Lecture 9

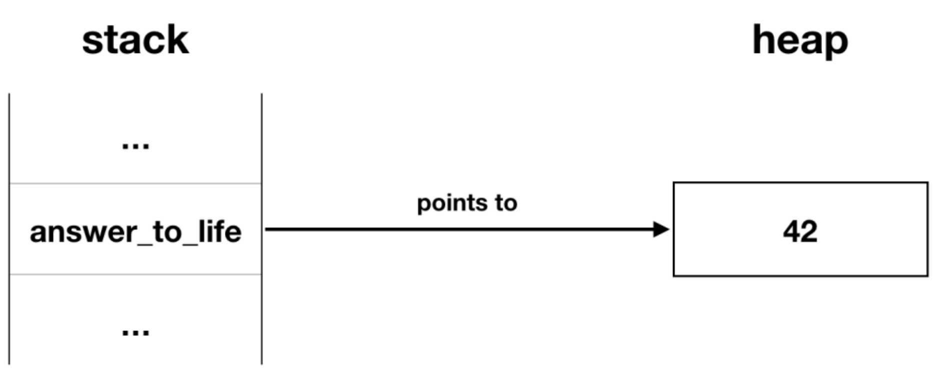
Interior Mutability

Rust Has Many Smart Pointers

- For example:

Box<T> **Rc<T>** **Cell<T>** **RefCell<T>** .....

- **Box<T>** is a container type designed to allocate and "hold" an object on the heap.
- It's the simplest form of allocation on the heap and the content is dropped when it goes out of scope.

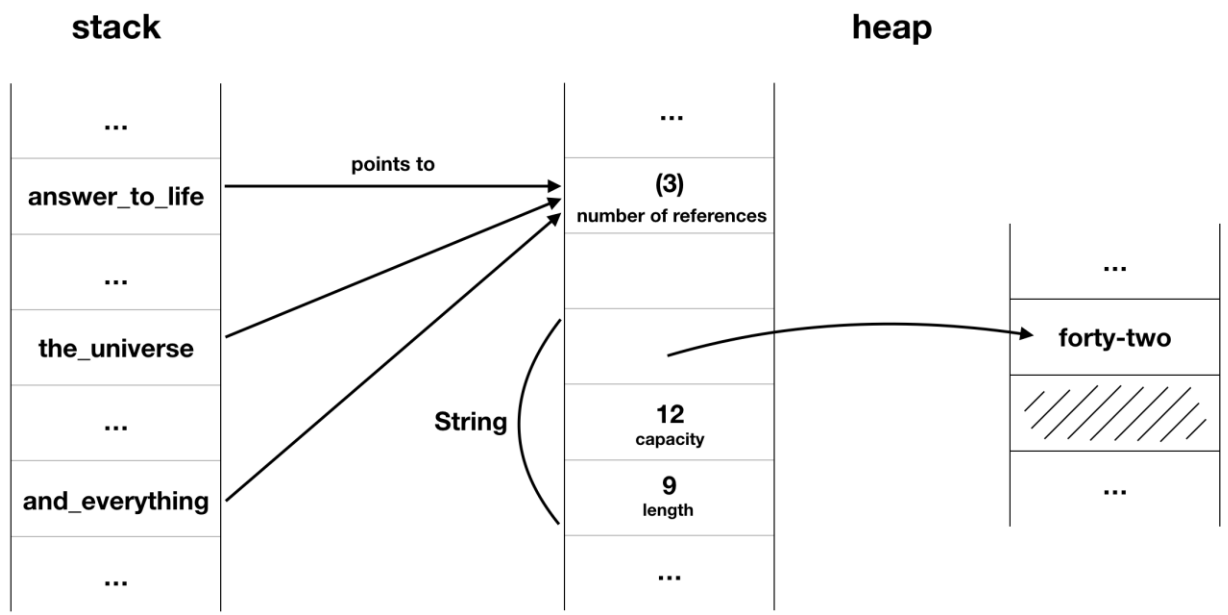


From Box to RC to .....

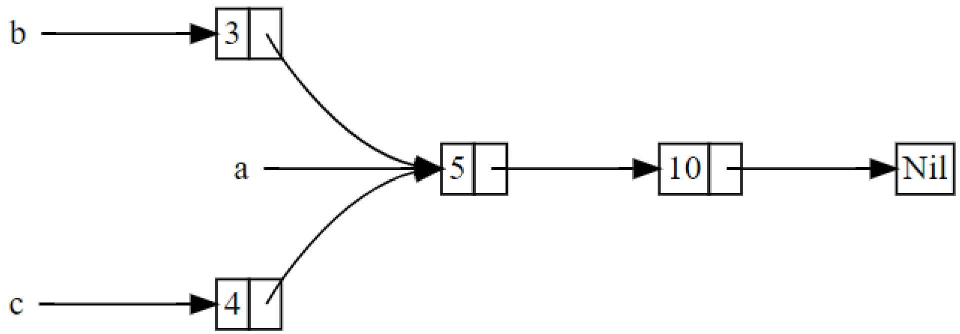
- Why do we need multiple pointers?
- Well still stop talking pointers and start talking environments or wraps
- Box wraps a T – but if we research Box it has very limited functionality
- RC has more functionality, more options
- However other wrappers have even more ....
- Rust defines a large number of wrapper types and asks the programmer to select the correct one for the circumstances.

