Lecture 8

Parallel Programming

PLQ Review: traits and enums

```rust
trait Shape {
 fn area(&self) -> f32;
}

impl Shape for Circle {
 fn area(&self) -> f32 {
   self.0  * self.0 * 3.14
 }
}
impl Shape for Rect {
 fn area(&self) -> f32 {
   self.0 * self.1
 }
}
```

```rust
enum Shape {
 Circle(Circle),
 Rect(Rect),
}
impl Shape {
 fn area(&self) -> f32 {
   match self {
     &Shape::Circle(Circle(r)) =>
       r * r * 3.14,
     &Shape::Rect(Rect(x, y)) =>
       x * y,
   }
 }
}
```

PLQ Review: traits and enums

```
trait Shape {
 fn area(&self) -> f32;
}
```

Can extend implementers but not functionality

- implement trait for a new type
- can’t add new trait method without breaking existing implementers

```
enum Shape {
 Circle(Circle),
 Rect(Rect),
}
```

Can extend functionality but not implementers

- Add new enum method to add functionality
- Can’t add new variant to enum without breaking existing methods

https://en.wikipedia.org/wiki/Expression_problem

Where are we? Where are we going?

Multi-threading is the motivation behind many of Rust's features. Today, we'll start tying together many features:

- limiting mutability
- smart pointers
- ownership
- traits

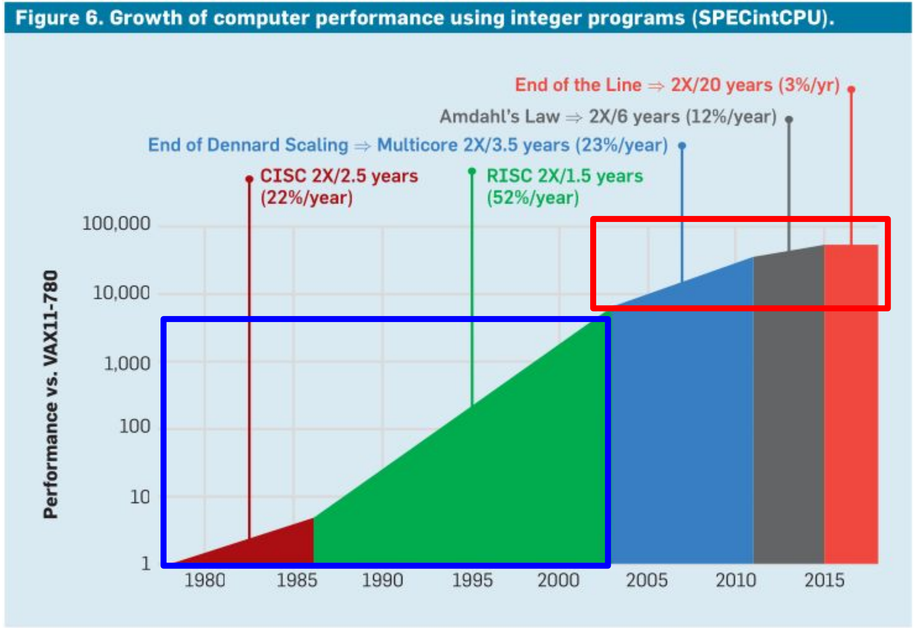
Coming up:

- Two lectures on parallelism
- One lecture on `unsafe` Rust
- One lecture on Rust ecosystem
- One flex lecture

Who cares about parallelism?

Easy to buy more cores, impossible to buy faster cores

- How to use more cores?



Why is parallelism hard?

```c
#define NUM_THREADS 10
#define INCREMENTS 10000

int counter = 0;

void* increment_counter(void* arg) {
    for (int i = 0; i < INCREMENTS; i++) {
        counter = counter + 1;
    }
    return NULL;
}
```

```c
int main() {
    pthread_t threads[NUM_THREADS];
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_create(&threads[i], NULL,
                       increment_counter, NULL);
    }
    // wait for threads to finish...

    printf("Counter: %d\n", counter);
}
```

What is printed?

