

ECE326

PROGRAMMING LANGUAGES

Lecture 19 : Traits and Iterators

Kuei (Jack) Sun

ECE

University of Toronto

Fall 2020

Trait

- A collection of methods for an unknown type
 - Trait refers to the type that implements it as *Self*
- Type that implements a trait can use its methods
 - Especially useful if the trait has default implementation
- Helps define shared behaviour abstractly
- Example

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

Example

```
pub struct NewsArticle {
    pub headline: String,      pub location: String,
    pub author: String,        pub content: String,
}
impl Summary for NewsArticle {
    fn summarize(&self) -> String {
        format!("{}", by {}, self.headline, self.author)
    }
}
pub struct Tweet {
    pub username: String,      pub content: String,
    pub reply: bool,           pub retweet: bool,
}
impl Summary for Tweet {
    fn summarize(&self) -> String {
        format!("{}", self.username, self.content)
    }
}
```

Example

```
pub struct Tweet {  
    pub username: String,    pub content: String,  
    pub reply: bool,        pub retweet: bool,  
}  
  
impl Summary for Tweet {  
    fn summarize(&self) -> String {  
        format!("{}", self.username, self.content)  
    }  
}  
  
let tweet = Tweet {  
    username: String::from("horse_ebooks"),  
    content: String::from("of course, as you probably already \  
                           know, people"),  
    reply: false,  
    retweet: false,  
};  
  
println!("1 new tweet: {}", tweet.summarize());
```

Trait Object

- Rust's equivalent of abstract base class
- Allows for runtime polymorphism
- Use `dyn` keyword to use objects as trait objects
 - Must be placed inside a `Box<T>`

```
fn random_animal(random_number: f64) -> Box<dyn Animal> {  
    if random_number < 0.5 {  
        Box::new(Sheep {})  
    } else {  
        Box::new(Cow {})  
    }  
}  
  
fn main() {  
    let animal = random_animal(0.234);  
    println!("It says {}", animal.noise());  
}
```

```
trait Animal {  
    fn noise(&self)  
        -> &'static str;  
}
```

Generic Traits

- A trait that takes type parameter
- Works the same as other generics
 - Can have trait bounds

```
trait Out<T> {  
    fn write(&mut self, value: T);  
}  
  
impl Out<i64> for ByteArray {  
    fn write(&mut self, value: i64) {  
        self.pointer += mem::size_of::<i64>();  
        let bytes = value.to_be_bytes();  
        self.buffer.extend_from_slice(&bytes);  
    }  
}
```

Return Type Polymorphism

- Calls different trait method depending on the type of the variable the method's return value is assigned to
 - Type inference does not work in this case
 - C++ does not support this

```
trait In<T> : Buffer {  
    fn from_raw(&mut self) -> T;  
}
```

```
impl In<i32> for ByteArray { ... }  
impl In<i64> for ByteArray { ... }
```

```
// calls ByteArray::In<i32>::from_raw. must specify type here  
let numcols: i32 = bytearray.from_raw();
```

This means the trait `In<T>` requires the trait `Buffer` to also be implemented.

where

- Allows specifying trait bounds more expressively

```
impl <A: TraitB + TraitC, D: TraitE> MyTrait<A, D> for YourType {}
```

```
// Expressing bounds with a `where` clause
```

```
impl <A, D> MyTrait<A, D> for YourType where A: TraitB + TraitC,  
      D: TraitE {}
```

- Can specify bounds that contains the type parameter

```
trait PrintInOption {  
    fn print_in_option(self);  
}  
impl<T> PrintInOption for T where Option<T>: Debug {  
    fn print_in_option(self) {  
        println!("{}", Some(self));  
    }  
}
```

“Option<T>: Debug”
is the trait bound
because that is
what's being printed.

Associated Type

- Defines generic types as internal types
 - And not as parameters

- Before:

```
trait Contains<A, B> {  
    fn contains(&self, _: &A, _: &B) -> bool;  
}  
fn difference<A, B, C>(container: &C) -> i32  
    where C: Contains<A, B> {  
    container.last() - container.first()  
}
```

- After

```
trait Contains {  
    type A;          type B;  
    fn contains(&self, &Self::A, &Self::B) -> bool;  
}
```

Explicitly requires
A and B as type
parameters for
generic structures
and functions

Associated Type

- Using a trait with associated types

```
impl Contains for Container {  
    type A = i32;  
    type B = i32;  
  
    // `&Self::A` and `&Self::B` are also valid here.  
    fn contains(&self, number_1: &i32, number_2: &i32) -> bool {  
        (&self.0 == number_1) && (&self.1 == number_2)  
    }  
  
    fn first(&self) -> i32 { self.0 }  
    fn last(&self) -> i32 { self.1 }  
}  
  
fn difference<C: Contains>(container: &C) -> i32 {  
    container.last() - container.first()  
}
```

```
/* named tuple */  
struct Container(i32, i32);
```

