Smart Pointers

Performant Software Systems with Rust — Lecture 11

Baochun Li, Professor

Department of Electrical and Computer Engineering

University of Toronto

What is a Pointer?

- A **pointer** is a variable that contains an address in memory — it **points to** some other data
- In Rust, a pointer is a **reference**, indicated by &, which borrows the value it points to

What is a Smart Pointer?

- A **smart pointer** is a data structure that acts like a pointer, but also has additional metadata and capabilities
  - Implemented using `struct`, and implements `Deref` and `Drop` traits
  - In many cases, a smart pointer **owns** the data it points to
- Examples of smart pointers we used
  - `String`
  - `Vec<T>`

Box<T>

- Stores data on the heap, and the pointer on the stack
- **Zero** performance overhead, but no additional capabilities either

- Use `Box<T>` in three situations:
  - Transfering ownership with a large amount of data — avoids copying
  - **Trait objects** — will get to this later
  - Having a type without a known size at compile time, but want to use a value of this type in a context that requires an exact size

Using `Box<T>` to Store Data on the Heap

```rust
fn main() {
    // stores an i32 value on the heap using Box<T>
    let b = Box::new(5);
    println!("b = {b}");
}
```

```
b = 5
```

When a Box Goes Out Of Scope

- It will be deallocated
- The deallocation happens both for the **box** (stored on the stack) and the data it points to (stored on the **heap**)

Trait Objects

Recall that we can use an `enum` to store different types of data in each cell, while still having a vector of these cells

```rust
enum SpreadsheetCell {
    Int(i32),
    Float(f64),
    Text(String),
}

let row = vec![
    SpreadsheetCell::Int(3),
    SpreadsheetCell::Text(String::from("blue")),
    SpreadsheetCell::Float(10.12),
];
```

There Is a Problem with Our Approach

- The types of cells in the vector are **fixed**
- But sometimes we are writing a library (crate) to be used by others, and want our library user to be able to extend the set of types
- For example, in a graphical UI, we wish the vector to store all the objects that can draw themselves
  - sounds like **abstract classes** in object-oriented programming!

Towards Using Trait Objects

- Let's say we wish to implement a **vector** that contains animals that can make a sound in our library
- But the types of these animals are to be defined by the users of our library
- We first define a **trait**

```
trait Animal {
    fn make_sound(&self);
}
```

Towards Using Trait Objects

- A user of our library defines two concrete types: `Dog` and `Cat`
- These types are of different sizes

```rust
struct Dog {
    name: String
}

struct Cat {
    lives: u8
}

impl Animal for Dog {
    fn make_sound(&self) {
        println!("Woof!");
    }
}

impl Animal for Cat {
    fn make_sound(&self) {
        println!("Meow!");
    }
```

Placing Trait Objects in Vectors

```rust
fn main() {
    // trait objects!
    let animals: Vec<dyn Animal> = vec![
        Dog { name: String::from("Rover") },
        Cat { lives: 9 }
    ];

    for animal in animals {
        animal.make_sound();
    }
}
```

Will it compile successfully?

Live Demo

Correct Solution

```rust
fn main() {
    let animals: Vec<Box<dyn Animal>> = vec![
        Box::new(Dog { name: String::from("Rover") }),
        Box::new(Cat { lives: 9 })
    ];

    // Use iter() to borrow each element
    for animal in animals.iter() {
        animal.make_sound();
    }
}
```

Why Boxing Trait Objects?

- Trait objects are **dynamically sized types** (DSTs) — we don't know their sizes at compile-time
- But the `T` in `Vec<T>` must implement the `Sized` trait
  - all types with known sizes automatically implement the `Sized` trait
- Boxed trait objects have **known sizes** — we know the size of a pointer!

Generic Types Implement `Sized`

```rust
fn generic<T>(t: T) {}
```

is the same as

```rust
fn generic<T: Sized>(t: T) {}
```

unless we explicitly opt out:

```rust
fn generic<T: ?Sized>(t: &T) {}
```

Dynamically Sized Types and Wide Pointers

- `str` is a **dynamically sized type**
  - since we don't know the size of a string slice
- `&str`, however, has a **known size**
  - it is a **wide pointer** (also called a **fat pointer**)
  - contains the address of `str` and its **length**

Wide Pointers for Slices

- Slices, such as `str` or `[T]`, is simply a view into some continuous data, such as a vector
- A **wide pointer** to a slice contains the address and the number of elements

