CIS 1905

Lecture 6: Smart Pointers

Beyond simple ownership

Logistics

HW3 deadline changes

Final Project Proposals

pollev.com/cis1905rust915

Some Reminders

Ownership Rules

1. Each value has an owner
2. There can only be one owner at a time
3. When the owner goes out of scope, the value is dropped

Borrowing Rules

1. At any given time, you can have either one mutable reference or many immutable references
2. References must always be valid

Some Reminders

Ownership Rules

1. Each value has an owner
2. ~~There can only be one owner at a time~~
3. When the owner goes out of scope, the value is dropped

Borrowing Rules

1. ~~At any given time, you can have either one mutable reference or many immutable references~~
2. References must always be valid

We can break* some of these rules with Smart Pointers.

The other ones require `unsafe` rust.

Smart Pointers

`Box<T>`

Single ownership on the heap

`Cow<'a, B>`

Lazy copies

`Rc<T>/Arc<T>`

Multiple ownership (without mutation)

`Vec<T>/…`

Dynamically sized collections

`Rc<RefCell<T>>/…`

Multiple ownership (with mutation)

`&[T], &dyn T`

Wide pointers

Box<T>

- Just allocates T on the heap
- T is owned by the Box; when the Box is dropped, T's heap space is freed
- The Box's control of its heap space is exclusive
- (if you're familiar with C++, this is like `std::unique_ptr`)

Enforces all of Rust's borrowing rules *at compile time*

```rust
fn main() {
    // Allocate the value 10.0 on the heap
    let x = Box::new(10.0);
    assert_eq!(*x, 10.0);

    // Deallocate the heap value
    drop(x);
}
```

Box<T> Use Case: Recursive Data Types

```
struct BTree<T> {      recursive type `BTree` has infinite size
    value: T,
    left_child: Option<Self>,      recursive without indirection
    right_child: Option<Self>,
}
```

size(BTree) = *size*(T) + *size*(Option<BTree>) = ∞

Fix the infinite recursion with Box:

```
struct BTree<T> {
    value: T,
    left_child: Option<Box<Self>>,
    right_child: Option<Box<Self>>,
}
```

size(BTree) = *size*(T) + *size*(Box<BTree>)
= *size*(BTree) = *size*(T) + *size*(usize)

Rc<T>

What if you want more than one owner?

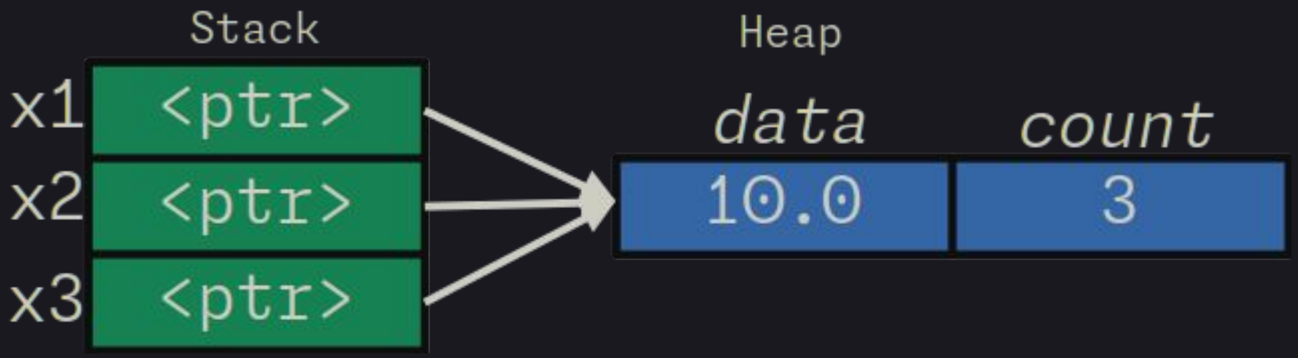
Rc stands for “reference counted”.

- The data lives on the **heap**
- The references live on the **stack**

Like a normal reference, but *everyone* is an owner—and you only get immutable access to the data

How does it work? *(simplified version)*

- `Rc::new(data)`
  - Allocates space on the heap for `data`, sets count to 1
- `impl Clone`
  - Copies the *pointer* and increments `count` by 1
  - Doesn't actually clone the `data`!
- `impl Drop`
  - Decrement `count`
  - If reference count is 0, then free the `data`



Doesn't this break the ownership rules?

Kind of, but not really

- Multiple owners point to the same data, but it is **immutable**
  - In this way, Rc behaves like a reference
- If all the owners go out of scope, the data is dropped
  - In this way, Rc behaves like an owned value

So yes, it breaks the rules, but it also maintains safety.

```rust
fn main() {
    let a = Rc::new(5.0);
    // data gets allocated
    let b = a.clone();
    {
        let c = b.clone();
        println!("{}", c);
    }
    drop(b);
    drop(a);
    // data gets dropped
}
```

What does this look like in practice?

