CMSC 330: Organization of Programming Languages

Traits in Rust
Traits Overview

- **Traits** allow us to abstract behavior that types can have in common
  - In situations where we use *generic type parameters*, we can use *trait bounds* to specify that the *generic type* must implement a trait

- Traits are a bit **like Java interfaces**
  - But we can implement traits over any type, not necessarily at the point we define the type
Defining a Trait

• Here is a trait with a single function

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

– Specify &self for “instance” methods
– Can also specify “associated” methods
  • Like static methods in Java
– Equivalent in Java:

```java
public interface Summarizable {
    String summary();
}
```
Implementing a Trait on a Type

- **Name of Trait**: `Summarizable`
- **Type on which we are implementing it**: `(i32, i32)`

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

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

• Here is a trait with a default implementation

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

```
impl Summarizable for (i32,i32,i32) {
    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

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

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

• Can specify multiple Trait Bounds using +

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

• We have seen several standard traits already
  – **Clone** holds if the object has a clone() method
  – **Copy** holds if you can copy it
    • I.e., it’s a primitive
  – **Deref** holds if you can dereference it
    • I.e., it’s a reference

• There are other useful ones too
  – **Display** if it can be converted to a string
  – **PartialOrd** if it implements a comparison operator
Putting all Together

• Finds the largest element in an array slice
  – Generic in the type T of the contents of the array

```rust
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 all Together

• Finds the largest element in an array slice
  – Generic in the type \( T \) of the contents of the array

```rust
fn largest<T: PartialOrd + Copy>(list: &[T]) -> T {
    ...
}
fn main() {
    let number_list = vec![34, 50, 25, 100, 65];
    let result = largest(&number_list);
    println!("The largest number is {}", result);
    let char_list = vec![\'y\', \'m\', \'a\', \'q\'];
    let result = largest(&char_list);
    println!("The largest char is {}", result);
}
```

prints

The largest number is 100
The largest char is \( y \)
Notes

• Trait implementations can be generic too

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

• Generic method implementations of structs and enums can include trait bounds