`Rc<T>` and `RefCell<T>`

CAS CS 392: Rust, in Theory and in Practice

February 27, 2025 (Lecture 11)

Outline

High Level

Ownership is nice, borrowing can make it ownership nice to work with, but we *still* may want data to have multiple owners:

```rust
// THIS DOES NOT COMPILE
use crate::List::{Cons, Nil};
enum List {Cons(i32, Box<List>), Nil}
fn main() {
    let a = Cons(5, Box::new(Cons(10, Box::new(Nil))));
    let b = Cons(3, Box::new(a));
    let c = Cons(4, Box::new(a));
}
```

Recall: Reference Counting

Languages like Swift allow for this using *reference counting*:

```
class Stuff {
    init() {print("allocating")}
    deinit {print("deallocating")}
}

var r1 : Stuff? = Stuff()
var r2 : Stuff? = r1
var r3 : Stuff? = r2

r1 = nil                                        // prints:
r2 = nil                                        // allocating
r3 = nil                                        // deallocating
```

- The number of references is maintained
- Once the count goes to zero, the associated memory can be freed

Recall: Lists

```rust
// THIS DOES NOT COMPILE
use crate::List::{Cons, Nil};
enum List {Cons(i32, Box<List>), Nil}
fn main() {
    let a = Cons(5, Box::new(Cons(10, Box::new(Nil))));
    let b = Cons(3, Box::new(a));
    let c = Cons(4, Box::new(a));
}
```

(Last time) We can build recursive list data types using boxes

Boxes still don't allow for complex reference structures, like two lists with the same tail because **a box must still have a single owner**

Mini-Puzzle

```
// THIS DOES NOT COMPILE
use crate::List::{Cons, Nil};
enum List {Cons(i32, Box<List>), Nil}
fn main() {
    let mut a = Cons(5,
                     Box::new(Cons(10,
                                   Box::new(Nil))));
    a = Cons(3, Box::new(a));
    let x = if let Cons(n, _) = a { n } else { 0 }
    println!("{}", x)
}
```

Is this okay?

The Reference Counting Type

```rust
use crate::List::{Cons, Nil};
use std::rc::Rc;
enum List {Cons(i32, Rc<List>), Nil}
fn main() {
    let a = Rc::new(Cons(5,
                         Rc::new(Cons(10,
                                      Rc::new(Nil)))));
    let b = Cons(3, Rc::clone(&a));
    let c = Cons(4, Rc::clone(&a));
}
```

We can get these kinds of structures with the *reference counted smart pointers*

`Rc<T>` is like a box, except that it's possible to clone the reference *without cloning the data itself*

`Rc::clone(&a)` just increments the reference count

Aside: Deref and Reference Counting

Rc<T> implements the Deref trait:

```
let x : Rc<i32> = Rc::new(5);
let x_ref : &Rc<i32> = &x;
let x_ref2 : &i32 = &x;
let _ : Rc<i32> = Rc::clone(x_ref);
// let _ : Rc<i32> = Rc::clone(x_ref2);
```

Rc::clone has the following signature:

```
fn clone(&self) -> Rc<T, A>
```

The point: How a reference is coerced can be determined by the type annotation (or by type inference)

Rc::clone vs. .clone()

We can also just use `.clone()`:

```
let x : Rc<i32> = Rc::new(5);
let _ = x.clone();
```

The Rust convention is to use `Rc::clone` to distinguish *shallow clones* from *deep clones*

(another verbose Rust convention)

References Counts

Clones increment the reference count:

```rust
impl<T: ?Sized, A: Allocator + Clone> Clone for Rc<T, A> {
    fn clone(&self) -> Self {
        unsafe {
            self.inner().inc_strong();
            Self::from_inner_in(self.ptr,
                                self.alloc.clone())
        }
    }
}
```

We can use `Rc::strong_count` to see what the current count is:

```rust
fn main() {
    let x = Rc::new(5); let y = x.clone();
    assert_eq!(2, Rc::strong_count(&x)); drop(y);
    assert_eq!(1, Rc::strong_count(&x));
}
```

Immutability

Reference counted smart pointers are immutable:

```rust
// THIS DOES NOT COMPILE
let x : Rc<i32> = Rc::new(5);
*x = 3;
```

If we have multiple references, we don't want to be able to mutate them!

