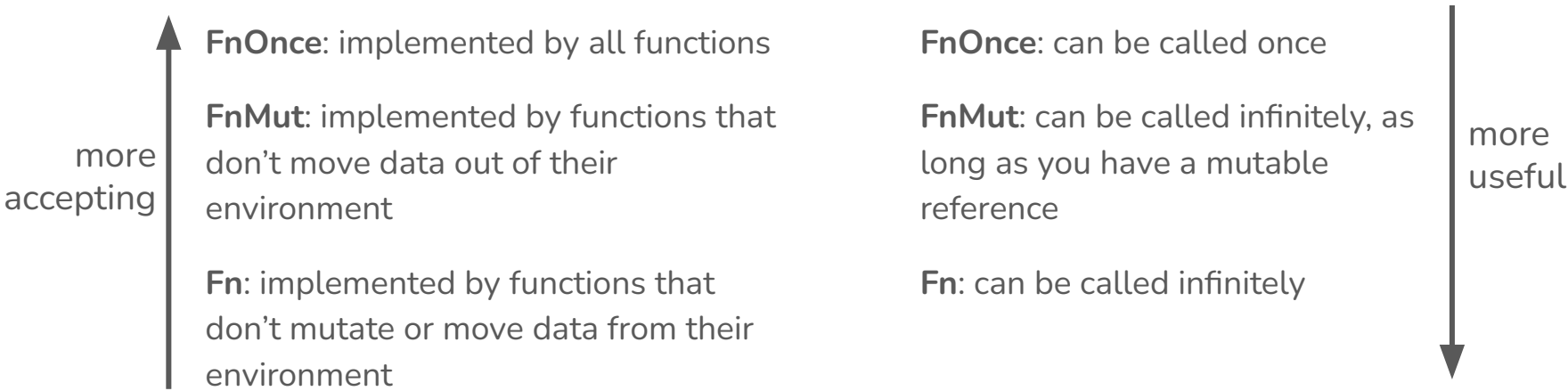


Lecture 7

Smart Pointers and Trait Objects

From PLQ: What Fn type should my function take?



Writing a function that takes another function? Take the highest **Fn...** trait you can that still suits your needs

Lifetimes

```
fn identity<'a>(s1: &'a str) -> &'a str {  
    s1  
}
```

Can be called with **any** reference (including static)

```
fn longer<'a>(s1: &'a str, s2: &'a str)  
    -> &'a str { ... }
```

Reference of input lifetimes must overlap

```
fn get_course() -> &'static str {  
    let x = "cis1905";  
    return x;  
}
```

static is special

*Lifetimes only make sense within the context of
a single function*

Quiz from last time

```
struct Foo<'a> {  
    bar: &'a i32  
}
```

```
fn baz<'a, 'b>(f: &'a Foo<'b>) -> &'??? i32  
{ /* omitted */ }
```

```
fn baz<'a, 'b>(f: &'a &'b i32) -> &'??? i32  
{ /* omitted */ }
```

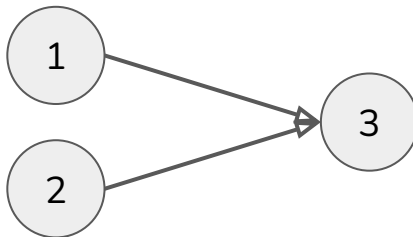
Will this compile? **No!**

Two separate lifetimes in the input

- can't infer output lifetime without ambiguity

Smart Pointers

Back to lists



```
struct List<T> {  
    value: T,  
    next: Option<Box<List<T>>>,  
}
```

```
impl<T> List<T> {  
    fn new(value: T) -> Self {  
        List { value, next: None }  
    }  
}
```

```
fn main() {  
    let mut list1 = List::new(1);  
    let mut list2 = List::new(2);  
    let node = Box::new(List::new(3));  
  
    list1.next = Some(node);  
    list2.next = Some(node);  
}
```

```
error[E0382]: use of moved value: `node`  
--> list.rs:17:21  
16 |     list1.next = Some(node);  
   |                      ---- value moved here  
17 |     list2.next = Some(node);  
   |                      ^^^^ value used here after move
```

Why `as_mut()`?

```
pub fn as_mut<T>(&mut Option<T>) -> Option<&mut T>
```

```
pub fn as_ref<T>(&Option<T>) -> Option<&T>
```

```
let mut x: Option<u32> = Some(5);
```

```
let as_mut_ref: &mut Option<u32> = &mut x;
```

```
as_mut_ref.unwrap() = 7; // bad!
```



type: `u32` (not assignable)

```
*as_mut_ref.as_mut().unwrap() = 7; // good!
```



type: `&mut u32` (assignable)

Recall: why ownership?