Operator Overloading

- There's a trait for every operator that can be overloaded

```
use std::ops;

struct Foo; // Unit-like struct:
struct Bar; // There's only one value struct FooBar;

// This implements Foo + Bar = FooBar
impl ops::Add<Bar> for Foo {
    type Output = FooBar; // Output is an associated type

    fn add(self, _rhs: Bar) -> FooBar {
        FooBar
    }
}
```

Drop trait

- Same as a destructor in C++
- Use if your structure does something special upon drop
 - Unlikely unless it's a low level construct
- `drop()` function
 - Deletes object immediately

```
struct Droppable { name: &'static str, }
impl Drop for Droppable {
    fn drop(&mut self) { println!("> Dropping {}", self.name); }
}

fn main() {
    let a = Droppable { name: "a" };
    drop(a); // `a` deleted here
            // instead of here
}
```

Iterator

- An object which performs the act of iterating
 - An agent which operates on an *iterable*
- Iterable
 - An object that can be iterated (e.g. container such as list)
- Stream
 - Sequence of data made available over time
 - Can have potentially infinite data
 - Example
 - Network connection, Rust range: (x..y), Python range(x, y)

Iterator

- Two requirements
 1. A way to retrieve the next element
 2. A way to signal end of iteration
- Python iterator
 - `iter()` built-in function

```
numbers = [2, 3, 5, 7]
it = iter(numbers)
while True:
    try:
        print(next(it))
    except StopIteration:
        print("End of List")
        break
```

`next()` function will retrieve the next element from iterator, or raise `StopIteration`

Output:

```
2
3
5
7
End of List
```

Iterator trait

- Rust iterator implements Iterator trait

```
use std::ops::Add;
struct Fibonacci<T> { curr: T, next: T, }
impl<T: Copy + Add<Output=T>> Iterator for Fibonacci<T> {
    type Item = T;
    fn next(&mut self) -> Option<T> {
        let new_next = self.curr + self.next;
        self.curr = self.next;
        self.next = new_next;
        Some(self.curr)
    }
}

fn fibonacci() -> Fibonacci<u32> {
    Fibonacci { curr: 0, next: 1 }
}

for i in fibonacci() { println!("{}", i); } // infinite loop
```

Rust uses Option to determine when iteration ends (i.e., when None is returned)

Iterators are used by for loop automatically

IntoIterator trait

- Containers are *iterables*, not iterators
- But for loop requires an *iterator*
- `into_iter()` function turns containers into iterators
- for loop is just an syntactic sugar

```
let v = vec![2, 3, 5, 7];
let mut iter = v.iter();
loop {
    match iter.next() {
        Some(x) => { /* body of for loop */ },
        None => break,
    }
}
```

```
for x in v {
    /* body */
}
```


Iterator Adapters

- Enjoyed by functional programmers
- Operates on iterators
- Example: Sieve of Eratosthenes

```
let starter: Vec<i32> = vec![2, 3, 5, 7];
let largest = 50;
let composites = starter.iter().map(|&x| -> Vec<i32> {
    ((x+1)..largest).filter(|&y| y % x == 0).collect()
}).flatten().collect::
```

Sieve of Eratosthenes

```
let starter: Vec<i32> = vec![2, 3, 5, 7];
let largest = 50;

// each integer in starter was mapped into a vector of i32
let composites = starter.iter().map(|&x| -> Vec<i32> {
    ((x+1)..largest).filter(|&y| y % x == 0).collect()
}).collect::();
println!("{}", composites);
```

```
[
  [4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36,
    38, 40, 42, 44, 46, 48],
  [6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48],
  [10, 15, 20, 25, 30, 35, 40, 45],
  [14, 21, 28, 35, 42, 49]
]
```

Iterator Adapters

- `map(closure)`: transforms each element
 - Can even return a different type
- `filter(closure)`: keeps element if closure returns true
- `collect()`: collects elements in iterator into container
 - With ambiguous integers, must specify type
- `flatten()`: turns nested vectors into a flattened vector
- `take(n)`: only iterate up to n times
- `skip(n)`: skip the first n iterations

fold

- Known as `reduce()` in Python
- `fold(accumulator, closure)`
 - Accumulator: an aggregate value of a collection
 - E.g. sum, max, min, average, etc.
 - The argument is the initial value of the accumulator
 - Closure takes two argument
 - `acc`: the accumulator
 - `x`: each element of the iterator

```
let a = [1, 2, 3];  
let sum = a.iter().fold(0, |acc, &x| acc + x); // sum = 6
```

Higher-order Function

- A function that either/or both:
 - Takes one or more functions as arguments
 - Returns a function as its result
- Normal functions are called “first-order” functions

```
fn twice<A>(function: impl Fn(A) -> A) -> impl Fn(A) -> A {  
    move |a| function(function(a))  
}  
  
fn plusthree(x: i32) -> i32 { x + 3 }  
  
fn main() {  
    let g = twice(plusthree);  
    println!("{}", g(7)); // sum = 13  
}
```

Fn trait is implemented
by all functions and
closures without
mutable references.