Lecture 9

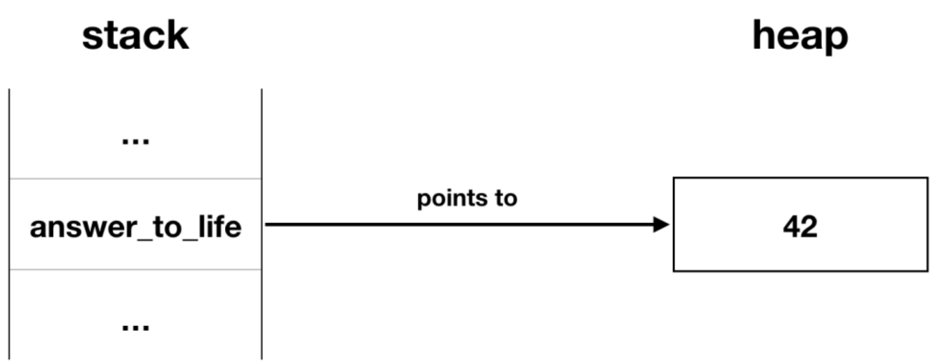
Interior Mutability

Rust Has Many Smart Pointers

- For example:

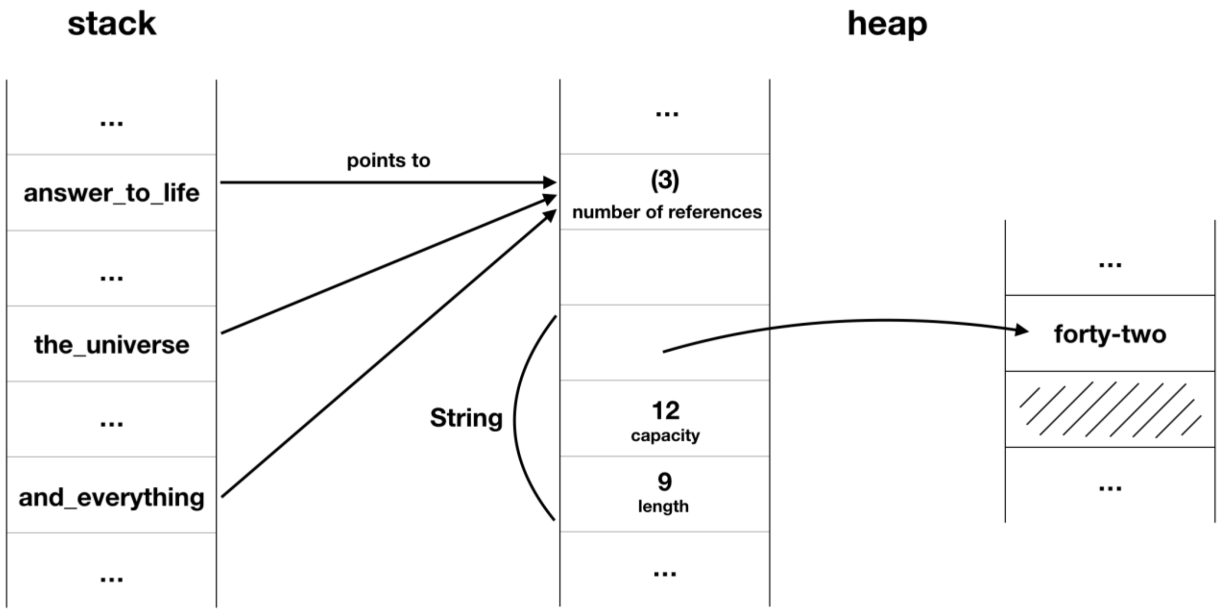
  Box<T> **Rc<T>** **Cell<T>** **RefCell<T>** **.....**

- **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.



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.



Rc<T> (Reference Counting)

- Whether you use a Box<T> or an Rc<T>, the Box or Rc itself is on the stack, but the data they "contain" lives on the heap.
- Box and Rc are nothing else than references (pointers) to objects stored on the heap.
- One provides shared ownership, the other doesn't.

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 }
```

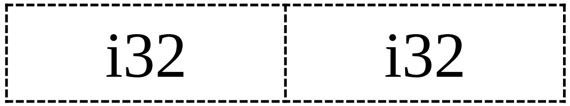
- An immutable *Point* can be seen as an immutable memory chunk. Now, consider a slightly different, magically-enhanced MagicPoint:

```
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.

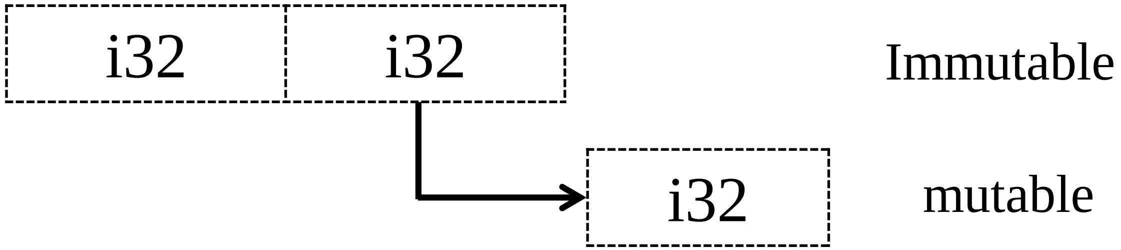
Interior Mutability

Point

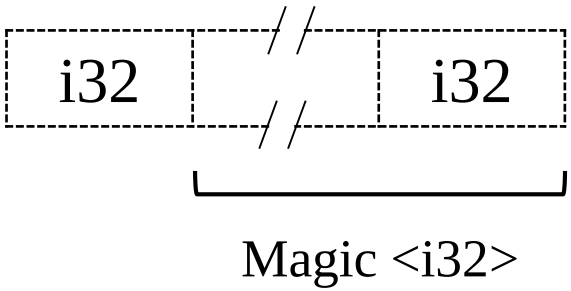


MagicPoint

- **Logical:**



- **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
}
```

Side note: Using Options by example

- Rust avoids nulls in the language.
- Instead, we can represent a value that might or might not exist with the Option type.
- In Rust, *Option<T>* is an enum that can either be None (no value present) or Some(x) (some value present).

```rust
struct FullName {
    first: Option<String>,
    last: String,
}
fn main(){
 let alice = FullName {
  first: Some(String::from("Alice")),
  last: String::from("Johnson")};

 let john = FullName {
  first: None,
  last: String::from("Doe")};

 println!("Alice's first name is {}", alice.first.unwrap()); //
       prints Alice
 println!("John's first name is {}", john.first.unwrap()); //
       panics
}
```

```
Alice's first name is Alice
thread 'main' panicked at 'called `Option::unwrap()` on a `None` value', src\libcore\opti
on.rs:378:21
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace.
error: process didn't exit successfully: `target\debug\ece421lab2.exe` (exit code: 101)
```

Side note: Using Options by example

- *pub fn unwrap(self) -> T* : Unwraps a result.

Yields the content of an Ok

Panics if the value is an Err, with a panic message provided by the Err's value.

```
let x: Result<u32, &str> = Ok(2);
assert_eq!(x.unwrap(), 2);

let x: Result<u32, &str> = Err("emergency failure");
x.unwrap(); // panics with `emergency failure`
```

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);
}
```

Output:

```
error[E0599]: no method named
`unwrap` found for type
`std::cell::Ref<'_, {integer}>`
in the current scope
  --> src\main.rs:19:17
19 |     let r1 = b1.unwrap();
   |                 ^^^^^^
method not found in
`std::cell::Ref<'_, {integer}>`
```

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());
```