ECE326

PROGRAMMING LANGUAGES

Lecture 15 : Structures and Generics

Kuei (Jack) Sun

ECE

University of Toronto

Fall 2020

Structure

- Syntax
  - struct *structname* { *fieldname*: *type*, *fieldname*: *type*, ... }
- Examples

```
struct Point { x: f32, y: f32, }
struct Rectangle { p1: Point, p2: Point, }
```

Field separated by comma! Definition does not end with semicolon!

- Instantiation

```
let p: Point = Point { x: 0.3, y: 0.4 };
let rect = Rectangle {
      p1: Point { x: 0.2, y: 0.5 },
      p2: p, // struct can be owners; p is moved into rect
};
```

Destructuring

- Structures can also be destructured

```
let p: Point = Point { x: 0.3, y: 0.4, z: 0.5 };

// destructure a structure
let Point { x: my_x, y: my_y, z: _ } = p;

// destructure just one of the fields
let Point { z: my_z, .. } = p;

println!("my 2D point at ({}, {})", my_x, my_y);
println!("z = {}", my_z);
```

```
my point at (0.3, 0.4)
z = 0.5
```

Methods

- Method definitions are separate from its structure
- Define methods within an *impl* block

```
// Implementation block, all Point methods go in here
impl Point {
      fn origin() -> Point {
            Point { x: 0.0, y: 0.0 }
      }
      fn new(x: f64, y: f64) -> Point {
            Point { x: x, y: y }
      }
      fn translate(&mut self, x: f64, y: f64) {
            self.x += x;
            self.y += y;
      }
}
```

origin and new are *static methods*.

translate is an *instance method*.

The type of *self* is Point and does not need to be specified.

Methods

```
impl Rectangle {
        fn perimeter(&self) -> f64 {
                let Point { x: x1, y: y1 } = self.p1;
                let Point { x: x2, y: y2 } = self.p2;
                2.0 * ((x1 - x2).abs() + (y1 - y2).abs())
        }
        fn translate(&mut self, x: f64, y: f64) {
                self.p1.translate(x, y);
                self.p2.translate(x, y);
        }
}

let mut square = Rectangle {
        p1: Point::origin(), p2: Point::new(1.0, 1.0),
};

println!("my perimeter is: {}", square.perimeter());
square.translate(2.0, 3.0);                 // OK – square mutable
```

Ownership

- Instance methods should always borrow self
  - Except for a “destructor”, i.e. instance destroyed afterwards

```
impl Rectangle {
      fn destroy(self) {
            println!("Destroying Rectangle");
      }
}
let rect = Rectangle{
      p1: Point::new(.1, .2), p2: Point::new(.3, .4)
};
rect.destroy();
// cannot use rect after this point in code
```

Generic Programming

- Making minimal assumption about structure of data
  - Maximize code reuse across different data types
- Problem with statically typed languages
  - Redundant implementation of common algorithm

```
int min(int a, int b) {
    return a < b ? a : b;
}
```

```
/* assumed operator< implemented */
Complex min(Complex a, Complex b) {
    return a < b ? a : b;
}
```

These two functions looks exactly the same, but you have to write them twice without generic programming because their arguments have different types.

Parametric Polymorphism

- Handling of values without depending on their types
  - E.g. concatenating lists without knowing type of element
- Also known as *Generics*
  - Can be applied to functions and types
- Type parameter
  - Identifier that specifies a generic type name (e.g. T)
- Parameterized type
  - The instantiation of a generic type with actual type arguments

Rust Generics

Generic Type

type parameter

parameterized type

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

fn main() {
    let integer : Point<i32> = Point { x: 5, y: 10 };
    let float = Point { x: 1.0, y: 4.0 };

    // error: x and y must be of same type
    let wont_work = Point { x: 5, y: 4.0 };
    let c = mixer(&integer, &Point{x: 2, y: 3});
    …
}

fn mixer<T: Copy>(a: &Point<T>, b: &Point<T>)
    -> Point<T> {
   Point::<T> { x: a.x, y: b.y }
}
```

Rust can usually deduce type argument from variable initialization (This is a Point<f64>)

Generic Function

Generic Programming

- Language Implementation for Generics
  - Shared Generics
    - Using generic pointers and callback functions
      - E.g. C (without using the preprocessor), Java
    - All parameterized types share the same compiled code
  - Metaprogramming
    - Writing program that generates code
      - E.g. C++ template, Rust generics
    - Each parameterized type has its own generated code
    - Abuse of metaprogramming may result in large executable size

Generic Methods

- If a type is generic, its methods must also be

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

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

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

Specialization

- *Impl* block can be specialized for a concrete type
- Only the concrete type will receive the extra methods