Reference Counting and Concurrency

We also have `Arc<T>` for (multi-)thread safe ("atomic") reference counting:

```
let counter = Arc::clone(&counter);
let handle = thread::spawn(move || {
    let mut num = counter.lock().unwrap();
    *num += 1;
});
```

`Rc<T>` should only be used the in single-threaded settings (generally better performance)

(We may or may not talk about concurrency next week)

Outline

RefCell<T>

```
fn main() {
    let x = RefCell::new(5);
    let mut y : RefMut<'_, i32> = x.borrow_mut();
    *y += 1; drop(y);
    println!("{}", x.borrow())
}
```

`RefCell<T>` owns the data it holds like a `Box<T>`

We get a compile time error if we create multiple mutable references to a `Box<T>`

We get a **runtime error** if we create multiple mutable references to a `RefCell<T>`

The Takeaway: We can mutate a value in a `RefCell<T>` even when it is immutable

Trade-offs

Compile-time errors give us better assurance that our code is correct

There are memory-safe operations that are not allowed by the borrow checker

The Rust compiler is conservative. *Annoyance is better than catastrophe*

Comparisons

smart pointer	# owners	mut ref allowed?	mut ref checked?
`Box<T>`	one	yes	compile-time
`Rc<T>`	many	no	N/A
`RefCell<T>`	one	yes	run-time

Example: Mock Objects

There's a nice example in the text on this:

```
struct MockMessenger {sent_messages: RefCell<Vec<String>>,}
impl MockMessenger {
    fn new() -> MockMessenger {
        MockMessenger {sent_messages: RefCell::new(vec![]),}
    }
}
impl Messenger for MockMessenger {
    fn send(&self, message: &str) {
        self.sent_messages
            .borrow_mut().push(String::from(message));
    }
}
```

The rough idea: If we *have* to work with something by (immutable) reference (e.g., because of a trait), we can have mutable state in a `RefCell`

Reference Counting + Interior Mutability

We can combine reference counting and interior mutability to have mutable values in things like lists:

```
enum List {Cons(Rc<RefCell<i32>>, Rc<List>), Nil}
fn main() {
    let value = Rc::new(RefCell::new(5));
    let a = Rc::new(Cons(Rc::clone(&value), Rc::new(Nil)));
    let b = Cons(Rc::new(RefCell::new(3)), Rc::clone(&a));
    let c = Cons(Rc::new(RefCell::new(4)), Rc::clone(&a));
    *value.borrow_mut() += 10;
}
```

Outline

Reference Cyles

Combining `RefCell<T>` and `Rc<T>` can create *reference cycles*:

```
let a = Rc::new(Cons(5, RefCell::new(Rc::new(Nil))));
let b = Rc::new(Cons(10, RefCell::new(Rc::clone(&a))));
if let Some(link) = a.tail() {
    *link.borrow_mut() = Rc::clone(&b);
}
```

Reference cycles can lead to leaked memory (it is impossible to bring the reference count down to 0)

This means /memory leaks are "safe" in Rust

An Implementation of Rc<T>

```
pub struct Rc<
    T: ?Sized,
    A: Allocator = Global,
> {
    ptr: NonNull<RcInner<T>>,
    phantom: PhantomData<RcInner<T>>,
    alloc: A,
}
// ...
fn clone(&self) -> Self {
    unsafe {
        self.inner().inc_strong();
        Self::from_inner_in(self.ptr, self.alloc.clone())
    }
}
```

What the heck is the NonNull? And what is PhantomData? And is it okay to use this unsafe code?

Outline

Task

Implement `hd`, `tl` and `get` for the following representation of a linked-list:

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