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

One owner -> statically determine when values
can be destructed (when their owner is no
longer accessible)

Is multiple ownership bad?

Yes—why?

- How to tell when values should be destructed?



**track at run-time instead
of compile-time**

No—why?

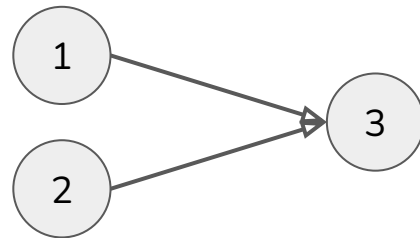
- Sometimes real programs need shared ownership

Multiple ownership via reference counting

Reasoning about shared ownership statically is impossible ->
record ownership data at runtime

- When a clone is made, increment **refcount**
- When an owner goes out of scope, decrement **refcount**
- When **refcount** is 0, deallocate

Rc: reference counted pointer



```
use std::rc::Rc;
```

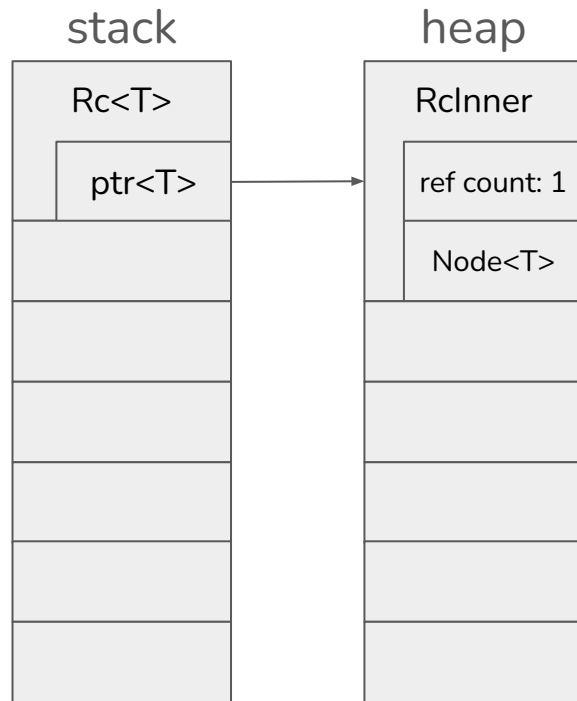
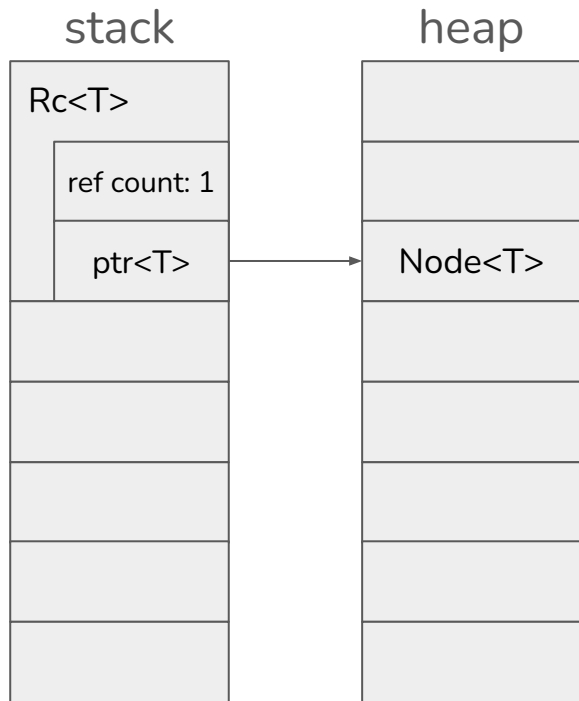
```
struct List<T> {  
    value: T,  
    next: Option<Rc<List<T>>>,  
}  
  
impl<T> List<T> {  
    fn new(value: T) -> Self {  
        List { value, next: None }  
    }  
}
```

```
fn main() {  
    let mut list1 = List::new(1);  
    let mut list2 = List::new(2);  
    let node = Rc::new(List::new(3));  
  
    list1.next = Some(Rc::clone(&node));  
    list2.next = Some(Rc::clone(&node));  
}
```

three owners of Node(3)

