Rust Traits

CPSC 310 - Programming Languages

Overview

- Traits abstract behavior that types can have in common
  - Similar to Java interfaces
  - But, we can implement traits over any type, anywhere in the code (not just at the type definition)
- Trait bounds can be used to specify when a generic type must implement a trait
  - Similar to Java's bounded type parameters

Defining a Trait

- Example: trait with a single function

```
pub trait Summarizable {
    fn summary(&self) -> String;
}
```

- The `pub` keyword makes any module, function, or data structure accessible from inside external modules.

Implementing a Trait on a Type

- Example

```
impl Summarizable for (i32, i32) {
    fn summary(&self) -> String {
        let &(x, y) = self;
        format!("{}", x+y)
    }
}

fn foo() {
    let y = (1,2).summary();
    let z = (1,2,3).summary(); // fails
}
```

Default Implementations

- Example: trait with a default implementation

```
pub trait Summarizable {
    fn summary(&self) -> String {
        String::from("none") // default implementation
    }
}

impl Summarizable for (i32, i32, i32) {} // use default

fn foo() {
    let y = (1,2).summary(); // "3"
    let z = (1,2,3).summary(); // "none"
}
```

Trait Bounds

- With generics, you can specify that a type variable must implement a trait

```
pub fn notify<T: Summarizable>(item: T) {
    println!("Breaking news! {}", item.summary());
}
```

- This method works on any type `T` that implements the `Summarizable` trait

Generics, Multiple Bounds

- Trait implementations can be generic too

```
pub trait Queue<T> {
    fn enqueue(&mut self, elem: T) -> (); ...
}
impl <T> Queue<T> for Vec<T> {
    fn enqueue(&mut self, elem: T) -> () { ... } ...
}
```

- Generic method implementations of structs and enums can include trait bounds
- Can specify multiple Trait Bounds using +

```
fn foo<T: Clone + Summarizable>(...) -> i32 {...} or
fn foo<T>(...) -> i32 where T: Clone + Summarizable {..
```

(Non)Standard Traits

- Some standard traits
  - `Clone` holds if the object has a `clone()` method
  - `Copy` holds if assignment duplicates the object
  - `Move` holds if assignment moves ownership
  - `Deref` holds if you can dereference it
- There are other useful ones too
  - `Display` if it can be converted to a string
  - `PartialOrd` if it implements a comparison operator

Putting it all Together

- Example: find the largest element in an array slice

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

Putting it all Together (continued)

- Example: find the largest element in an array slice

```
fn main() {
    let int_list = vec![3, 1, 5, 4, 2];
    let result = largest(&int_list);
    println!("Largest int: {}", result);
    let char_list = vec!['p', 'z', 'q', 'y'];
    let result = largest(&char_list);
    println!("Largest char: {}", result);
}
```