```
impl Point<f32> {
      fn distance_from_origin(&self) -> f32
      {
            (self.x.powi(2) + self.y.powi(2)).sqrt()
      }
}

let p = Point { x: 2.5, y: 3.5 };
let dist = p.distance_from_origin();
println!("distance from origin: {}", dist);
```

Nested Generics

- The methods of a generic type can itself be generic

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

impl<T, U> Point<T, U> {
    fn mixup<V, W>(self, other: Point<V, W>) -> Point<T, W> {
        Point { x: self.x, y: other.y, }
    }
}

let p1 = Point { x: 5, y: 10.4 };
let p2 = Point { x: "Hello", y: 'c'};
let p3 = p1.mixup(p2);
println!("p3.x = {}, p3.y = {}", p3.x, p3.y);
// p3.x = 5, p3.y = c
```

Trait Bound

- Constrain type parameters to have certain behaviours

```
fn largest<T>(list: &[T]) -> T
{
        let mut largest = list[0];
        for &item in list {
                if item > largest {
                        largest = item;
                }
        }
        largest
}
```

&[T] is an immutable array slice with element type T

binary operation `>` cannot be applied to type `T
note: an implementation of `std::cmp::PartialOrd` might be missing for `T`

Trait Bound

- Constrain type parameters to have certain behaviours

```rust
fn largest<T: PartialOrd>(list: &[T]) -> T
{
        let mut largest = list[0];
        for &item in list {
                if item > largest {
                        largest = item;
                }
        }
        largest
}
```

cannot move out of type `[T]`, a non-copy slice

In Rust, only primitive types can be copied by default. list[0] attempts to move ownership of element to *largest*.

Trait Bound

- Constrain type parameters to have certain behaviours

```
fn largest<T: PartialOrd + Copy>(list: &[T]) -> T
{
      let mut largest = list[0];
      for &item in list {
            if item > largest {
                  largest = item;
            }
      }
      largest
}
```

- Now only accepts types that can be compared and copied

Traits

- Shared behaviours across types
- Can have default implementation
  - Unlike Java interface
- Similar to mixin, but cannot define member variables
- Can depend on other traits

```
pub trait Summary {
	fn summarize(&self) -> String;
}
```

Printing

```
use std::fmt; // Debug can be auto-generated

#[derive(Debug)]
struct MyType { x: u32, y: u32 } // but not Display

impl fmt::Display for MyType {
      fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
            write!(f, "x={}, y={}", self.x, self.y)
      }
}

let t = MyType{ x:1, y:2};

println!("{}", t);   // x=1, y=2
println!("{:?}", t); // MyType { x: 1, y: 2 }
```

Printing

```
use std::fmt; // Debug can be auto-generated

#[derive(Debug)]
struct MyType { x: u32, y: u32 } // but not Display

impl fmt::Display for MyType {
                          t fmt::Formatter) -> fmt::Result {
                              ={}", self.x, self.y)
        }
}

let t = MyType{ x:1, y:2};

println!("{}", t);    // x=1, y=2
println!("{:?}", t); // MyType { x: 1, y: 2 }
```

Annotation to use default implementation of the Debug trait

Printing

```rust
use std::fmt; // Debug can be auto-generated

#[derive(Debug)]
struct MyType { x: u32, y: u32 } // but not Display

impl fmt::Display for MyType {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "x={}, y={}", self.x, self.y)
    }
}

let t = MyType{ x:1, y:2};

println!("{}", t);    // x=1, y=2
println!("{:?}", t); // MyType { x: 1, y: 2 }
```

Explicit implementation of the Display trait for MyType

Making Copies

- Copy trait
  - Any move ownership now makes a full copy instead
- Clone trait
  - More explicit, requires calling clone() method

```
#[derive(Debug,Clone,Copy)]
struct Point { x: f32, y: f32, z: f32 }

let p = Point{x: 1., y: 2., z: 3. };

let q = p.clone();        // Clone
let r = p;                // Copy
```

Traits

- You can add traits to existing types, even primitives

```
use std::convert::TryInto;

trait Tetration {
    // Self is the type of self (e.g. i64)
    fn tetration(&self, n: i32) -> Self;
}

impl Tetration for i64 {
    fn tetration(&self, n: i32) -> i64 {
        if n == 0 { 1 }
        else {
            self.pow(self.tetration(n-1).try_into().unwrap())
        }
    }
}

let v = 3_i64.tetration(3);              // 3_i64 is of type i64
println!("tet({}, {}) = {}", 3, 3, v); // tet(3, 3) = 7625597484987
```

TryInto is a trait with default implementation that raises an error if conversion causes an integer overflow