You should use Rc when:

- You have some *really big* piece of data
- It's expensive to **clone**
- Lots of things need **access** to it
- It doesn't make sense to have a **single** owner
- You don't know **how long** it will be needed
- You want it to be **easy**

Cloning is free!

```rust
/// A really big texture that's expensive to clone around
pub struct Texture;
impl Texture {
    pub fn load(path: &str) -> Rc<Self> {
        unimplemented!()
    }
}

/// A sprite in your game
pub struct Sprite {
    pub name: String,
    pub texture: Rc<Texture>,
}
impl Sprite {
    pub fn new(name: impl ToString, texture: Rc<Texture>) -> Self {
        Self {
            name: name.to_string(),
            texture,
        }
    }
}

fn main() {
    let texture = Texture::load("big_texture.png");

    let player = Sprite::new("Main Player", texture.clone());

    let enemy1 = Sprite::new("Enemy", texture.clone());
    let enemy2 = Sprite::new("Enemy", texture.clone());
    let enemy3 = Sprite::new("Enemy", texture.clone());
}
```

Cyclical references

Beware of cyclical references!

The reference count will never hit zero, so your data will never be freed.

- This is a memory leak

Solution: Weak<T>

- Same as a Rc<T>, but doesn't increment the count (the data is **not** guaranteed to be there)


```rust
let weak = Rc::downgrade(&strong);
```

Detour: `Cell<T>`

- Cell is not a pointer type (but it builds towards one...)
- Let's you *mutate* an *immutable* value in a controlled way

In general, it's useless...

- You can't read the value, only replace it with a new one

But for `Copy` types...

- You can get the value by copying it, so it becomes useful

Doesn't this break the rules?

- Single owner: not broken, cells are still owned values
- One writer XOR multiple readers: not broken, since you're never actually mutating anything; you're just *replacing* one value with another

Detour: RefCell<T>

- Also not a pointer! Allows us to *mutate* any *immutable* value, not just Copy ones!
- Enforces the borrow checker's rules *at runtime*

```rust
let test: RefCell<String> = RefCell::new(String::from_str("Hello").unwrap());
{
    let inner_value: Ref<'_, String> = test.borrow();
    println!("{inner_value}") // Prints "Hello"
}
{
    let mut mutable_ref: RefMut<'_, String> = test.borrow_mut();
    mutable_ref.push_str(string: " World!");
}
{
    let inner_value: Ref<'_, String> = test.borrow();
    println!("{inner_value}") // Prints "Hello World!", even though test is immutable!
}
```

Detour: `RefCell<T>`

So this is fine:

not `mut`

```rust
let test: RefCell<String> = RefCell::new(String::from_str("Hello").unwrap());
{
    let inner_value: Ref<'_, String> = test.borrow();
    println!("{inner_value}") // Prints "Hello"
}
{
    let mut mutable_ref: RefMut<'_, String> = test.borrow_mut();
    mutable_ref.push_str(string: " World!");
}
{
    let inner_value: Ref<'_, String> = test.borrow();
    println!("{inner_value}") // Prints "Hello World!", even though test is immutable!
}
```

Detour: `RefCell<T>`

But this isn't:

```
{
    let immutable_ref: Ref<'_, String> = test.borrow();
    let mut mutable_ref: RefMut<'_, String> = test.borrow_mut(); // This will panic!!
    println!("{immutable_ref}");
    mutable_ref.push_str(string: "oops");
}
```

Why isn't this ok?

`Rc<RefCell<T>>`

What if you want more than one owner and mutability via runtime borrow-checking?

- For when you really want to wing it 😀
- ~Almost~ a Java reference; like `std::shared_ptr` in C++
- Allows multiple owners to mutate the same heap location, so long the borrow rules are upheld at runtime

Let’s make a graph!

Attempt 1

```
struct Node {
    value: i32,
    children: Vec<Box<Node>>,
}
```

Well, that’s not right…

- Each node needs to own its children, so we can’t make cycles :(
- We need *multiple ownership!*

Let’s make a graph!

Attempt 2

```rust
struct Node {
    value: i32,
    children: Vec<Rc<Node>>,
}
```

Well, that’s not right…

- We can’t modify anything anymore! Our references to nodes are immutable
- So we still can’t make a cycle…
- We need *runtime-checked borrowing!*

Let’s make a graph!