https://godbolt.org/z/PbPe81aMc

Data race

Multiple accesses with at least one writer

```
for (int i = 0; i < INCREMENTS; i++) {
 int cur_count = counter;
 int new_count = cur_count + 1;
 counter = new_count;
}
```

time ↓

Thread 1:	Thread 2:
`cur_count <- 11`	
	`cur_count <- 11`
`new_count <- 12`	
`counter <- 12`	
	`new_count <- 12`
	`counter <- 12`

Mutex ("mutual exclusion")

```
#define NUM_THREADS 10
#define INCREMENTS 10000

int counter = 0;
pthread_mutex_t counter_mutex;

void* increment_counter(void* arg) {
    for (int i = 0; i < INCREMENTS; i++) {
        pthread_mutex_lock(&counter_mutex);
        counter++;
        pthread_mutex_unlock(&counter_mutex);
    }
    return NULL;
}
```

"critical section"

```
int main() {
    pthread_t threads[NUM_THREADS];
    pthread_mutex_init(&counter_mutex, NULL);
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_create(&threads[i], NULL,
                       increment_counter, NULL);
    }
    // wait for threads to finish...

    printf("Counter: %d\n", counter);
}
```

https://godbolt.org/z/P6PM9eTjE

Critical section

Guarantee that only one thread will be executing critical section at a time

- Other threads wait if mutex is locked

```
for (int i = 0; i < INCREMENTS; i++) {
 lock(mutex);
 int cur_count = counter;
 int new_count = cur_count + 1;
 counter = new_count;
 unlock(mutex)
}
```

time

Thread 1:	Thread 2:
`lock(mutex)`	
`cur_count <- 11`	
`new_count <- 12`	`lock(mutex)`
`counter <- 12`	…
`unlock(mutex)`	…
	mutex acquired

Mutex challenges

What are bugs you can make when using mutexes?

Mutex challenges

- Forget to use one at all
- Forget to lock
- Forget to unlock
- Lock in wrong order
- Lock while already locked
- Unlock while already unlocked

Rust’s bold claims

Impossible to forget to protect data with a mutex

- Compile-time guarantee of no data races!

Impossible to

- Forget to lock
- Forget to unlock
- Unlock while already unlocked

Can still do:

- Lock in wrong order
- Lock while already locked

Initially [safety and concurrency] seemed orthogonal, but to our amazement, the solution turned out to be identical: **the same tools that make Rust safe also help you tackle concurrency head-on**.”

Compiler enforces rules for safe concurrency. **“Thread safety isn't just documentation; it's law.”**

https://blog.rust-lang.org/2015/04/10/Fearless-Concurrency.html

What does parallelism look like in Rust?

```rust
use std::thread;

const NUM_THREADS: usize = 10;
const INCREMENTS_PER_THREAD: usize = 10000;
```

```rust
fn main() {
    let mut counter = 0;
    for _ in 0..NUM_THREADS {
        thread::spawn(|| {
            for _ in 0..INCREMENTS_PER_THREAD {
                counter += 1;
            }
        });
    }
    // wait for threads to finish...

    println!("Counter: {}", counter);
}
```

What does parallelism look like in Rust?

```
use std::thread;

const NUM_THREADS: usize = 10;
const INCREMENTS_PER_THREAD: usize = 10000;
```

```
fn main() {
    let mut counter = 0;
    for _ in 0..NUM_THREADS {
        thread::spawn(|| {
            for _ in 0..INCREMENTS_PER_THREAD {
                counter += 1;
            }
        });
    }
    // wait for threads to finish...

    println!("Counter: {}", counter);
}
```

important part

thread executes a closure

Same data race problem as before!

Which (if any) of Rust's ownership/borrowing rules are violated?

One (of many) problems

```
error[E0499]: cannot borrow `counter` as mutable more than once at a time
 --> race.rs:10:36
   |
10 |           thread::spawn(|| {
   |                         ^^ `counter` mutably borrowed in previous iteration of loop
11 |               for _ in 0..INCREMENTS_PER_THREAD {
12 |                   counter += 1;
   |                   ------- borrows occur due to use of `counter` in closure
13 |               }
14 |           });
```

Need to be able to mutate counter, but can't give out multiple mutable references?