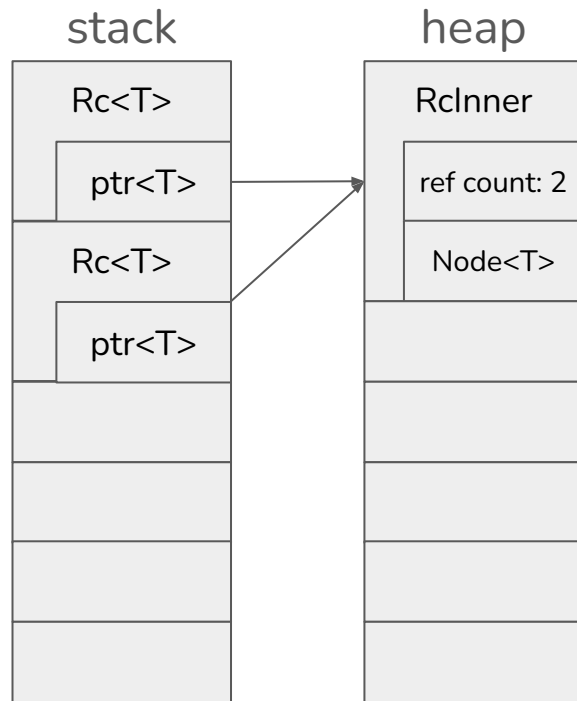
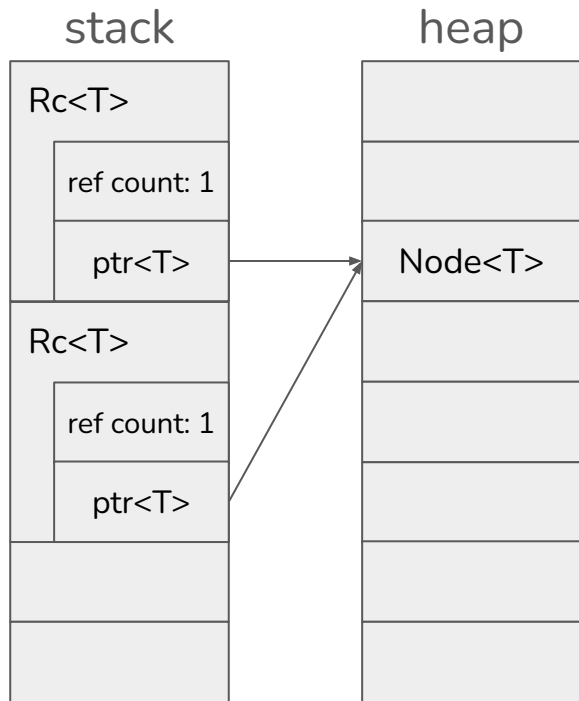
Rc allows shared ownership by figuring out when to run destructor at run-time

How to implement Rc?



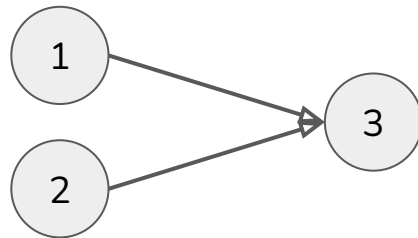
A or B?

How to implement Rc?



A or B?

Rc: reference counted pointer



```
use std::rc::Rc;
```

```
struct List<T> {  
    value: T,  
    next: Option<Rc<List<T>>>,  
}  
  
impl<T> List<T> {  
    fn new(value: T) -> Self {  
        List { value, next: None }  
    }  
}
```

```
fn main() {  
    let mut list1 = Rc::new(List::new(1));  
    let mut list2 = Rc::new(List::new(2));  
    let node = Rc::new(List::new(3));  
  
    list1.next = Some(Rc::clone(&node));  
    list2.next = Some(Rc::clone(&node));  
}
```

But, **next** isn't a field on **Rc**?

Types that are like pointers implement the **Deref** trait so that they can be treated like the inner type

auto-deref:

`&mut Rc<List<i32>>`



`&mut List<i32>`

Using Rc as Garbage Collection

Don't want to think about ownership? Just wrap everything in Rc

- Don't actually do this, but it works in theory

Rc summary

Some data structures require shared ownership

- Reuse data instead of cloning
- graphs, linked lists, DAGs

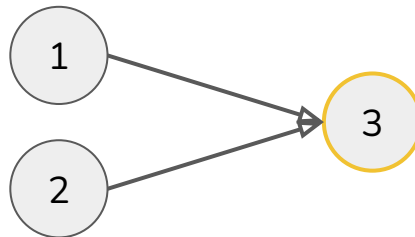
Reasoning about shared ownership statically is impossible -> record ownership data at runtime