Rc<T> (Reference Counting)

- **Rc<T>** provides shared ownership over some content .
- It counts the uses of the reference pointing to the same piece of data on the heap. **Read**
- when the last reference is dropped, the data itself will be dropped and the memory properly freed.



Several things want a .....



Who owns a?

Think of a as a package, I want to be able to use my package multiple times, correct?

Surely – this will work?

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

use crate::List::{Cons, 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));
}
```

Compiler says – no way

$ cargo run

```
let a = Cons(5, Box::new(Cons(10, Box::new(Nil))));
| - move occurs because `a` has type `List`,
which does not implement the `Copy` trait
let b = Cons(3, Box::new(a));
| - value moved here
| let c = Cons(4, Box::new(a));
| ^ value used here after move error:
```

No moving twice!!!

b and c play nicely -- sharing

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

use crate::List::{Cons, Nil};
use std::rc::Rc;
```

Reference counting – multiple sharing

```
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));
}
```

Ummmmm … sharing; lets rewrite main

```
fn main() {
    let a = Rc::new(Cons(5, Rc::new(Cons(10, Rc::new(Nil)))));
    println!("count after creating a = {}", Rc::strong_count(&a));
    let b = Cons(3, Rc::clone(&a));
    println!("count after creating b = {}", Rc::strong_count(&a));
    {
        let c = Cons(4, Rc::clone(&a));
        println!("count after creating c = {}", Rc::strong_count(&a));
    }
    println!("count after c goes out of scope = {}",
Rc::strong_count(&a));
}
```

We share by counting the # of users

count after creating a = 1

count after creating b = 2

count after creating c = 3

count after c goes out of scope = 2

Structure is only dropped when # users = 0

Hey, isn't this lecture on interior mutability?

Interior mutability in Rust: what, why, how?

- Some data structures need to mutate one or more of their fields even when they are declared immutable!
- How do we get selective field mutability?

How Rc is implemented?

- clone takes a read-only reference to self, so the reference count can't be updated!

```
struct NaiveRc<T> {
    reference_count: usize,
    inner_value: T,
}

impl Clone for NaiveRc<T> {
    fn clone(&self) -> Self {
        self.reference_count += 1;
        // ...
    }
}
```

How Rc is implemented?

- We could implement a special, differently-named cloning function that takes &mut self
- bad for usability!
- So, how did they solve this problem in Rc?
- This is an instance of interior mutability.

Interior Mutability

- The heuristic is that avoiding mutability when possible is good.
- And yet, in some cases you need a few mutable fields in data structures.
- Interior mutability gives you that additional flexibility.
- To explain what interior mutability is, let’s first review exterior mutability.

Exterior Mutability

- Exterior mutability is the sort of mutability you get from mutable references (&mut T).
- Exterior mutability is checked and enforced at compile-time.

```
struct Foo { x: u32 };

let foo = Foo { x: 1 };
foo.x = 2; // The borrow checker will complain about this
and abort compilation

let mut bar = Foo { x: 1 };
bar.x = 2; // 'bar' is mutable, so you can change the
content of any of its fields
```

Interior Mutability

- Interior mutability is when you have an immutable reference (i.e., &T) but you can mutate the data structure.

```
struct Point { x: i32, y: i32 }
```

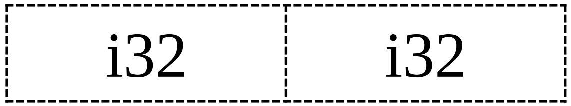
- An immutable *Point* can be seen as an immutable memory chunk. Now, consider a slightly different, magically-enhanced MagicPoint:

```
struct MagicPoint { x: i32, y: Magic<i32> }
```

- For now, ignore how Magic works, and think of it as a pointer to a mutable memory address, a new layer of indirection.