One (of many) problems

```
error[E0373]: closure may outlive `main`, but borrows `counter`, which is owned by `main`
    |
10  |         thread::spawn(|| {
    |                       ^^ may outlive borrowed value `counter`
11  |             for _ in 0..INCREMENTS_PER_THREAD {
12  |                 counter += 1;
    |                 ------- `counter` is borrowed here
    |
 note: function requires argument type to outlive `'static`
```

How long can a thread live?
Who owns counter?

Questions to solve:

Need to be able to mutate counter, but can't give out multiple mutable references?

How long can a thread live? Who owns counter?

One (of many) problems

```
error[E0373]: closure may outlive `main`, but borrows `counter`, which is owned by `main`
    |
10  |         thread::spawn(|| {
    |                       ^^ may outlive borrowed value `counter`
11  |             for _ in 0..INCREMENTS_PER_THREAD {
12  |                 counter += 1;
    |                 ------- `counter` is borrowed here
    |
 note: function requires argument type to outlive `'static`
  --> race.rs:10:22
    |
10  |           let handle = thread::spawn(|| {
    |  ______________________^
11  | |             for _ in 0..INCREMENTS_PER_THREAD {
12  | |                 counter += 1;
13  | |             }
14  | |         });
    | |_________^
help: to force the closure to take ownership of `counter`, use  the `move` keyword
    |
10  |         let handle = thread::spawn(move || {
    |                                    ++++
```

`Rc/RefCell` to the rescue?

```rust
use std::rc::Rc;
use std::cell::RefCell;
use std::thread;

const NUM_THREADS: usize = 10;
const INCREMENTS_PER_THREAD: usize = 10000;
```

Shared ownership and interior mutability

```rust
fn main() {
    let counter = Rc::new(RefCell::new(0));
    for _ in 0..NUM_THREADS {
        let counter = Rc::clone(&counter);
        thread::spawn(move || {
            for _ in 0..INCREMENTS_PER_THREAD {
                *counter.borrow_mut() += 1;
            }
        });
    }
    // wait for threads to finish...
    println!("Counter: {}", counter.borrow_mut());
}
```

Rc/RefCell to the rescue?

