INTRO TO RUST LANG

BOX AND TRAIT OBJECTS

Benjamin Owad, David Rudo, and Connor Tsui

Today: Box and Trait Objects

- `Box<T>`
- The `Deref` and `Drop` trait
- Trait Objects
- Object Safety

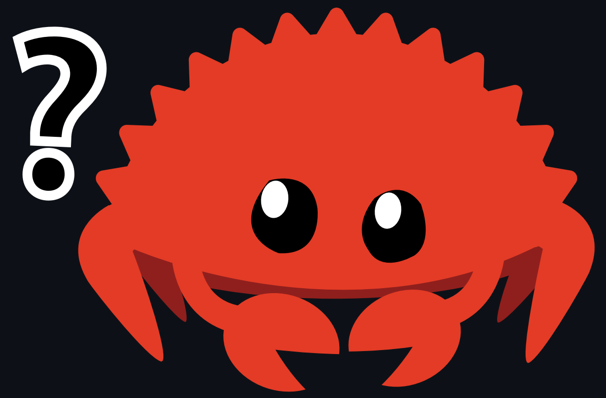
Motivation for `Box<T>`

Let's Make a List

Let's say we wanted to make recursive-style list

```rust
enum List {
  Cons(i32, List),
  Nil,
}

fn main() {
  // List of 1,2,3
  let list = Cons(1, Cons(2, Cons(3, Nil)));
}
```



Cargo's Suggestion

```
error[E0072]: recursive type `List` has infinite size
 --> src/main.rs:1:1
  |
1 | enum List {
  | ^^^^^^^^^
2 |     Cons(i32, List),
  |               ---- recursive without indirection
  |
help: insert some indirection (e.g., a `Box`, `Rc`, or `&`) to break the cycle
  |
2 |     Cons(i32, Box<List>),
  |               ++++    +
```

- Rust is upset because we've defined a type with infinite size
- The suggestion provided is to use a `Box<List>`

Indirection with `Box<T>`

```
let singleton = Cons(1, Box::new(Nil));
let list = Cons(1, Box::new(Cons(2, Box::new(Cons(3, Box::new(Nil))))));
```

- In the suggestion "indirection" means we store a *pointer* to a list rather than a list directly
    - Because a pointer has a fixed size our enum is no longer infinite!
- We create a `Box` with the `new` function

Cost of `Box<T>`

- `Box<T>` is a simple smart pointer, it just allocates on the heap**
- Boxes don't have performance overhead
  - Except for the overhead of allocation and pointer indirection
- `Box<T>` is a pointer type that fully owns the data, treated the same as any other owned value.
- They provide no other "special" capabilities

When to use `Box<T>`

- When you have a type of unknown size **at compile time** and you need its exact size
  - `List` from before
- When you have a large amount of data and want to transfer ownership and ensure no data is copied
  - Copying a pointer is faster than copying a large chunk of data
- When you want to own a value and you care only that it’s a type that implements a particular trait rather than being of a specific type
  - We'll get to this *soon*

Using Values in the `Box`

```
fn main() {
    let x = 5;
    let y = Box::new(x);

    assert_eq!(5, x);
    assert_eq!(5, *y);
}
```

- Just like a reference we can dereference a `Box<T>` to get `T`
- `Box<T>` implements the `Deref` trait which customizes the behavior of `*`

`Deref` Trait

The deref trait is defined as follows:

```
pub trait Deref {
    type Target: ?Sized;

    // Required method
    fn deref(&self) -> &Self::Target;
}
```

- Behind the scenes `*y` is actually `*(y.deref())`
- Note this does not recurse infinitely
- We are now able to treat smart pointers just like regular pointers!

Deref Coercion

```rust
fn hello(name: &str) {
    println!("Hello, {name}!");
}

fn main() {
    let m = Box::new(String::from("Rust"));
    hello(&m);
}
```

- Converts a reference to a type that implements the Deref trait into a reference to another type
  - Example: deref coercion can convert `&String` to `&str` because `String` implements the Deref trait such that it returns `&str`
- Here we see `Box<String>` deref coerces to `&str`

Deref Coercion Rules

Note Rust will coerce mutable to immutable but not the reverse

- From `&T` to `&U` when `T: Deref<Target=U>`
- From `&mut T` to `&mut U` when `T: DerefMut<Target=U>`
- From `&mut T` to `&U` when `T: Deref<Target=U>`

`&mut T` to `&mut U` Example

```rust
fn foo(s: &mut [i32]) {
    // Borrow a slice for a second.
}

// Vec<T> implements Deref<Target=[T]>.
let mut owned = vec![1, 2, 3];

foo(&mut owned);
```

&mut T to &U Example

```rust
fn foo(s: &[i32]) {
    // Borrow a slice for a second.
}

// Vec<T> implements Deref<Target=[T]>.
let mut owned = vec![1, 2, 3];

foo(&mut owned);
```

The Drop Trait

```rust
pub trait Drop {
    // Required method
    fn drop(&mut self);
}
```

- Determines what happens when value goes out of scope (dropped)
- You can provide an implementation of `Drop` on any type
- This is how Rust doesn't need you to carefully clean up memory

Drop Trait Example

```rust
struct CustomSmartPointer {
    data: String,
}

impl Drop for CustomSmartPointer {
    fn drop(&mut self) {
        println!("Dropping CustomSmartPointer with data `{}`!", self.data);
    }
}
```

