

CSE 210: Principles of Software Engineering

Rust: 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)

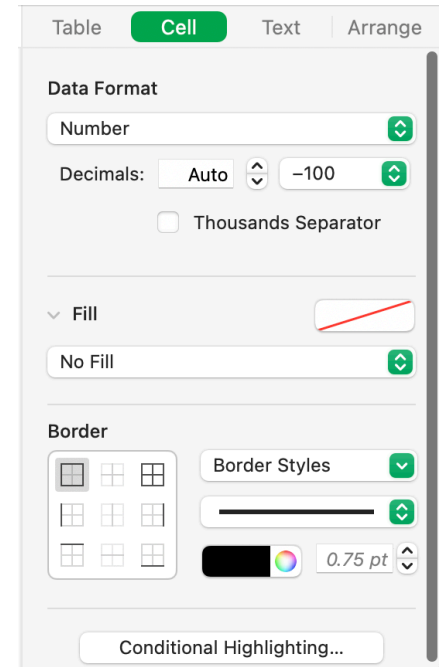
But: Mutation and Sharing is Useful

- Example: a simple spreadsheet

```
struct CellStyle { fontSize: f64 }  
struct Cell { style: CellStyle }  
struct Table { cells: [Cell; 128] }
```

– So: a **Table** owns its **Cells**

- But: a format inspector needs to read *and write* the cell data
 - Ensuring only one borrowed mutable reference would be awkward
 - Easier if the inspector has its own reference



Another Example

- Suppose you have a multiplayer chess game
 - Local data structures record the board state
 - Maybe the board is owned by the window that contains it
- What happens when a new move comes in from the network?
 - That's handled by a different software component, not the window
- Simplest design is to have multiple (mutable) references to the board
 - But Rust doesn't allow that

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
 - Also a pain to program: [Experimental GcRef](#) to ease this

Multiple Pointers to a Value

- What's wrong with this code?

```
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  
}
```

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

- **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

Managing Lifetimes Dynamically

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

- Benefit of ownership: compiler knows when to free memory

```
{  
  let nil_box = Box::new(List::Nil);  
  // free memory HERE (nil_box is going out of scope)  
}
```

- Suppose **Box** *didn't* own its data:

```
let nil_box = Box::new(List::Nil);  
let one_list = List::Cons(1, nil_box);  
{  
  let two_list = List::Cons(2, nil_box);  
  // two_list is going out of scope; free nil_box too?  
}
```

error[E0382]: use of
moved value: `nil_box`

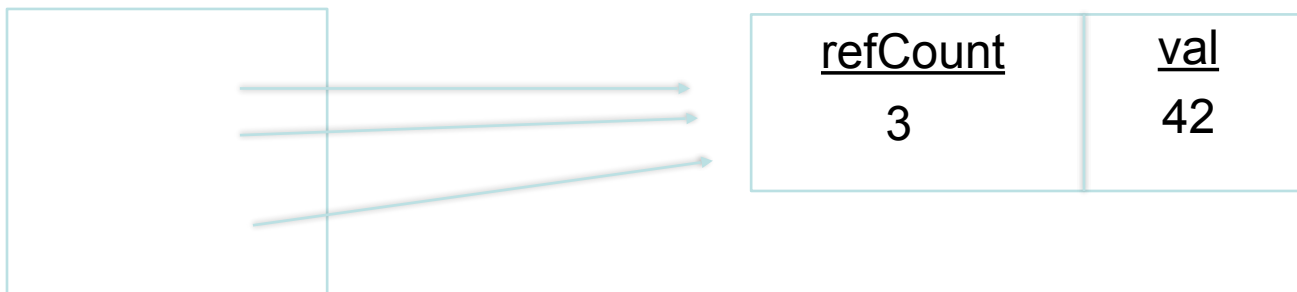
- (**Box** does own its data so the above pattern is not allowed.)

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



stack (for example)

heap

- **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>

Quiz 1

```
fn print_refcount(r: Rc<i32>) {  
    println!("{}", Rc::strong_count(&r));  
}  
  
fn main() {  
    let forty_two = Rc::new(42);  
    print_refcount(forty_two);  
    {  
        let v = Rc::clone(&forty_two);  
        print_refcount(v); // What does this print?  
    }  
}
```

- A. 0
- B. 1
- C. 2
- D. This code doesn't compile

Quiz 1

```
fn print_refcount(r: Rc<i32>) {
    println!("{}", Rc::strong_count(&r));
}

fn main() {
    let forty_two = Rc::new(42);
    print_refcount(forty_two);
    {
        let v = Rc::clone(&forty_two);
        print_refcount(v); // What does this print?
    }
}
```

- A. 0
- B. 1
- C. 2

D. This code doesn't compile

```
error[E0382]: borrow of moved value: `forty_two`
--> src/main.rs:46:27
```

```
43 |         let forty_two = Rc::new(42);
    |         ----- move occurs because
    |         `forty_two` has type `std::rc::Rc<i32>`, which
    |         does not implement the `Copy` trait
```

Quiz 2

```
fn print_refcount(r: &Rc<i32>) {  
    println!("{}", Rc::strong_count(r));  
}  
  
fn main() {  
    let forty_two = Rc::new(42);  
    {  
        let v = Rc::clone(&forty_two);  
    }  
    print_refcount(&forty_two); // What does this print?  
}
```

- A. 0
- B. 1
- C. 2
- D. This code doesn't compile

Quiz 2

```
fn print_refcount(r: &Rc<i32>) {  
    println!("{}", Rc::strong_count(r));  
}  
  
fn main() {  
    let forty_two = Rc::new(42);  
    {  
        let v = Rc::clone(&forty_two);  
    }  
    print_refcount(&forty_two); // What does this print?  
}
```

A. 0

B. 1 *v* went out of scope, so the reference count is 1 (once again).

C. 2

D. This code doesn't compile

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
 - These are not required for project 5

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 example (from Rust book)

```
use std::cell::Cell;

struct SomeStruct {
    regular_field: u8,
    special_field: Cell<u8>,
}

let my_struct = SomeStruct {
    regular_field: 0,
    special_field: Cell::new(1),
};

let new_value = 100;

// ERROR: `my_struct` is immutable
// my_struct.regular_field = new_value;

// WORKS: although `my_struct` is immutable, `special_field` is a `Cell`,
// which can always be mutated
my_struct.special_field.set(new_value);
assert_eq!(my_struct.special_field.get(), new_value);
```

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

Ref<T> vs. &T

- Both **Ref<T>**, returned by **borrow***, and **&T**, implement **Deref**
 - Code that uses them will be similar

&T

```
let x = 42;  
let r = &x;  
assert_eq!(*r, 42);
```

Ref<T>

```
let cell = RefCell::new(42);  
let cell_ref : Ref<i32> = cell.borrow();  
assert_eq!(*cell_ref, 42);
```

Static vs. Dynamic Borrow Tracking

- `&T` and `&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 **Refs** and **RefMuts** 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.

A Quick Summary

- **&mut**: use when you only need one mutable reference
- **Rc**: reference-counted, shared reference to the heap
- **RefCell/Cell**: mutable contents *even when immutable*
 - Borrowing via a special **Ref** value, which ensures that Rust's borrow checking rules are followed *dynamically*
 - Combine with **Rc** for shared mutability
- **Ref/RefMut**: only used for accessing **RefCell**.

Conclusions

- Ideally, design Rust programs so each value has one owner
 - But that's not always possible
 - Even when it is, those designs may have other costs
- When necessary, use `Rc` and `RefCell` to relax Rust's static constraints
 - Part of a programming discipline called [interior mutability](#).
 - With great power comes great responsibility!