- When a clone is made, increment **refcount**
- When an owner goes out of scope, decrement **refcount**
- When **refcount** is 0, deallocate

Rc is a primitive form of garbage collection

- e.g. in Python, every value is reference counted

Shared ownership woes



```
use std::rc::Rc;
```

```
struct List<T> {  
    value: T,  
    next: Option<Rc<List<T>>>,  
}
```

```
impl<T> List<T> {  
    fn new(value: T) -> Self {  
        List { value, next: None }  
    }  
}
```

```
fn main() {  
    let mut list1 = List::new(1);  
    let mut list2 = List::new(2);  
    let node = Rc::new(List::new(3));  
  
    list1.next = Some(Rc::clone(&node));  
    list2.next = Some(Rc::clone(&node));  
    node.value = 5;  
}
```

```
error[E0594]: cannot assign to data in an `Rc`  
--> refcell.rs:22:3  
   |  
22 |     node.value = 5;  
   |     ^^^^^^^^^^^^^ cannot assign
```

Shared ownership ≠ shared mutability

Since `Rc`ed values can have multiple owners,
never safe to give out mutable references to
inner type `T`!

How to mutate shared values without violating
Rust's safety guarantees? (no dangling references)

```
fn main() {  
    let mut v1: Rc<Vec<i32>> = Rc::new(vec![1]);  
    let mut v2: Rc<Vec<i32>> = Rc::copy(&v1);  
  
    let v1_mut: &mut Vec<i32> = &mut *v1;  
    let v2_mut: &mut Vec<i32> = &mut *v2;  
    let first: &mut i32 = &mut v2_mut[0];  
    v1_mut.pop();  
  
    println!("{:?}", first); // dangling!  
}
```

Shared ownership ≠ shared mutability

Since `Rc`ed values can have multiple owners,
never safe to give out mutable references to
inner type `T`!

How to mutate shared values without violating
Rust's safety guarantees? (no dangling references)

```
fn main() {  
    let mut v1: Rc<Vec<i32>> = Rc::new(vec![1]);  
    let mut v2: Rc<Vec<i32>> = Rc::copy(&v1);
```

`Rc`: shared ownership -> dynamically track owners

`??`: shared mutation -> dynamically track mutators

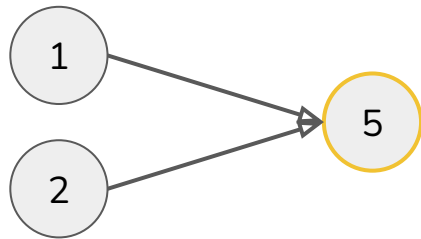
```
    let mut v1_mut = v1.borrow_mut();
```

```
    v1_mut.pop();
```

```
    println!("{:?}", first); // dangling!
```

```
}
```

List Attempt #3



```
use std::rc::Rc;

struct List<T> {
    value: T,
    next: Option<Rc<RefCell<List<T>>>>,
}

impl<T> List<T> {
    fn new(value: T) -> Self {
        List { value, next: None }
    }
}
```

```
fn main() {
    let mut list1 = List::new(1);
    let mut list2 = List::new(2);
    let node =
        Rc::new(RefCell::new(List::new(3)));

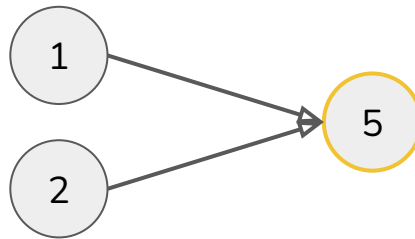
    list1.next = Some(Rc::clone(&node));
    list2.next = Some(Rc::clone(&node));
    node.borrow_mut().value = 5;
```

RefCell: count mutable/immutable references at run-time

- create new refs with **borrow/borrow_mut**
- panics if more than one **mut** OR **mut** and non-**mut** at same time



List Attempt #3



```
use std::rc::Rc;

struct List<T> {
    value: T,
    next: Option<Rc<RefCell<List<T>>>>,
}

impl<T> List<T> {
    fn new(value: T) -> Self {
        List { value, next: None }
    }
}
```

```
fn main() {
    let mut list1 = List::new(1);
    let mut list2 = List::new(2);
    let node =
        Rc::new(RefCell::new(List::new(3)));

    list1.next = Some(Rc::clone(&node));
    list2.next = Some(Rc::clone(&node));
    let ref1 = node.borrow_mut();
};
```