Attempt 3

```
struct Node {
    value: i32,
    children: Vec<Rc<RefCell<Node>>>,
}
```

Yay cycles! (But watch out for cycles)

```
let node1: Rc<RefCell<Node>> = Rc::new(RefCell::new(Node { value: 1, children: vec![] }));
let node2: Rc<RefCell<Node>> = Rc::new(RefCell::new(Node { value: 2, children: vec![] }));
RefCell::borrow_mut(self: &node1).children.push(Rc::clone(self: &node2));
RefCell::borrow_mut(self: &node2).children.push(node1);
```

`Cow<'a, B>`

what?

Clone **O**n **W**rite

*Essentially a "lazy clone"

*You probably won't use this one

```
pub enum Cow<'a, B> { ... }
```

`Borrowed(&'a B)`

When you call `Cow::from(x)`, you get a `Borrowed(&x)`.

This is essentially free since all you did is take a reference. It will stay this way until you need to mutate the data.

`Owned(<B as ToOwned>::Owned)`

There are two ways to get this

1. `Cow::to_mut(&mut self)`
   Clones the data, and gives you a mutable reference

2. `Cow::into_owned(self)`
   Clones the data and gives it to you

`Cow` only clones the data once you need mutable access, then it stays that way

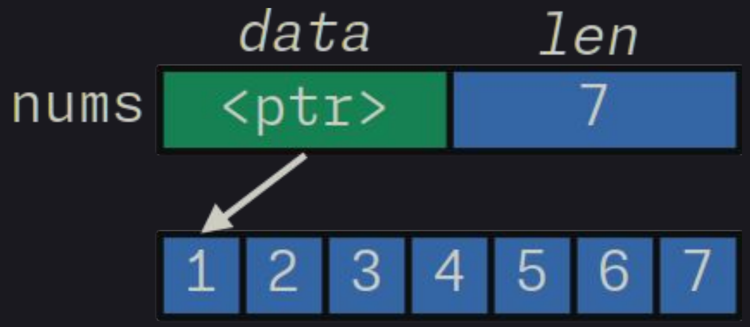
Wide pointers

Some pointers are called "wide pointers", which means they carry extra information

There are two kinds of wide pointers

- Slices (`&[T]`), which store the **length** of the slice (this also includes `&str`)
- Trait objects (`&dyn T`), which store a pointer to the **vtable**

The pointers are "wide" because they take up **2** words instead of just **1**.



Slice example

Wide pointers

They don't have to be behind *references*, they just have to be *pointers*

Some weird looking but syntactically-correct examples:

```
Box<dyn Error>
```

Owning pointer to some type that implements Error

```
Rc<str>
```

Reference-counted pointer to an immutable string slice

```
Cow<[i32]>
```

Clone-on-writeable integer array slice

Thread-Safe Smart Pointers

Most of the normal smart pointers are not thread safe for efficiency reasons, so Rust gives us thread-safe alternatives

Normal	Thread-Safe
`Rc`	`Arc`: atomically reference counted Rc
`Cell<T>`	`AtomicT`: (like `AtomicI32`)
`RefCell<T>`	`RWLock<T>`: one writer XOR many readers `Mutex<T>`: one writer XOR one reader

unsafe

If smart pointers break the ownership and borrowing rules, how are they implemented?

1. Unsafe rust
   a. Lets you do bad things (i.e. work with raw (C-style) pointers, work with uninitialized memory, transmute types, etc.)
   b. Must be contained within an `unsafe` block
   c. By writing `unsafe` code, you "promise" not to cause undefined behavior or otherwise break Rust's contracts
2. Compiler intrinsics
   a. Some types (such as `Box`) are so fundamental that they are implemented at the compiler level

```rust
fn main() {
    let x: Box<i32> = unsafe { Box::new(MaybeUninit::uninit().assume_init()) };
    println!("x = {}", x);
}
```

`x = -1824071680` (undefined behavior)

Challenge: Binary Tree

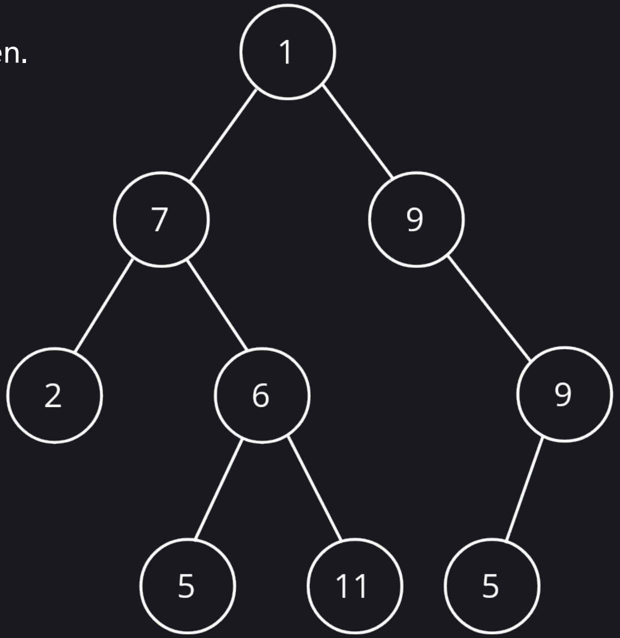
Implement a binary tree on your own!