Wide Pointers for Trait Objects

- A **wide pointer** to a trait object, such as `dyn Animal`, consists of a data pointer and a `vtable` pointer
  - the data pointer addresses the data (of some unknown type `T`) that the trait object is storing
  - the `vtable` is a `struct` of function pointers, pointing to the concrete piece of machine code for each method

Back to `Box<T>`

- Implements the `Deref` trait, allowing its values to be treated like **references**
- Implements the `Drop` trait, allowing the memory it boxes to be deallocated

Box<T> Implements the Deref Trait

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

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

Defining Our Own Simple Box

```rust
// MyBox is simply a tuple struct with one element of type T
#[derive(Debug)]
struct MyBox<T>(T);

impl<T> MyBox<T> {
    fn new(x: T) -> MyBox<T> {
        MyBox(x)
    }
}
```

Can it be dereferenced?

```rust
fn main() {
    let y = MyBox::new(5);
    assert_eq!(5, *y);
}
```

Implementing the `Deref` Trait

```rust
use std::ops::Deref;

impl<T> Deref for MyBox<T> {
    type Target = T; // uses an associated type

    fn deref(&self) -> &Self::Target {
        &self.0
    } // *y -> *(y.deref())
}
```

Deref Coercion

```rust
fn main() {
    let m = MyBox::new(String::from("Rust"));
    let hello = |name: &str| -> println!("Hello, {name}!");
    hello(&m); // equivalent to hello(&(*m)[..])
}
```

```
Hello, Rust!
```

Deref Coercion and the `DerefMut` Trait

- 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>`

The `Drop` Trait

By implementing the `drop` method (that takes `&mut self`) in the `Drop` trait, you specify the code to run when a value goes out of scope

```rust
impl<T> Drop for MyBox<T> {
    fn drop(&mut self) {
        println!("Dropping MyBox<T>!");
    }
}
```

Dropping a Value Early by with `std::mem::drop`

```rust
fn main() {
    let y = MyBox::new(5);
    assert_eq!(5, *y);

    let m = MyBox::new(String::from("Rust"));
    let hello = |name: &str| println!("Hello, {name}!");
    drop(y);
    hello(&m);
}
```

Live Demo

`Rc<T>` — The Reference Counted Smart Pointer

- Allowing multiple ownership of the same data
- Read-only, no mutability
- Uses **reference counting** to ensure memory safety
- Single-threaded, otherwise use `Arc<T>`

```rust
use std::rc::Rc; // remember to import Rc

fn main() {
    // Create a new reference-counted string
    let first = Rc::new(String::from("Hello"));
    println!("Rc after creation: {}", Rc::strong_count(&first));

    {
        // Create a second reference to the same data
        let second = Rc::clone(&first);
        println!("Rc after clone: {}", Rc::strong_count(&first));

        // Both references can read the data
        println!("First: {}", *first);
        println!("Second: {}", *second);
    } // second is dropped here

    // Reference count decreases when second goes out of scope
```

`RefCell<T>` and the Interior Mutability Pattern

- **Interior mutability** is a design pattern in Rust that allows you to mutate data even when there are immutable references to that data
- We do this with `RefCell<T>`

`RefCell<T>`

- Single ownership, unlike `Rc<T>`
- Moves borrowing checks from compile-time to runtime
- Only use when compile-time borrowing rules are too restrictive
- **Panics** if borrowing rules are violated at runtime
- Often used with `Rc` for shared mutable state in single-threaded contexts
- Use `Arc<Mutex<T>>` instead of `Rc<RefCell<T>>` in multi-threaded contexts

```rust
use std::cell::RefCell;
use std::rc::Rc;

fn main() {
    // Create a RefCell wrapped in an Rc for shared mutable access
    let counter = Rc::new(RefCell::new(0));

    // Create multiple references to the same data
    let counter_ref1 = Rc::clone(&counter);
    let counter_ref2 = Rc::clone(&counter);

    // Modify the value through the first reference
    *counter_ref1.borrow_mut() += 1;
    // `borrow()` returns a Ref<T> which derefs to &T
    println!("After first modification: {}", counter_ref1.borrow());

    // Modify the value through the second reference
    *counter_ref2.borrow_mut() += 2;
```

Live Demo

Required Additional Reading

The Rust Programming Language, Chapter 15.1 – 15.5, 18.2, 20.3

Rust for Rustaceans, Jon Gjenset, **Chapter 2 (Types)**