, which works like an `if-let`, but iterates until the condition fails to match.

```
while let Resultish::Err(s) = make_request() {
    println!("Total and utter failure: {}", s);
}
```

Inner Bindings

- With more complicated data structures, use `@` to create variable bindings for inner elements.

```
#[derive(Debug)]
enum A { None, Some(B) }
#[derive(Debug)]
enum B { None, Some(i32) }

fn foo(x: A) {
    match x {
        a @ A::None              => println!("a is A::{:?}", a
        ref a @ A::Some(B::None) => println!("a is A::{:?}", *a
        A::Some(b @ B::Some(_))  => println!("b is B::{:?}", b
    }
}

foo(A::None);             // ==> x is A::None
foo(A::Some(B::None));    // ==> a is A::Some(None)
foo(A::Some(B::Some(5))); // ==> b is B::Some(5)
```

Lifetimes

- There's one more piece to the ownership puzzle: Lifetimes.
- Lifetimes generally have a pretty steep learning curve.
  - We may cover them again later on in the course under a broader scope if necessary.
- Don't worry if you don't understand these right away.

Lifetimes

- Imagine This:
    1. I acquire a resource.
    2. I lend you a reference to my resource.
    3. I decide that I'm done with the resource, so I deallocate it.
    4. You still hold a reference to the resource, and decide to use it.
    5. You crash 😿.
- We've already said that Rust makes this scenario impossible, but glossed over how.
- We need to prove to the compiler that *step 3* will never happen before *step 4*.

Lifetimes

- Ordinarily, references have an implicit lifetime that we don't need to care about:

```
fn foo(x: &i32) {
// ...
}
```

- However, we can explicitly provide one instead:

```
fn bar<'a>(x: &'a i32) {
// ...
}
```

- `'a`, pronounced "tick-a" or "the lifetime *a*" is a *named* lifetime parameter.
  - `<'a>` declares generic parameters, including lifetime parameters.
  - The type `&'a i32` is a reference to an `i32` that lives at least as long as the lifetime `'a`.

Lifetimes

- The compiler is smart enough not to need `'a` above, but this isn't always the case.
- Scenarios that involve multiple references or returning references often require explicit lifetimes.
  - Speaking of which...

Multiple Lifetime Parameters

```
fn borrow_x_or_y<'a>(x: &'a str, y: &'a str) -> &'a str;
```

- In `borrow_x_or_y`, all input/output references all have the same lifetime.
    - `x` and `y` are borrowed (the reference is alive) as long as the returned reference exists.

```
fn borrow_p<'a, 'b>(p: &'a str, q: &'b str) -> &'a str;
```

- In `borrow_p`, the output reference has the same lifetime as `p`.
    - `q` has a separate lifetime with no constrained relationship to `p`.
    - `p` is borrowed as long as the returned reference exists.

Lifetimes

- Okay, great, but what does this all mean?
  - If a reference `R` has a lifetime `'a`, it is *guaranteed* that it will not outlive the owner of its underlying data (the value at `*R`)
  - If a reference `R` has a lifetime of `'a`, anything else with the lifetime `'a` is *guaranteed* to live as long `R`.
- This will probably become more clear the more you use lifetimes yourself.

Lifetimes - `structs`

- Structs (and struct members) can have lifetime parameters.

```
struct Pizza(Vec<i32>);
struct PizzaSlice<'a> {
    pizza: &'a Pizza,  // <- references in structs must
    index: u32,        //    ALWAYS have explicit lifetimes
}

let p1 = Pizza(vec![1, 2, 3, 4]);
{
    let s1 = PizzaSlice { pizza: &p1, index: 2 }; // this is o
}

let s2;
{
    let p2 = Pizza(vec![1, 2, 3, 4]);
    s2 = PizzaSlice { pizza: &p2, index: 2 };
    // no good - why?
}
```

Lifetimes - `structs`

- Lifetimes can be constrained to "outlive" others.
  - Same syntax as type constraint: `<'b: 'a>`.

```
struct Pizza(Vec<i32>);
struct PizzaSlice<'a> { pizza: &'a Pizza, index: u32 }
struct PizzaConsumer<'a, 'b: 'a> { // says "b outlives a"
    slice: PizzaSlice<'a>, // <- currently eating this one
    pizza: &'b Pizza,      // <- so we can get more pizza
}

fn get_another_slice(c: &mut PizzaConsumer, index: u32) {
    c.slice = PizzaSlice { pizza: c.pizza, index: index };
}

let p = Pizza(vec![1, 2, 3, 4]);
{
    let s = PizzaSlice { pizza: &p, index: 1 };
    let mut c = PizzaConsumer { slice: s, pizza: &p };
    get_another_slice(&mut c, 2);
}
```

Lifetimes - `'static`

- There is one reserved, special lifetime, named `'static`.
- `'static` means that a reference may be kept (and will be valid) for the lifetime of the entire program.
    - i.e. the data referred to will never go out of scope.
- All `&str` literals have the `'static` lifetime.

```
let s1: &str = "Hello";
let s2: &'static str = "World";
```

Structured Data With Lifetimes

- Any struct or enum that contains a reference must have an explicit lifetime.
- Normal lifetime rules otherwise apply.

```
struct Foo<'a, 'b> {
  v: &'a Vec<i32>,
  s: &'b str,
}
```

Lifetimes in `impl` Blocks

- Implementing methods on `Foo` struct requires lifetime annotations too!
- You can read this block as "the implementation using the lifetimes `'a` and `'b` for the struct `Foo` using the lifetimes `'a` and `'b`."

```
impl<'a, 'b> Foo<'a, 'b> {
  fn new(v: &'a Vec<i32>, s: &'b str) -> Foo<'a, 'b> {
    Foo {
      v: v,
      s: s,
    }
  }
}
```