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

cons
cons
```

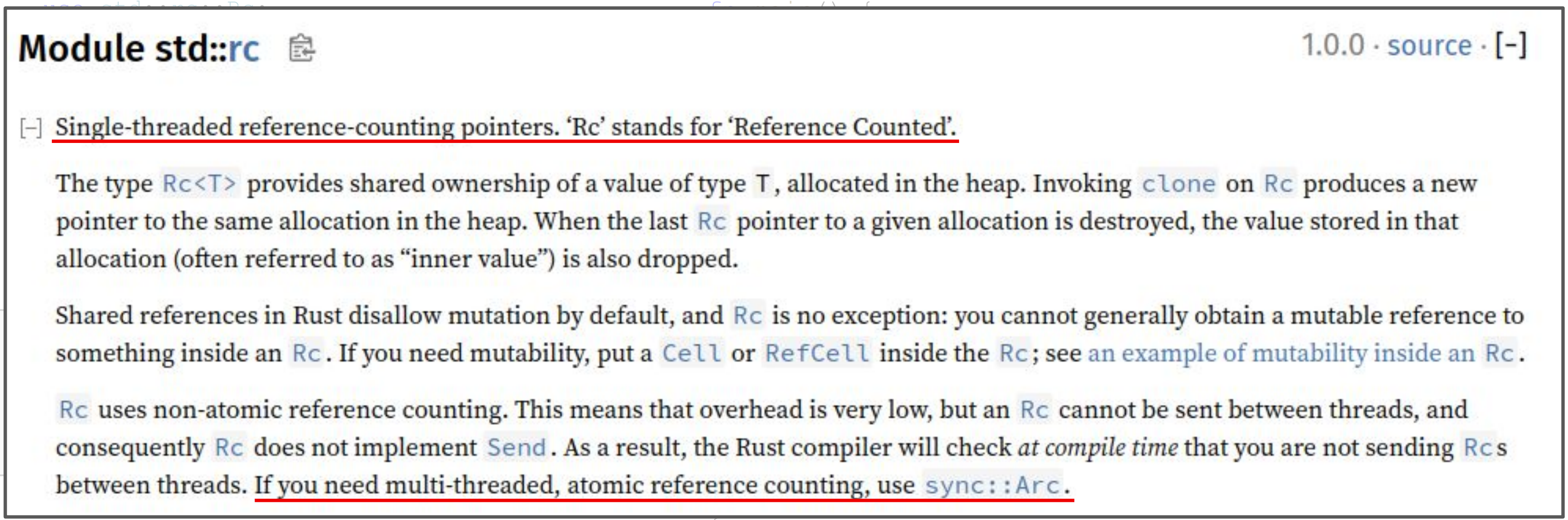
```
fn main() {
```

```
error[E0277]: `Rc<RefCell<i32>>` cannot be sent between threads safely
  --> race4.rs:14:36
   |
14 |         let handle = thread::spawn(move || {
   |                                    ^
   |  __________________________________|
   | |
15 | |           for _ in 0..INCREMENTS_PER_THREAD {
16 | |               *counter.borrow_mut() += 1;
17 | |           }
18 | |       });
   | |_________^ `Rc<RefCell<i32>>` cannot be sent between threads safely
   |
```

Sh

```
}
```

`Rc`/`RefCell` to the rescue?



Module std::rc

1.0.0 · source · [-]

[-] Single-threaded reference-counting pointers. 'Rc' stands for 'Reference Counted'.

The type `Rc<T>` provides shared ownership of a value of type `T`, allocated in the heap. Invoking `clone` on `Rc` produces a new pointer to the same allocation in the heap. When the last `Rc` pointer to a given allocation is destroyed, the value stored in that allocation (often referred to as "inner value") is also dropped.

Shared references in Rust disallow mutation by default, and `Rc` is no exception: you cannot generally obtain a mutable reference to something inside an `Rc`. If you need mutability, put a `Cell` or `RefCell` inside the `Rc`; see an example of mutability inside an `Rc`.

`Rc` uses non-atomic reference counting. This means that overhead is very low, but an `Rc` cannot be sent between threads, and consequently `Rc` does not implement `Send`. As a result, the Rust compiler will check *at compile time* that you are not sending `Rc`s between threads. If you need multi-threaded, atomic reference counting, use `sync::Arc`.

Shared ownership and interior mutability

Single Threaded:

`Rc`: shared ownership with a reference count

`RefCell`: interior mutability by panicking if multiple mutable borrows

Multi-Threaded:

`Arc`: shared ownership with an *atomic* reference count

`Mutex`: interior mutability by making other threads wait

Rust parallelism third attempt

```
use std::sync::{Arc,Mutex};
use std::thread;

const NUM_THREADS: usize = 10;
const INCREMENTS_PER_THREAD: usize = 10000;
```

```
fn main() {
    let counter = Arc::new(Mutex::new(0));
    for _ in 0..NUM_THREADS {
        let counter = Arc::clone(&counter);
        thread::spawn(move || {
            for _ in 0..INCREMENTS_PER_THREAD {
                let mut guard = counter.lock().unwrap();
                *guard += 1;
            }
        });
    }
    // wait for threads to finish...
    println!("Counter: {}", counter.lock().unwrap());
}
```

`Mutex` provides interior mutability!
Exclusive ownership while mutex locked

Zooming in on `thread::spawn`

```
let counter = Arc::new(Mutex::new(0)); 1
for _ in 0..4 {
  let counter: Arc<Mutex<u32>> =          2
     Arc::clone(&counter);