Each vertex should have associated data and two optional children.

You should support the following operations:

- `new(data: T)`
- `add_left_child(child: Self)`
- `add_right_child(child: Self)`
- `contains(data: T)`

You may want to `#[derive(Debug)]` to test your tree.



Sample Solution

```rust
#[derive(Clone, Debug)]
struct BTree<T> {
    value: T,
    left: Option<Box<BTree<T>>>,
    right: Option<Box<BTree<T>>>,
}
```

```rust
impl<T> BTree<T> {
    fn new(value: T) -> Self {
        Self {
            value,
            left: None,
            right: None,
        }
    }

    fn add_left_child(&mut self, child: Self) {
        self.left = Some(Box::new(child));
    }

    fn add_right_child(&mut self, child: Self) {
        self.right = Some(Box::new(child));
    }

    fn contains(&self, value: &T) -> bool
    where
        T: PartialEq,
    {
        return self.value == *value
            || self.left.as_ref().is_some_and(|l| l.contains(value))
            || self.right.as_ref().is_some_and(|r| r.contains(value));
    }
}
```

Challenge: Improved Binary Tree

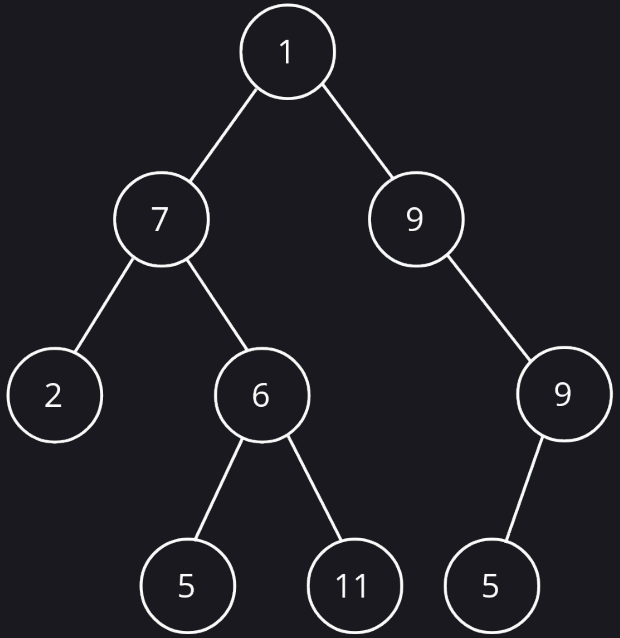
Now, make it hold pointers to both the parent *and* the child.

You should support the following additional operations:

- `get_left_child(&self)`
- `get_right_child(&self)`
- `get_parent(&self)`

Hint: change `new` to return type of `Rc<RefCell<Self>>`

Hint: change `add_left_child` and `add_right_child` to take in `parent` and `child` instead of `self` and `child`



Sample Solution

```rust
#[derive(Clone, Debug)]
struct BTree<T> {
    value: T,
    left: Option<Rc<RefCell<BTree<T>>>>,
    right: Option<Rc<RefCell<BTree<T>>>>,
    parent: Option<Rc<RefCell<BTree<T>>>>,
}
```

Sample Solution

```
impl<T> BTree<T> {
    fn new(value: T) -> Rc<RefCell<Self>> {
        Rc::new(RefCell::new(Self {
            value,
            left: None,
            right: None,
            parent: None,
        }))
    }

    fn add_left_child(parent: Rc<RefCell<Self>>, child: Rc<RefCell<Self>>) {
        child.borrow_mut().parent = Some(parent.clone());
        parent.borrow_mut().left = Some(child);
    }

    fn add_right_child(parent: Rc<RefCell<Self>>, child: Rc<RefCell<Self>>) {
        child.borrow_mut().parent = Some(parent.clone());
        parent.borrow_mut().right = Some(child);
    }
```

Sample Solution

```
fn contains(&self, value: &T) -> bool
where
    T: PartialEq,
{
    return self.value == *value
        || self
            .left
            .as_ref()
            .is_some_and(|l| l.borrow().contains(value))
        || self
            .right
            .as_ref()
            .is_some_and(|r| r.borrow().contains(value));
}
```

Sample Solution

```
    fn get_left_child(&self) -> Option<Rc<RefCell<Self>>> {
        self.left.clone()
    }

    fn get_right_child(&self) -> Option<Rc<RefCell<Self>>> {
        self.right.clone()
    }

    fn get_parent(&self) -> Option<Rc<RefCell<Self>>> {
        self.parent.clone()
    }
}
```