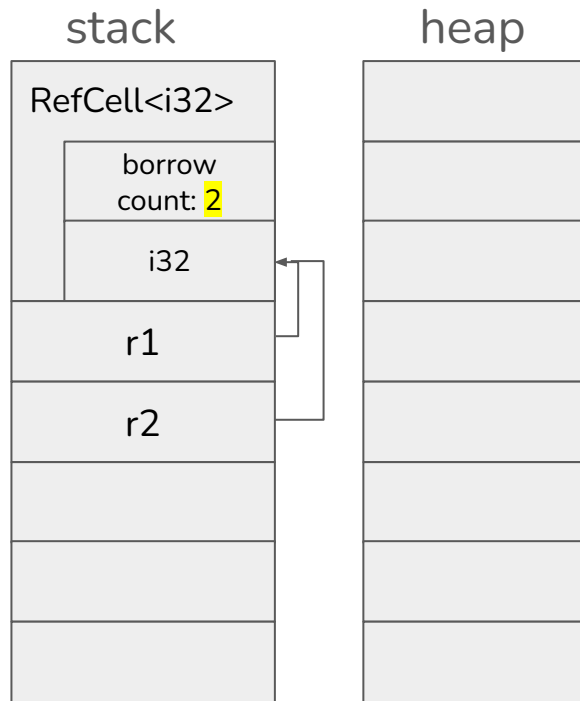
RefCell: count mutable/immutable references at run-time

- create new refs with **borrow/borrow_mut**
- panics if more than one **mut** OR **mut** and non-**mut** at same time

compiles ✓...
panics 😞

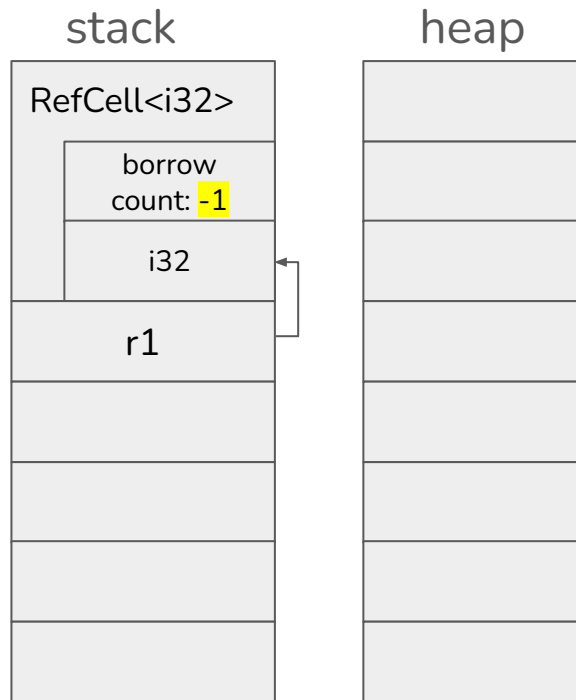
RefCell is not a reference/pointer!

```
use std::cell::RefCell;
fn main() {
    let v = RefCell::new(1);
    let r1 = v.borrow();
    let r2 = v.borrow();
}
```



RefCell is not a reference/pointer!

```
use std::cell::RefCell;
fn main() {
    let v = RefCell::new(1);
    let r1 = v.borrow_mut();
}
```



RefCell recapped

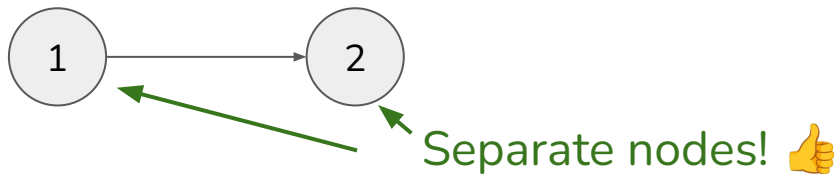
Sometimes the compiler can't statically verify that you follow the reference rules

- Offload reference checking to run-time

The Rule of References:

- **At any given time, you can have either one mutable reference or any number of immutable references.**
- References must always be valid.

