CMSC 330: Organization of Programming Languages

Reference Counting

and Interior Mutability

Rust Ownership and Mutation

- Recall Rust ownership rules
  - Each value in Rust has a variable that's called its *owner*; there can be only one
  - When the owner goes out of scope, the value will be dropped
- Recall Rust mutability rules
  - Mutation can occur only through mutable variables (e.g., the owner) or references
  - Rust permits only one borrowed mutable reference (and no immutable ones at the same time)

Relaxing Rust's Restrictions

- Architecturally, designating one owner that all accesses must go through can be awkward
  - We might end up wanting shared mutable access to the owner!

- Rust provides APIs by which you can get around the compiler-enforced restrictions against multiple mutable references
  - Use reference counting to manage lifetimes safely
  - Track borrows at run-time to overcome limited compiler analysis
  - Discipline is called interior mutability
  - But: extra checks at space and time overhead; some previous compile-time failures now occur at run-time

Multiple Pointers to a Value

- What's wrong with this code?

```
enum List {
  Nil,
  Cons(i32,Box<List>)
}
```

```
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));//fails
}
```

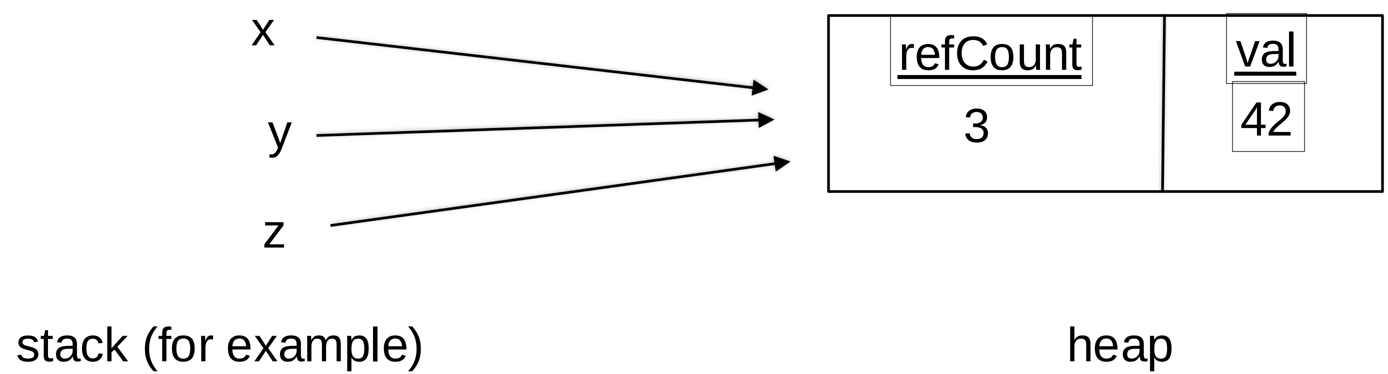
  - **`Box::new`** takes ownership of its argument, so the second **`Box::new(a)`** call fails since **`a`** is no longer the owner
- How to allow something like this code?
  - Problem: Managing lifetime

Rc<T>: Multiple Owners, Dynamically

- This is a *smart pointer* that associates a counter with the underlying reference
- Calling **clone** copies the pointer, not the pointed-to data, and bumps the counter by one
  - By convention, call **`Rc::clone(&a)`** rather than **`a.clone()`**, as a visual marker for future performance debugging
    - In general, calls to **`x.clone()`** are possible issues
- Calling **drop** reduces the counter by one
- When the counter hits zero, the data is **freed**

Rc::clone "Shares" Ownership

- **`Rc`** associates a refCount with the value



- **`let x = Rc::new(42);`**
- **`let y = Rc::clone(x);`**
- **`let z = Rc::clone(x);`**

does heap allocation, like Box::new, but uses reference counting

clone() increments reference count

clone() increments reference count

Lists with Sharing

```
enum List {
  Nil,
  Cons(i32,Rc<List>)
}

use List::{Cons, 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));//ok
}
```

