ECE326

PROGRAMMING LANGUAGES

Lecture 17b : Rust Built-in Traits

Kuei (Jack) Sun

ECE

University of Toronto

Fall 2020

Trait

- A collection of methods for an unknown type
- Type that implements a trait can use its methods
  - Especially useful if the trait has default implementation
    - E.g., Clone, Copy, Debug
- Helps define shared behaviour abstractly
- Rust has many built-in traits
- Operator overloading
  - A trait for every operator that can be overloaded

Relational Operator

- PartialEq, Eq, PartialOrd, Ord
  - All have default implementation
- PartialEq and Eq
  - Enables == and !=
- PartialOrd and Ord
  - Enables all 6 relational operators
  - Default implementation uses lexicographical order
- Lexicographical order
  - Used in dictionaries and encyclopedias

Relational Operator

- Partial Order
  - Not reflexive: does not satisfy *a* == *a* for all *a* in *type*
    - e.g. for float, NaN != NaN is true
- Total Order satisfies reflexivity
- PartialOrd and Eq
  - Requires PartialEq
- Ord
  - Requires PartialOrd and Eq
  - Required for sorting

NaN: not a number

Relation Operator

```
#[derive(PartialEq, Eq, PartialOrd, Ord, Debug)]
struct Point { x: i32, y: i32, }

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

let p1 = Point{ x: 1, y: 5 };
let p2 = Point{ x: 3, y: 0 };
let p3 = Point{ x: 1, y: 9 };
let mut v = vec![ p1, p2, p3 ];
v.sort();
println!("largest in {:?}: {:?}", v, largest(&v)); }
```

Copy trait not needed because largest returns &T

By lexicographical order, p1 is smallest

Compound Assignment

- AddAssign, SubAssign, DivAssign, MulAssign, etc
  - Must import from std::ops
  - Does not have default implementation

```
use std::ops::AddAssign;

impl AddAssign for Point {
    fn add_assign(&mut self, other: Self) {
        *self = Self { x: self.x + other.x,
                       y: self.y + other.y, };
    }
}

let mut point = Point { x: 1, y: 0 };
point += Point { x: 2, y: 3 };
println!("point = {:?}", point); // point = Point { x: 3, y: 3 }
```

Generic Trait

- A trait that has a generic type parameter
- Arithmetic Operators
  - Add, Sub, Div, Mul, etc
  - Must declare output type

The type of the operator's return value is also T

```
use std::ops::Mul;
struct Rectangle<T> { width: T, height: T }

impl<T: Mul<Output=T> + Copy> Rectangle<T> {
    fn area(&self) -> T { self.width * self.height }
}

let r = Rectangle { width: 5.4, height: 3.8 };
println!("area = {}", r.area()); // area = 20.52
```

Default

- Default value of a type
  - 0 for integers, empty string for String
- Has default implementation

```
fn total<T: Default + AddAssign + Copy>(list: &[T]) -> T {
    let mut sum = Default::default();
    for &item in list { sum += item; }
    sum
}

let v = vec![3, 2, 5];
let empty: Vec<f32> = Vec::new();
println!("{}", total(&v));           // 10
println!("{}", total(&empty));       // 0.
```

From

- Convert from one type to another
  - Automatically implements the Into trait for the other type

```
#[derive(Debug)]
enum Answer { Yes, No, Maybe }
impl From<& str> for Answer {
    fn from(s: & str) -> Self {
        match s.to_lowercase().as_str() {
            "yes" => Answer::Yes,
            "no" => Answer::No,
            _ => Answer::Maybe,
        }
    }
}

println!("{:?}", Answer::from("yes")); // Yes
let ans: Answer = "NO".into();
println!("{:?}", ans);                // No
```