  thread::spawn(move || { 3
    let mut guard = counter.lock().unwrap();
    *guard += 1;
  });
}
```

(1) make shared resource

(2) clone shared resource handle once per thread

(3) move shared resource handle into thread

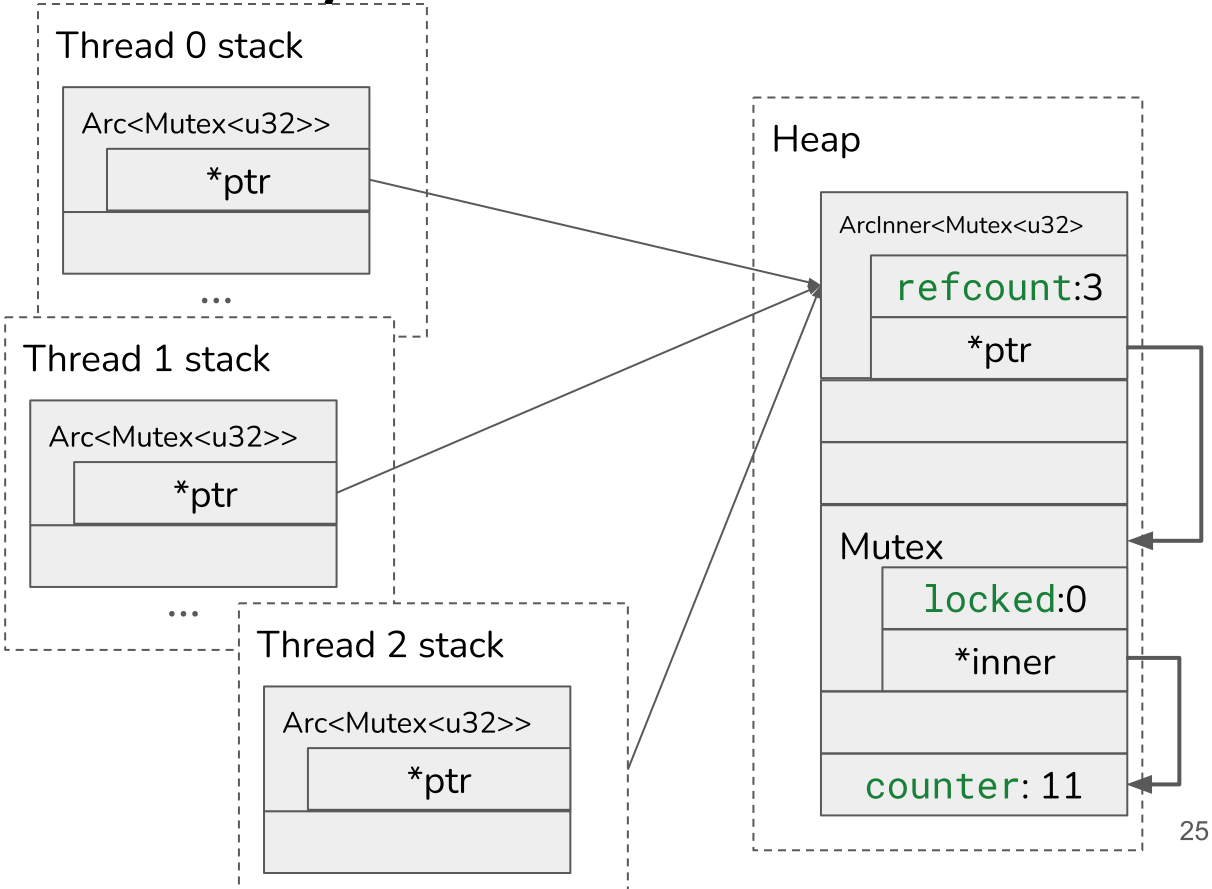
Very common pattern when spawning threads

Shared-state concurrency

Threads have separate stacks but a shared heap

Share ownership of *same* mutex

Control access of shared counter via mutex



Zooming in on Mutexes

```rust
use std::sync::{Mutex, MutexGuard};

fn increment(m: &Mutex<u32>) {
    let guard: MutexGuard<u32> = m.lock().unwrap();
    *guard += 1;
    // guard is dropped -> mutex is unlocked
}

fn main() {
    let counter = Mutex::new(0);
    increment(&counter);
}
```

Quiz

Why is it impossible to forget to unlock a mutex?

Quiz

Why is it impossible to forget to unlock a mutex?

