APPS@UCU

Rust #4: Smart pointers, Concurrency and Async

Sultanov Andriy

Contents

Smart pointers

Smart pointers
General

Rust standard library has a few useful smart pointers:

- Box<T>
- Rc<T>
- Cell<T>
- RefCell<T>

Box<T>

Boxes are Rust’s way to store data on the heap.

They are useful when you want to handle values whose sizes can not be known at compile time as if their size was known. This works since the size of the pointer to the heap that the Box keeps on the stack is known.

Box<T>

An example of moving a type that is typically stored on the stack onto the heap:

```
let val: u8 = 42;
let boxed: Box<u8> = Box::new(val);

// We can easily dereference the box
println!("{}", boxed); // prints 42
```

The value is going to be deallocated once it goes out scope, calling Box's drop implementation.

Box<T>

We could have just as well explicitly dereferenced the Box:

```
let val: u8 = 42;
let boxed: Box<u8> = Box::new(val);

// We can easily dereference the box
println!("{}", boxed); // prints 42

// And explicitly like this:
println!("{}", *boxed);
```

Smart pointers
Deref trait

Implementing Deref allows Rust to treat smart pointers as references to the values they hold.

You can dereference types impementing Deref explicitly using the * operator, or you can omit the dereferencing and let Rust implicitly coerce your type to the type you need.

Thus, when calling a method on a type that implements Deref and does not implement the method, Rust will recursively dereference this type until it finds the appropriate method.

Smart pointers
Deref trait

Rust will coerce types according to these rules:

- 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>

Box<T>

If you want to create an unsized data structure, for example a recursive cons list, you can just use Box:

```
enum List<T> {
    Cons(T, Box<List<T>>),
    Nil,
}

// Creating a new list
let list: List<i32> = List::Cons(1, Box::new(List::Cons(2, Box::new(List::Nil))));
println!("{:?}", list); // prints Cons(1, Cons(2, Nil))
```

What will end up being stored is an i32 and a pointer usize to another List on the heap.

Sized

Boxes can also be useful if your function takes a trait object as a parameter (which can't possibly have its size known at compile time).
TODO

Rc<T>

TODO

Cell<T>

TODO

RefCell<T>

TODO

Concurrency

Concurrency

TODO: Talk about Arc, Mutex, mspc, and the way Rust's system prevents data races at compile time.

Async

Async

Rust has, relatively recently, finally settled with its asynchronous implementation. We do not have the time to cover all the details of its implementation, and therefore its history and efficiency, however, we will see the ease with which your Rust programs can become asynchronous.
TODO: talk about runtimes, executors, reactors, libraries, futures, async, await, closures etc.

Looking back

Looking back

TODO: Having learned these powerful abstractions and techniques blah blah blah

And ahead...

Looking back

TODO: We haven’t covered quite a lot of important stuff in Rust. Here are some pointers on what you could do and where you should probably look

Thank you!