Nb. `Rc::strong_count` returns the current ref count

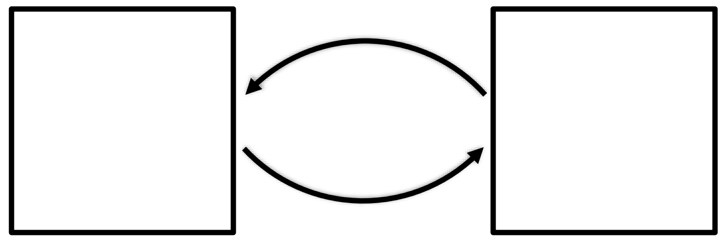
Reference Counting: Summary

- To *create*: `let r = Rc::new(...);`
- To *copy* a pointer: `let s = Rc::clone(&r);`
  - Increments the reference count
- To *move* a reference: `let t = s;`
  - Does *not* increment reference count; `s` no longer the owner
- To *free* is automatic: `drop` is called when variables go out of scope, reducing the count; freed when 0

- See docs:
  - https://doc.rust-lang.org/book/ch15-04-rc.html
  - https://doc.rust-lang.org/std/rc/index.html

Risks of Reference Counts

- Cyclic data is problematic
  - Suppose the arrows are **`Rc`** references



  - Reference counts are always positive; will never be deallocated!

- Can fix by using *weak references* (see docs)
  - App must be prepared for referent to be revoked

Rc References: Mutation?

- With **`Rc`** I can now make multiple references and safely manage lifetimes. Great! Let's see if I can mutate the reference's contents

```
let mut b = Rc::new(42);
*b = 43;
```

```
warning: variable does not need to be mutable
 --> src/main.rs:4:9
  |
4 |     let mut b = Rc::new(42);
  |         ----^
  |         |
  |         help: remove this `mut`
  |
  = note: `#[warn(unused_mut)]` on by default

error[E0594]: cannot assign to data in an `Rc`
 --> src/main.rs:5:5
  |
5 |     *b = 43;
  |     ^^^^^^^ cannot assign
  |
  = help: trait `DerefMut` is required to modify through a dereference,
but it is not implemented for `Rc<i32>`
```

Rc References: No Mutation!

```
error[E0594]: cannot assign to data in an `Rc`
 --> src/main.rs:5:5
  |
5 |     *b = 43;
  |     ^^^^^^^ cannot assign
  |
  = help: trait `DerefMut` is required to modify through a dereference, but it is not implemented for
`Rc<i32>`
```

`Rc` only allows *immutable* contents

```
let mut b = Rc::new(42);
b = Rc::new(43); // fresh heap alloc
```

`mut b` means that I can reassign **`b`**, but not the object it references!

Digression: Cells are Mutable

- **Cell<T>**: like **Box<T>** but with mutable contents

```
pub fn set(&self, val: T)
```

  - *moves* the data in

```
pub fn get(&self) -> T
```

  - *copies* the data out

```
pub fn take(&self) -> T
```

  - *moves* the data out, leaving **Default::default()**

```
pub fn get_mut(&mut self) -> &mut T
```

  - requires a `&mut self`

Cell Limitations

- **`Cell`** is great if
  - you can copy the contents in and out
  - and you have mutable references to the cell whenever you want to modify the cell's contents
  - and you can reason statically about lifetimes
- But what if you can't or don't?
  - e.g., you want to access contents of cell without copying it out (maybe it's a struct that's not **`Copy`**)
- Enter: **`RefCell`**

RefCell<T>

```
pub const fn new(value: T) -> RefCell<T>
```

- Looks similar…

```
pub fn borrow(&self) -> Ref<'_, T>
```

- This is a *dynamic* borrow
- *"The borrow lasts until the returned* ***`Ref`*** *exits scope. Multiple immutable borrows can be taken out at the same time…Panics if the value is currently mutably borrowed. "*

```
pub fn borrow_mut(&self) -> RefMut<'_, T>
```

- Note **`&self,`** not **`&mut self`**!
- *"The borrow lasts until the returned* ***`RefMut`*** *or all* ***`RefMuts`*** *derived from it exit scope. The value cannot be borrowed while this borrow is active."*

`Ref` and **`RefMut`** are *only* for use with **`RefCell`**

Static vs. Dynamic Borrow Tracking

- **`&T`** an d **`&mut T`**: *static* (compile-time) tracked of borrows

- **`RefCell<T>::borrow*`**: *dynamic* (run-time) tracked of borrows

```
pub fn borrow(&self) -> Ref<'_, T>
pub fn borrow_mut(&self) -> RefMut<'_, T>
```

  - **`Ref<'_, T>`**, **`RefMut<'_, T>`** implement dynamic tracking of outstanding, borrowed references
  - If **`borrow_mut()`** with an outstanding **`Ref`**, panic!

- Static tracking is better if you can make it work
  - no run time overhead; earlier bug detection

How Does Dynamic Borrowing Work?