```rust
fn copy_inner(m: &Mutex<u32>) -> u32 {
    let guard: MutexGuard<u32> = m.lock().unwrap();
    let value = *guard;
    return value
    // guard is dropped -> mutex is unlocked
}
```

Impossible to not drop `MutexGuard`

Quiz

Why is it impossible to forget to lock a mutex?

Quiz

Why is it impossible to forget to lock a mutex?

Only `lock` gives access to inner value

```
fn no_lock1(m: &Mutex<u32>) -> u32 {
    let value = m.???();
    return v;
}
```

Can't have reference to inner value outlive `guard`

```
fn no_lock2(m: &Mutex<u32>) -> &mut u32 {
    let mut guard = m.lock().unwrap();
    let v: &mut u32 = &mut *guard;
    return v;
}
```

Quiz

Why is it impossible to unlock a mutex twice?

Quiz

Why is it impossible to unlock a mutex twice?

```
fn double_unlock(m: &Mutex<u32>) -> u32 {
    let guard = m.lock().unwrap();
    let value = *guard;
    drop(guard);
    drop(guard);
    return value;
}
```

Quiz code

https://godbolt.org/z/P179e9a3z

Compared to C

```c
int counter = 0;
pthread_mutex_t counter_mutex;
```

```rust
let counter: Mutex<u32> =
Mutex::new(0);
```

Nothing associates mutex with data!

Mutex has ownership of protected data

Quiz takeaways

Ownership and reference lifetimes make it impossible to misuse a mutex

Initially [safety and concurrency] seemed orthogonal, but to our amazement, the solution turned out to be identical: **the same tools that make Rust safe also help you tackle concurrency head-on**.”

Compiler enforces rules for safe concurrency. **“Thread safety isn't just documentation; it's law.”**

https://blog.rust-lang.org/2015/04/10/Fearless-Concurrency.html

How does the compiler know?

```
error[E0277]: `Rc<RefCell<i32>>` cannot be sent between threads safely
  --> race4.rs:14:36
   |
14 |           let handle = thread::spawn(move || {
   |                                      ^
   |  ____________________________________|
   | |
15 | |             for _ in 0..INCREMENTS_PER_THREAD {
16 | |                 *counter.borrow_mut() += 1;
17 | |             }
18 | |         });
   | |_________^ `Rc<RefCell<i32>>` cannot be sent between threads safely
   |
```

suspicious: is this analysis special-cased for standard library types?

How does the compiler know?