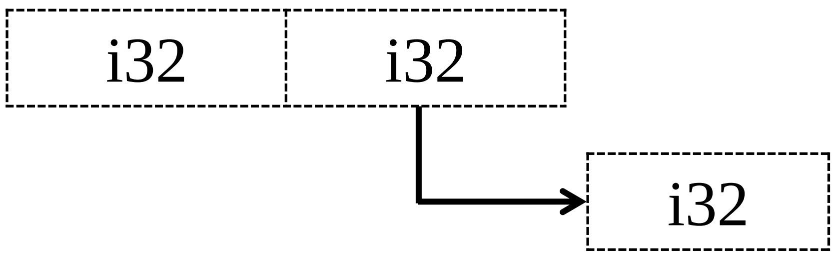
Interior Mutability

Point



MagicPoint

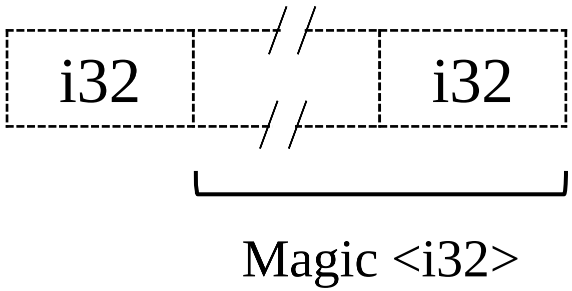
- **Logical:**



Immutable

mutable

- **In memory:**



Interior Mutability

- It is a pointer to a mutable memory address, a new layer of indirection.
- If you have an immutable MagicPoint, you can't assign new values to any of its fields.
- You don't need to change the content of y, only the destination of that magical pointer.
- Even though the API for Magic will make it seem as if you're relying on indirection to access and update the wrapped value.
- When relying on interior mutability, you are giving up the compile-time safety guarantees that exterior mutability gives you.

How can we use Interior Mutability?

- Rust standard library provides two wrappers, *std::cell::Cell* and *std::cell::RefCell*, that allow us to introduce interior mutability. **Do Magic**
- Both wrappers give up compile-time borrow checking on the inner value, but give different safety guarantees and serve different purposes.
- RefCell makes run-time borrow checks, while Cell does not.

1- Using Cell

- Cell is quite simple to use: you can read and write a Cell's inner value by calling get or set on it.

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

fn foo(cell: &Cell<u32>) {
    let value = cell.get();
    cell.set(value * 2);
}

fn main() {
    let cell = Cell::new(0);
    let value = cell.get();
    let new_value = cell.get() + 1;
    foo(&cell);
    cell.set(new_value); // oops, we clobbered the work
done by foo
}
```

2- Using RefCell

- RefCell requires to call borrow or borrow_mut (immutable and mutable borrows) before using it, yielding a pointer to the value.

```
use std::cell::RefCell;
fn main() {
    let x = 42;
    let rc = RefCell::new(x);
}
```

Which to pick?

	Cell	RefCell
Semantics	Copy	Move
Provides	Values	References
Panics?	Never	1. Mutable borrow and immutable borrow. 2. More than one mutable borrow.
Use with	Primitive types	Clone types

```
use std::cell;
use std::cell::RefCell;
fn main() {
 let x = 42;
 let c = cell::Cell::new(x);
 println!("value1: {}", c.get());
 c.set(0); //lying to compiler.
 println!("value2: {}", c.get());

 let rc = RefCell::new(x);
 let b1 = rc.try_borrow();
 if let Err(e) = b1 {
  println!("error1: {}", e);
  return;
 }
 let r1 = b1.unwrap();
 println!("value3: {}", *r1);
 let b2 = rc.try_borrow_mut();
 if let Err(e) = b2 {
  println!("error2: {}", e);
  return;
 } //Never reached!
 let r2 = b2.unwrap();
 println!("value4: {}", *r2);
}
```

Output:

```
value1: 42
value2: 0
value3: 42
error2: already borrowed
```

```rust
use std::cell;
use std::cell::RefCell;
fn main() {
 let x = 42;
 let c = cell::Cell::new(x);
 println!("value1: {}", c.get());
 c.set(0); //lying to compiler.
 println!("value2: {}", c.get());

 let rc = RefCell::new(x);
 let b1 = rc.try_borrow();
 if let Err(e) = b1 {
  println!("error1: {}", e);
  return;
 }
 let r1 = b1.unwrap();
 println!("value3: {}", *r1);
 let b2 = rc.try_borrow();
 if let Err(e) = b2 {
  println!("error2: {}", e);
  return;
 } //Reached here now!
 let r2 = b2.unwrap();
 println!("value4: {}", *r2);
}
```

Output:

```
value1: 42
value2: 0
value3: 42
value4: 42
```

```rust
use std::cell;
use std::cell::RefCell;
fn main() {
 let x = 42;
 let c = cell::Cell::new(x);
 println!("value1: {}", c.get());
 c.set(0); //lying to compiler.
 println!("value2: {}", c.get());

 let rc = RefCell::new(x);
 let b1 = rc.borrow();
 if let Err(e) = b1 {
  println!("error1: {}", e);
  return;
 }
 let r1 = b1.unwrap();
 println!("value3: {}", *r1);
 let b2 = rc.try_borrow();
 if let Err(e) = b2 {
 println!("error2: {}", e);
 return;
 }
 let r2 = b2.unwrap();
 println!("value4: {}", *r2);
}
```