Another RefCell example



```
struct List<T> {  
    pub value: T,  
    pub next: Option<Box<List<T>>>,  
}
```

```
impl<T> List<T> {  
    fn first(&mut self) -> &mut T { todo!() }  
    fn last(&mut self) -> &mut T { todo!() }  
    fn ends(&mut self) -> (&mut T, &mut T) {  
        (self.first(), self.last())  
    }  
}
```

error[E0499]: cannot borrow `\*self` as mutable more than once at a time

--> partition_bad.rs:22:20

21 | fn ends(&mut self) -> (&mut T, &mut T) {

22 | (self.first(), self.last())

^^^^

^^^^

|

|

second mutable borrow occurs here

first mutable borrow occurs here

A quick aside on computability

Halting problem:

“Writing a program that decides whether a turing machine halts on a given input is impossible”



Rice's theorem:

“Statically deciding any non-trivial property of a program is impossible”

Things you can't decide just by looking at a program

- Does this program leak memory?
- Does this program have a use-after-free bug?
- Does this function always produce the same output as another function?
- Does this program have a race condition?

Rust and decidability

Programs Rust would like to disallow for you at compile time

- Multiple mutable references at the same time
- Reference pointing to invalid memory
- etc.

Pick One:

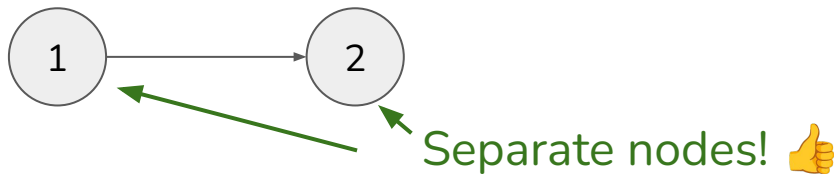
Unsound

- All valid programs are allowed 👍
- Some invalid programs are allowed 🚨

Incomplete

- Some valid programs aren't allowed 🚨
- All invalid programs aren't allowed. 👍

Another RefCell example



Incomplete

- Some valid programs aren't allowed 🚨
- All invalid programs aren't allowed. 👍

```
fn first(&mut self) -> &mut T { todo!() }  
fn last(&mut self)  -> &mut T { todo!() }  
fn ends(&mut self)  -> (&mut T, &mut T) {  
    (self.first(), self.last())  
}  
}
```

```
error[E0499]: cannot borrow `*self` as mutable more than once at a time  
--> partition_bad.rs:22:20
```

```
21 |     fn ends(&mut self) -> (&mut T, &mut T) {  
22 |         (self.first(), self.last())  
    |         ^^^^             ^^^^  
    |         |               |  
    |         first mutable borrow occurs here second mutable borrow occurs here
```

A quick aside on computability

Halting problem:

“Writing a program that decides whether a turing machine halts on a given input is impossible”



Rice's theorem:

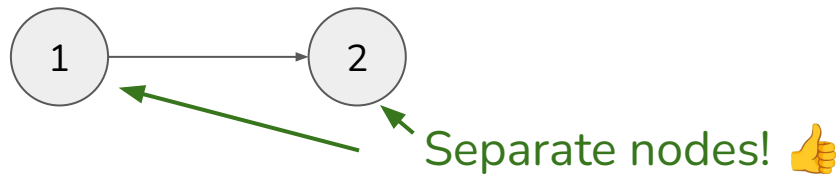
“**Statically** deciding any non-trivial property of a program is impossible”

Things you can't decide just by looking at a program

- Does this program leak memory?
- Does this program have a use-after-free bug?
- Does this function always produce the same output as another function?
- Does this program have a race condition?

What about deciding **dynamically** (at run time)?

Another RefCell example



```
struct List<T> {  
    pub value: T,  
    pub next: Option<Box<List<T>>>,  
}  
  
impl<T> List<T> {  
    fn first(&mut self) -> &mut T { todo!() }  
    fn last(&mut self) -> &mut T { todo!() }  
    fn ends(&mut self) -> (&mut T, &mut T) {  
        (self.first(), self.last())  
    }  
}
```

```
use std::cell::RefCell;  
use std::cell::RefMut;  
struct List<T> {  
    pub value: T,  
    pub next: Option<Rc<RefCell<List<T>>>>,  
}  
impl<T> List<T> {  
    fn first(&self) -> RefMut<T> { todo!() }  
    fn last(&self) -> RefMut<T> { todo!() }  
    fn ends(&self) -> (RefMut<T>, RefMut<T>) {  
        (self.first(), self.last())  
    }  
}
```



&T/&mut T -> checked by compiler

Ref<T>/RefMut<T> checked dynamically by RefCell

RefCell/Rc takeaways

When needing multiple ownership, often use

`Rc<RefCell<T>>`

It's not that the compiler isn't smart enough to validate your program, it's that **it's impossible to validate your program**