```
error[E0277]: `Rc<RefCell<i32>>` cannot be sent between threads safely
  --> race4.rs:14:36
   |
14 |             let handle = thread::spawn(move || {
   |                                        ^
   |  ______________________________________|
   | |
15 | |               for _ in 0..INCREMENTS_PER_THREAD {
16 | |                   *counter.borrow_mut() += 1;
17 | |               }
18 | |           });
   | |__________^ `Rc<RefCell<i32>>` cannot be sent between threads safely
   |
  = help: within `[closure]`, the trait `Send` is not implemented for `Rc<RefCell<i32>>`
```

Send and Sync

Send: it is safe to send this type to another thread

Sync : it is safe to share this type between threads

- T is Sync if and only if &T is Send

“safe” -> won’t cause memory safety errors or data races

Types that are...

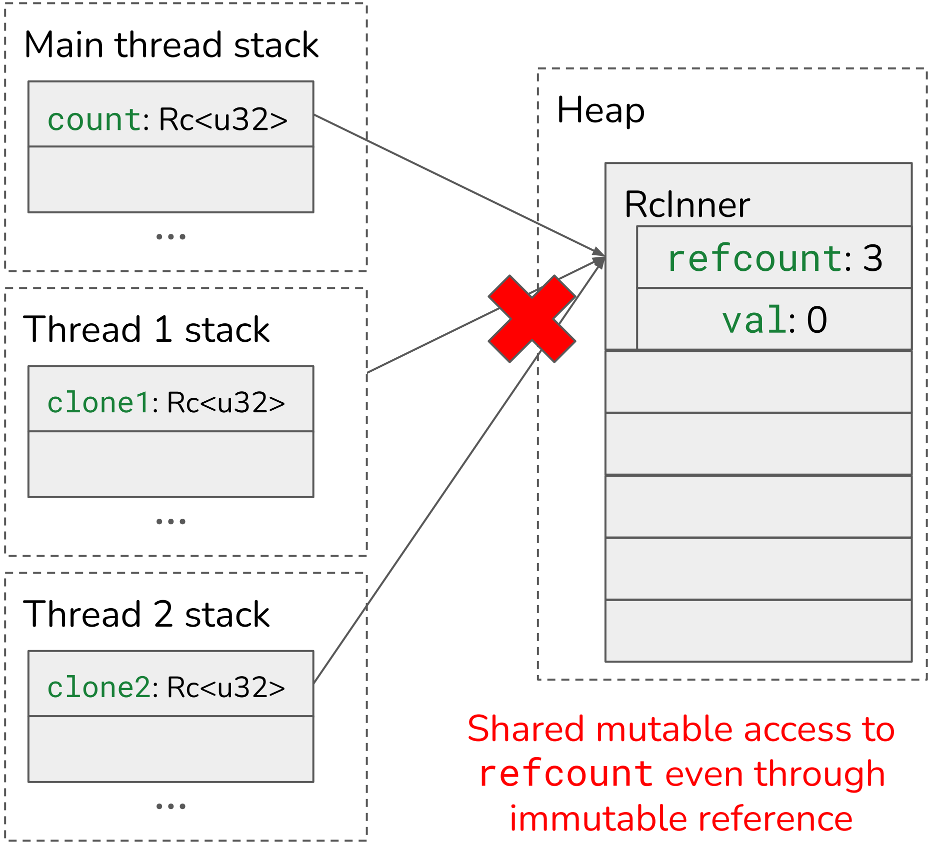
	not `Send`	`Send`
not `Sync`	`Rc`	`RefCell`

`Sync`	incredibly rare	Most structs
		`Mutex`
		`Arc`

Not Sync or Send: Rc

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

fn main() {
    let count = Rc::new(0);
    let clone1 = Rc::clone(&count);
    let clone2 = Rc::clone(&count);
    thread::spawn(move || {
        drop(clone1);
    });
    thread::spawn(move || {
        drop(clone2);
    });
}
```



Shared mutable access to refcount even through immutable reference

Not Sync or Send: Rc

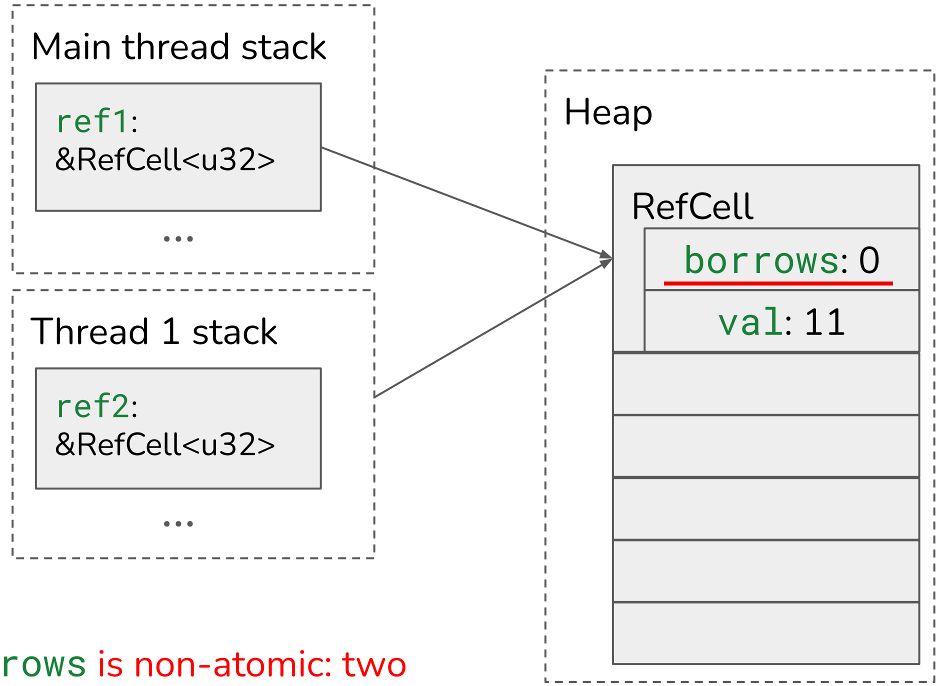
```
error[E0277]: `Rc<i32>` cannot be sent between threads safely
  --> rc.rs:8:19
   |
8  |        thread::spawn(move || {
   |        ------------- ^------
   |        |             |
   |  ______|_____________within this `[closure]`
   | |      |
   | |      required by a bound introduced by this call
9  | |          drop(clone1);
10 | |      });
   | |_____^ `Rc<i32>` cannot be sent between threads safely
   |
   = help: within `[closure]`, the trait `Send` is not implemented for `Rc<i32>`
```