```
_if let Err(e) = b1 {
   |         ^^^^^^   -- this
expression has type
`std::cell::Ref<'_, {integer}>`
   |         |
   |         expected struct
`std::cell::Ref`, found enum
`std::result::Result`
   |
   = note: expected struct
`std::cell::Ref<'_, {integer}>`
                found enum
`std::result::Result<_, _>`
```

Borrow and try_borrow

pub fn borrow(&self) -> Ref<T>*

- Immutably borrows the wrapped value.
- The borrow lasts until the returned Ref exits scope.
- Multiple immutable borrows can be taken out at the same time.
- It panics if the value is currently mutably borrowed.

pub fn try_borrow(&self) -> Result<Ref<T>, BorrowError>

- The non-panicking variant of borrow.
- Immutably borrows the wrapped value, returning an error if the value is currently mutably borrowed.

Borrow and try_borrow

pub fn borrow(&self) -> Ref<T>*

```
use std::cell::RefCell;

let c = RefCell::new(5);

let borrowed_five = c.borrow();
let borrowed_five2 = c.borrow();
```

```
let c = RefCell::new(5);
let _m = c.borrow_mut();
let _b = c.borrow(); // panics
```

pub fn try_borrow(&self) -> Result<Ref<T>, BorrowError>

```
use std::cell::RefCell;

let c = RefCell::new(5);
{
 let m = c.borrow_mut();
 assert!(c.try_borrow().
        is_err());
}
{
 let m = c.borrow();
 assert!(c.try_borrow().is_ok());
}
```

* What is **Ref<T>**?

Ref<T>

- Wraps a borrowed reference to a value in a RefCell box.
- A wrapper type for an immutably borrowed value from a RefCell<T>.

```
use std::cell::{RefCell, Ref};

let c = RefCell::new((5, 'b'));
let b1: Ref<(u32, char)> = c.borrow();
let b2: Ref<u32> = Ref::map(b1, |t| &t.0);
assert_eq!(*b2, 5)
```

Borrow_mut and try_borrow_mut

pub fn borrow_mut(&self) -> RefMut<T>*

- Mutably borrows the wrapped value.

- 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.

- It panics if the value is currently borrowed.

pub fn try_borrow_mut(&self) -> Result<RefMut<T>,BorrowMutError>

- The non-panicking variant of borrow_mut.

- mutably borrows the wrapped value, returning an error if the value is currently borrowed.

Borrow_mut and try_borrow_mut

pub fn borrow_mut(&self) -> RefMut<T>*

```
use std::cell::RefCell;

let c = RefCell::new(5);

*c.borrow_mut() = 7;

assert_eq!(*c.borrow(), 7);
```

```
let c = RefCell::new(5);
let _m = c.borrow();
let _b = c.borrow_mut(); panics
```

pub fn try_borrow_mut(&self) -> Result<RefMut<T>,BorrowMutError>

```
use std::cell::RefCell;

let c = RefCell::new(5);
{
 let m = c.borrow();
 assert!(c.try_borrow_mut().
        is_err());
}
 assert!(c.try_borrow_mut().
        is_ok());
```

Consider these definitions

```
use std::cell::Cell;
#[derive(Copy, Clone)]
struct House {
bedrooms: u8,
}

impl Default for House {
fn default() -> self {
House { bedrooms: 1 }
}
}
```

- Go Investgiate cell at:

https://doc.rust-lang.org/std/cell/

Once you understand cell go to next slide

Go look up cell documentation – what does this do?

```
fn main() {
let my_house = House { bedrooms: 2 };
let my_dream_house = House { bedrooms: 5 };

let my_cell = Cell::new(my_house);
println!("My house has {} bedrooms.", my_cell.get().bedrooms);

my_cell.set(my_dream_house);
println!("My new house has {} bedrooms.", my_cell.get().bedrooms);

let my_new_old_house = my_cell.replace(my_house);
println!("My house has {} bedrooms, it was better with {}",my_cell.get().bedrooms,my_new_old_house.bedrooms);

let my_new_cell = Cell::new(my_dream_house);
my_cell.swap(&my_new_cell);
println!("my current house has {} bedrooms! (my new house {})",my_cell.get().bedrooms,my_new_cell.get().bedrooms);

let my_final_house = my_cell.take();
println!("My final house has {} bedrooms, the shared one {}",my_final_house.bedrooms,my_cell.get().bedrooms);
}
```

Did you get it correct?

My house has 2 bedrooms.

My new house has 5 bedrooms.

My house has 2 bedrooms, it was better with 5

my current house has 5 bedrooms! (my new house 2)

My final house has 5 bedrooms, the shared one 1