Inherited mutability vs. interior mutability

```
struct Person {  
    name: String  
}
```

```
fn clear_name(p: &mut Person) {  
    p.name = name;  
}
```

Inherited mutability: can't mutate the fields unless you have a **&mut** reference

```
fn clear_name(p: &RefCell<Person>) {  
    p.borrow_mut().name = name;  
}
```

Interior mutability: allows mutating even with an immutable reference (safety is checked by some other mechanism)

Trait Objects

New generic syntax:

Exactly the same

```
fn foo<T: Debug>(value: T) { todo!() }
```



```
fn foo(value: impl Debug) { todo!() }
```

Recall: No cost to use traits

```
trait Draw {  
    fn draw(&self) -> String  
}  
  
struct Circle {  
    radius: i32  
}  
  
impl Draw for Circle {  
    fn draw(&self) -> String { todo!() }  
}  
  
struct Rect {  
    size: (i32, i32),  
}  
  
impl Draw for Rect {  
    fn draw(&self) -> String { todo!() }  
}
```

```
fn show(shape: impl Draw) {  
    println!("{}", shape.draw());  
}  
  
pub fn main() {  
    show(Circle { radius: 1 });  
    show(Rect { size: (1, 1) });  
}  
  
crate::show<Circle>:  
    sub     rsp, 152  
    mov     dword ptr [rsp + 12], edi  
    ...  
  
crate::show<Rect>:  
    sub     rsp, 152  
    mov     dword ptr [rsp + 12], edi  
    ...
```

Returning generics

```
fn make_drawable(is_circle: bool) -> impl Draw {  
    if is_circle {  
        Circle { radius: 1 }  
    } else {  
        Rect { size: (1, 1) }  
    }  
}
```

```
error[E0308]: `if` and `else` have incompatible types  
--> draw.rs:36:5  
   |  
33 | /   if is circle {  
34 | |       Circle { radius: 1 }  
   | |       ----- expected because of this  
35 | |   } else {  
36 | |       Rect { size: (1, 1) }  
   | |       ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ expected `Circle`, found `Rect`  
37 | |   }  
   | |___- `if` and `else` have incompatible types
```

Returning generics

```
fn make_drawable() -> impl Draw {  
    if rand::thread_rng().gen() {  
        Circle { radius: 1 }  
    } else {  
        Rect { size: (1, 1) }  
    }  
}
```

Impossible to know whether **Circle** or **Rect** will be returned

```
pub fn main() {  
    let s = make_drawable();  
    println!("{}", std::mem::size_of_val(&s)); ???  
}
```

```
error[E0308]: `if` and `else` have incompatible types  
--> draw.rs:36:5  
   |  
33 | /   if rand::thread_rng().gen() {  
34 | |       Circle { radius: 1 }  
   | |       ----- expected because of this  
35 | |   } else {  
36 | |       Rect { size: (1, 1) }  
   | |       ^^^^^^^^^^^^^^^^^^^^^ expected `Circle`, found `Rect`  
37 | |   }  
   | |_____- `if` and `else` have incompatible types
```

Quick Quiz

```
fn foo<T: Draw>(_v: T) -> T {  
    Circle { radius: 1 }  
}
```

Is this program valid?

Quick Quiz

```
fn foo<T: Draw>(_v: T) -> T {  
    Circle { radius: 1 }  
}
```

Is this program valid?

No! Could be instantiated with **T=Rect** and then returning a **Circle** is improper

Trait Objects

```
fn make_drawable() -> impl Draw {  
    if rand::thread_rng().gen() {  
        Circle { radius: 1 }  
    } else {  
        Rect { size: (1, 1) }  
    }  
}
```

Generics have no run-time cost because we can resolve them at compile-time, but what if we can't?

Trait Objects

```
fn make_drawable() -> Box<dyn Draw> {  
    if rand::thread_rng().gen() {  
        Box::new(Circle { radius: 1 })  
    } else {  
        Box::new(Rect { size: (1, 1) })  
    }  
}
```

What type is in the box?