Send but not Sync: RefCell

```rust
use std::{thread, cell::RefCell};

fn increment(r: &RefCell<u32>) {
    let mut count = r.borrow_mut();
    *count += 1;
}

fn main() {
    let count = RefCell::new(11);
    let ref1 = &count;
    let ref2 = &count;
    thread::spawn(move || {
        increment(ref2);
    });
    increment(ref1);
}
```



borrows is non-atomic: two threads could both successfully borrow_mut at the same time

Send but not Sync: `RefCell`

```
error[E0277]: `RefCell<u32>` cannot be shared between threads safely
  --> refcell.rs:13:19
   |
13 |       thread::spawn(move || {
   |  _____-------------_^
   | |      |
   | |      required by a bound introduced by this call
14 | |          increment(ref1);
15 | |      });
   | |_____^ `RefCell<u32>` cannot be shared between threads safely
   |
   = help: the trait `Sync` is not implemented for `RefCell<u32>`
   = note: if you want to do aliasing and mutation between multiple threads, use `std::sync::Mutex` instead
   = note: required for `&RefCell<u32>` to implement `Send`
```

Send and Sync are special

Automatically derived for all types whose members are Send/Sync

You won’t implement Send/Sync for your types, but you may use them as bounds for type parameters in generic functions

Quiz:

If the lock is always automatically released, is it possible to have a deadlock in Rust?

Quiz:

If the lock is always automatically released, is it possible to have a deadlock in Rust?

Yes! Double lock, as shown before

What else?

Quiz:

If the lock is always automatically released, is it possible to have a deadlock in Rust?

Yes!

```
fn swap1(a: Arc<Mutex<u32>>, b: Arc<Mutex<u32>>) {
    let mut guard_a = a.lock().unwrap();
    let mut guard_b = b.lock().unwrap();
    // do the swap
}
```

thread 1 waiting

```
fn swap2(a: Arc<Mutex<u32>>, b: Arc<Mutex<u32>>) {
    let mut guard_b = b.lock().unwrap();
    let mut guard_a = a.lock().unwrap();
    // do the swap
}
```

thread 2 waiting

Quiz:

If the lock is always automatically released, is it possible to have a deadlock in Rust?

```
fn swap1(a: Arc<Mutex<u32>>, b:
Arc<Mutex<u32>>) {
    let mut guard_a = a.lock().unwrap();
    let mut guard_b = b.lock().unwrap();
    // do the swap
}

fn swap2(a: Arc<Mutex<u32>>, b:
Arc<Mutex<u32>>) {
    let mut guard_b = b.lock().unwrap();
    let mut guard_a = a.lock().unwrap();
    // do the swap
}
```

Yes!

```
fn main() {
    let a = Arc::new(Mutex::new(10));
    let b = Arc::new(Mutex::new(20));

    let a_cloned = Arc::clone(&a);
    let b_cloned = Arc::clone(&b);
    thread::spawn(move || {
        swap1(a_cloned, b_cloned);
    });
    swap2(Arc::clone(&a), Arc::clone(&b));
}
```

One last form of interior mutability

```
use std::sync::atomic::{AtomicUsize, Ordering};

fn increment_atomic(counter: &AtomicUsize) {
   counter.fetch_add(1, Ordering::SeqCst);
}
```

Atomics: allow mutation through a shared reference.

Other threads are guaranteed not to observe intermediate values.

What's this?

- it's complicated: just use `Ordering:SeqCst`