Drop Trait In

```
fn main() {
    let c = CustomSmartPointer {
        data: String::from("my stuff"),
    };
    let d = CustomSmartPointer {
        data: String::from("other stuff"),
    };
    println!("CustomSmartPointers created.");
}
```

```
CustomSmartPointers created.
Dropping CustomSmartPointer with data `other stuff`!
Dropping CustomSmartPointer with data `my stuff`!
```

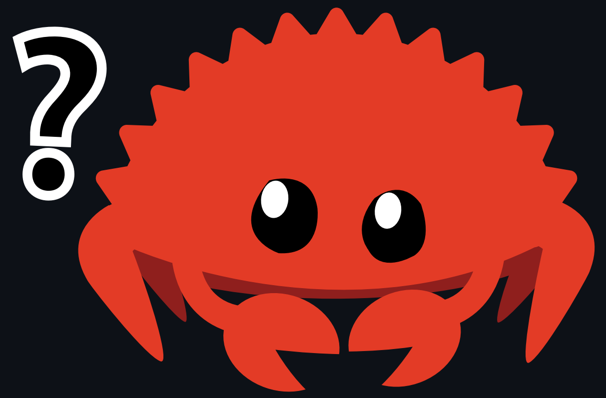
- Items are dropped in reverse order of creation

Dropping Manually

```rust
let c = CustomSmartPointer {
    data: String::from("some data"),
};
println!("CSM created.");
c.drop();
println!("CSM dropped before the end of main.");
```

```
error[E0040]: explicit use of destructor method
  --> src/main.rs:16:7
   |
16 |     c.drop();
   |     --^^^^--
   |     | |
   |     | explicit destructor calls not allowed
   |     help: consider using `drop` function: `drop(c)`
```

- Rust won't let you explicitly call the drop function to avoid double drops



Dropping Manually

```
fn main() {
  let c = CustomSmartPointer {
      data: String::from("some data"),
  };
  println!("CustomSmartPointer created.");
  drop(c);
  println!("CustomSmartPointer dropped before the end of main.");
}
```

- This code works since we use `std::mem::drop` instead
  - This is different that calling `c.drop()`
- You can think of this as `drop` taking ownership of `c` and dropping it
  - Actual source code: `pub fn drop<T>(_x: T) {}`

Object-Oriented Features of Rust

What we know

```rust
pub struct AveragedCollection {
    list: Vec<i32>,
    average: f64,
}

impl AveragedCollection {
  pub fn add(&mut self, value: i32) {
      self.list.push(value);
      self.update_average();
  }
}
```

- Encapsulation with `impl` blocks
- Public and private methods with crates and `pub`

Inheritence?

- Rust structs cannot inherit methods or data from another struct
- If we want code re-use:
    - We have traits (and even super traits)
- If we want polymorphism:
    - Rust has something called "trait objects"

Polymorphism

- Polymorphism != Inheritence
- Polymorphism = "Code that can work with multiple data types"
  - For inheritence this is usually subclases
- Rust polymorphism:
  - Generics - Abstract over different possible types
  - Trait bounds - Impose constraints on what types must provide

Trait Objects

```
pub trait Draw {
  fn draw(&self);
}

pub struct Screen {
  pub components: Vec<Box<dyn Draw>>,
}
```

- We want to implement a struct `Screen`
    - It holds a Vector of Drawable items
    - We use the `dyn` keyword to describe any type that implements Draw
        - We need to use a box since Rust doesn't know the size of `dyn Draw`

Trait Objects and Closures

```rust
fn returns_closure() -> Box<dyn Fn(i32) -> i32> {
  Box::new(|x| x + 1)
}

fn main() {
  let closure = returns_closure();
  print!("{}", closure(5)); // prints 6
}
```

- We can use trait objects to return dynamic types
- A Box is needed since `dyn Fn` has no known size
- Now with dereferencing coercion this isn't an awkward type to use!

Working With Trait Objects

```
impl Screen {
  pub fn run(&self) {
    for component in self.components.iter() {
      component.draw();
    }
  }
}
```

- Note this is different than a struct that uses trait bounds
- A generic parameter can only be substituted with one concrete type at a time
- Trait objects allow for multiple concrete types to fill in for the trait object **at runtime**

Generic Version

```rust
pub struct Screen<T: Draw> {
    pub components: Vec<T>,
}

impl<T> Screen<T>
where
    T: Draw,
{
    pub fn run(&self) {
        for component in self.components.iter() {
            component.draw();
        }
    }
}
```

- What's wrong with this version?
  - Well if we wanted a screen with multiple different types in it, it'd be much harder

Dynamically Sized Types

- Recall that we needed a `Box<dyn Draw>` before.
- `dyn Draw` is an example of a dynamically sized type (DST)
- Pointers to DSTs are double the size
    - Stores the a vtable pointer with it

DST Rules

- Traits may be implemented for DSTs
  - Unlike with generic type parameters, `Self: ?Sized` is the default in trait definitions
- They can be type arguments to generic type parameters having the special `?Sized` bound
- Ex: `struct Bar<T: ?Sized>(T);`
  - `?` marks an anti-trait (specifies a type **doesn't** implement a trait)

Next Lecture: ISD

Instructors still debating

- Thanks for coming!