- don't know, all we know is we can call **draw** on it

```
fn foo() {  
    let s: Box<dyn Draw> = make_drawable();  
    s.draw();  
}
```

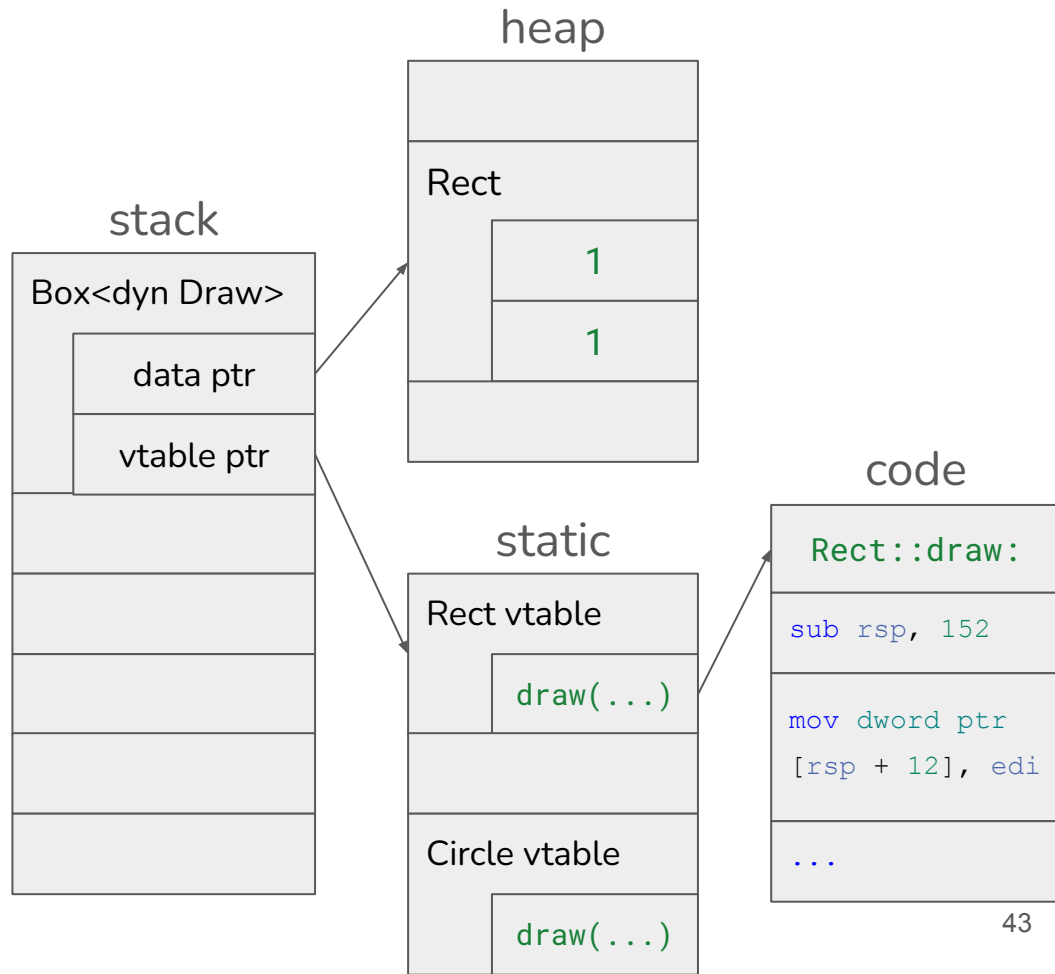
Working with trait objects

Unknown size: always behind a reference of some sort

- `Box<dyn Draw>`
- `&dyn Draw`
- `&mut dyn Draw`
- ...

Trait object layout

```
pub fn main() {  
    let rect = Rect { size: (1, 1) };  
    let trait_obj: Box<dyn Draw> =  
        Box::new(sq);  
}
```



Cost of using trait objects

Normal function call

```
call    b15b6ba806fc18e4
```

Call of static address

- Can be inlined by compiler
- No branch misprediction

Trait object function call

```
load  
mov    rax, qword ptr [rax + 24]  
lea     rdi, [rsp + 16]  
call    rax
```

Call of dynamic address

- Can't be inlined by compiler
- Possible branch misprediction

Another example

```
fn show_all(v: Vec<&dyn Draw>) {  
    for item in v {  
        println! ("{}", item.draw());  
    }  
}  
  
fn main() {  
    show_all(vec![  
        Box::new(Circle { radius: 1 }),  
        Box::new(Rect { size: (1, 1) } )]);  
}
```

Vec that has “different types” in it! (normally not allowed)

Allows implementing patterns from object oriented programming

Today's theme: offloading checks to run-time

Check at compile-time: no run-time performance penalty

- single ownership
- `&` and `&mut` references
- generics with `<T>`

Check at run-time: more flexibility

- multiple ownership with `Rc`
- `Ref` and `RefMut` references from `RefCell`
- generics with `dyn T`