

# ECE326

## PROGRAMMING LANGUAGES

### **Lecture 31 : Structures and Generics**

Kuei (Jack) Sun

ECE

University of Toronto

Fall 2019

# Structure

- Syntax

- *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: point,  
};
```

# Destructuring

- Structures can also be destructured

```
let p: Point = Point { x: 0.3, y: 0.4 };  
let pair = (1, 0.1);
```

```
// destructure a structure
```

```
let Point { x: my_x, y: my_y } = p;
```

```
// destructure a tuple
```

```
let Pair(integer, decimal) = pair;
```

```
println!("my point at ({}, {})", my_x, my_y);
```

```
println!("pair contains {:?} and {:?}", integer, decimal);
```

# 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),  
};  
  
let len = square.perimeter();  
println("my perimeter is: {}", len);  
square.translate(2.0, 3.0);           // OK - square mutable
```

# Ownership

- Recall that `&` operator means borrow
- 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
```

# Generics

- Enables generic programming
- Simpler than C++ template programming
  - Only data types are allowed in type parameter
- Example

```
struct Point<T> { x: T, y: T, }  
  
fn main() {  
    let integer = 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 };  
}
```

# Generics

- 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
  - Similar to C++ templates
- 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.iter() {
        if item > largest {
            largest = item;
        }
    }
    largest
}
```

&[T] is an  
immutable array  
slice (similar to  
pointer to an array)

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

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

- Now only accepts types that can be compared and copied

# Traits

- Shared behaviours across types
- Similar to mixin, but cannot define member variables
- Can have default implementation
  - Unlike interface in Java (also known as protocol)
- Can depend on other traits

```
// pub means public - can be used by other modules
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 {  
    Annotation to use default  
    implementation of the Debug trait = { }", 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 {  
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {  
        write!(f, "x={}, y={}", self.x, self.y)  
    }  
}
```

Explicit implementation of the  
Display trait for MyType

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

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

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

# Making Copies

- Copy trait
  - Any move ownership now makes a bitwise 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
```