- Each **`RefCell`** has a *borrow count* to track outstanding **`Ref`**s and **`RefMut`**s for that **`RefCell`**
  - **`RefCell`** **`borrow`** and **`borrow_mut`** increment the count
  - When a **`Ref`** (or **`RefMut`**) goes out of scope, Rust calls **`drop()`**, which decrements the borrow count

```
use std::cell::RefCell;
let c = RefCell::new(5); // imm_count=0
let m = c.borrow();      // imm_count=1
let b = c.borrow_mut();  // panic!
```

Shared Mutable Data

- Back to the beginning: We were looking for a way to have *shared, mutable* data. How do we do it? **Use `Rc<RefCell<T>>`**
  - The `RefCell` permits mutating `T` (at risk of run-time borrow errors)
  - `Rc` permits sharing, e.g., within a data structure
- Note: `Rc<RefCell<u32>>` has *two* counts:
  - Reference count for `Rc` (should this `RefCell` be deallocated?)
    - Incremented via `Rc::clone()`
    - Dynamic version of lifetime
  - Borrow count for `RefCell` (are `borrow(), borrow_mut()` safe?)
    - Incremented via `RefCell` `borrow` and `borrow_mut`
    - Dynamic version of borrow checking

Quiz 3

```
let r1 = Rc::new(RefCell::new(42));
let r2 = r1.clone();
let m = (*r1).borrow_mut();
*m = 43;
println!("{:?}", *r2.borrow());
```

A. "42"

B. "43"

C. panic

D. Compiler error

Quiz 3

```
let r1 = Rc::new(RefCell::new(42));
let r2 = r1.clone();
let m = (*r1).borrow_mut();
*m = 43;
println!("{:?}", *r2.borrow());
```

A. "42"

B. "43"

C. panic

D. Compiler error

```
error[E0596]: cannot borrow `m` as mutable, as it is not declared as mutable
  --> src/main.rs:10:10
   |
9  |         let m = (*r1).borrow_mut();
   |             - help: consider changing this to be mutable: `mut m`
10 |         *m = 43;
   |          ^ cannot borrow as mutable
```

Quiz 3

```
let r1 = Rc::new(RefCell::new(42));
let r2 = r1.clone();
let m = (*r1).borrow_mut();
*m = 43;
println!("{:?}", *r2.borrow());
```

`borrow_mut()` returns a `DerefMut`
`DerefMut:`
```
   pub fn deref_mut(&mut self) -> &mut Self::Target
```

To mutate the referenced value, we need a *mutable* `DerefMut`

Quiz 4

```
let r1 = Rc::new(RefCell::new(42));
let r2 = r1.clone();
let mut m = (*r1).borrow_mut();
*m = 43;
println!("{:?}", *r2.borrow());
```

A. "42"

B. "43"

C. panic

D. Compiler error

Quiz 4

```
let r1 = Rc::new(RefCell::new(42));
let r2 = r1.clone();
let mut m = (*r1).borrow_mut();
*m = 43;
println!("{:?}", *r2.borrow());
```

A. "42"

B. "43"

C. panic

D. Compiler error

`m`'s mutable borrow of the `RefCell` is still outstanding when `borrow()` is invoked.

Quiz 5

```
let r1 = Rc::new(RefCell::new(42));
let r2 = r1.clone();
{
    let mut m = (*r1).borrow_mut();
    *m = 43;
}
println!("{:?}", *r2.borrow());
```

A. "42"

B. "43"

C. panic

Quiz 5

```
let r1 = Rc::new(RefCell::new(42));
let r2 = r1.clone();
{
    let mut m = (*r1).borrow_mut();
    *m = 43;
}
println!("{:?}", *r2.borrow());
```

```
A. "42"
B. "43"
C. panic
```

Summary

- From the book [1]:
  - **`Rc<T>`** enables multiple owners of the same data; **`Box<T>`** and **`RefCell<T>`** have single owners.
  - **`Box<T>`** allows immutable or mutable borrows checked at compile time; **`Rc<T>`** allows only immutable borrows checked at compile time; **`RefCell<T>`** allows immutable or mutable borrows checked at runtime.
  - Because **`RefCell<T>`** allows mutable borrows checked at runtime, you can mutate the value inside the **`RefCell<T>`** even when the **`RefCell<T>`** is immutable.

[1] https://doc.rust-lang.org/book/ch15-05-interior-mutability.html
Additional examples: https://doc.rust-lang.org/rust-by-example/std/rc.html