ECE326

PROGRAMMING LANGUAGES

Lecture 35 : Traits

Kuei (Jack) Sun

ECE

University of Toronto

Fall 2019

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> {
    // Explicitly requires `A` and `B`.
    fn contains(&self, _: &A, _: &B) -> bool;
}
```

- After

```
trait Contains {
    type A;
    type B;
    // Updated syntax to refer to these new types generically.
    fn contains(&self, &Self::A, &Self::B) -> bool;
}
```

Associated Type

- Using a trait with associated types

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

```
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()
}
```

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
      }
}
```