

# Traits

**Rust, in Practice and in Theory**  
**Lecture 7**

# Generic Types

```
use std::collections::VecDeque;

fn reverse<T>(v: Vec<T>) -> VecDeque<T> {
    let mut out = VecDeque::new();
    for item in v {
        out.push_front(item);
    }
    out
}
```

Generic types allow us to write parametrically polymorphic functions

# Generic Structs and Enums

```
struct Point<T> {  
    x: T,  
    y: T,  
}
```

```
enum Result<T, E> {  
    Ok(T),  
    Err(E),  
}
```

We can also define generic structures and enumerations (just like parametric types in OCaml)

Note the syntax for multiple type parameters

# Generic Methods

```
impl<T> Point<T> {  
    fn x(&self) -> &T {  
        &self.x  
    }  
}
```

```
impl Point<f32> {  
    fn norm(&self) -> f32 {  
        (self.x.powi(2)  
         + self.y.powi(2))  
        .sqrt()  
    }  
}
```

We can define generic methods, we can give type parameters to implementations

We can also specify concrete types for generic structures and enumerations

# Monomorphization

```
enum Option<T> {  
    Some(T),  
    None,  
}
```



```
enum Option_i32 {  
    Some(i32),  
    None,  
}  
  
enum Option_f64 {  
    Some(f64),  
    None,  
}
```

Rust's compiler performs *monomorphization* on generic structures and functions

This means fast code, but (potentially) slow compile times and (potentially) large binaries

# Traits

# High Level

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

Traits allow us to define shared behavior of types

On the surface they are very simple, but Rust provides quite a bit of functionality with Traits

# Implementing Traits

```
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, self.location)  
    }  
}
```

We can implement traits for any type using

```
impl <Trait_id> for <TypeId> <Block>
```



# Useful Traits

- » Copy: copying instead of moving on assignment
- » Clone: cloning
- » Display: user-end printing
- » Debug: programmer-end printing
- » Deref: dereferencing operator (a bit tricky)
- » PartialEq: (==)
- » PartialOrd: (<), (<=), (>), (>=)...

# Copying and Cloning

```
struct MyStruct;  
  
impl Copy for MyStruct { }  
  
impl Clone for MyStruct {  
    fn clone(&self) -> MyStruct {  
        *self  
    }  
}
```

Copy is not overloadable (it's bit-wise)

Cloning is explicit (but can be derived)

# Derived Traits

```
[derive(Copy, Clone)]  
struct MyStruct;
```

Many basic traits can be derived (only traits with *derive pragmas*)

Example: A structure is copyable/clonable if all of its fields are

# Existential Types

```
pub fn notify(item: &impl Summary) {  
    println!("Breaking news! {}", item.summarize());  
}  
  
fn returns_summarizable() -> impl Summary {  
    Tweet {  
        username: String::from("horse_ebooks"),  
        content: String::from(  
            "of course, as you probably already know, people",  
        ),  
        reply: false,  
        retweet: false,  
    }  
}
```

Rust supports a kind of existential type by allowing us to specify a trait as a type

# Existential Types

```
pub fn notify(item: &impl Summary) {  
    ...  
}  
  
fn returns_summarizable() -> impl Summary {  
    ...  
}
```

We should think of **impl Summary** as " $\exists T . T$  is summarizable"

As noted in the text, this does *not* allow for dynamic dispatch (why?)

# Using Traits

# Trait Bounds

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


Trait bounds allow us to *restrict* type parameters

We should read "**<T: Trait>**" as "for any T which implements **Trait**"

# Where can we put Trait Bounds?

```
struct Foo<T> {  
    value: T  
}
```

```
impl<T: Clone> Foo<T> { }
```



okay

not okay

Seemingly anywhere

We can have a trait bound wherever we've *introduced* a type parameter



# "where" Syntax

```
fn some_function<T: Display + Clone, U: Clone + Debug>(t: &T, u: &U) -> i32 {
```

```
fn some_function<T, U>(t: &T, u: &U) -> i32
where
    T: Display + Clone,
    U: Clone + Debug,
{
```

When in doubt, we can write all trait bounds in where clauses (including trait bounds on **Self**)

# Advanced: Blanket Implementations

```
impl<T: Display> ToString for T {  
    // --snip--  
}
```

Blanket Implementations allow us to implement a trait for apply types satisfying another trait

# Advanced: Supertraits

```
pub trait Ord: Eq + PartialOrd {  
    // Required method  
    fn cmp(&self, other: &Self) -> Ordering;  
    ...  
}
```

We can also put trait bounds *on traits*, giving us a notion of supertraits

This allows us to build trait hierarchies