# Rust Lesson 5

Concurrency

# Spawning a Thread

```
let thread = thread::spawn(|| {
    println!("tada");
});
thread.join();
```

# Spawning a Thread

```
pub fn spawn<F, T>(f: F) -> JoinHandle<T>
where
    F: FnOnce() -> T + Send + 'static,
    T: Send + 'static {}
```

# Send and Sync

- Send:
  - Implemented by types that can cross thread boundaries.
- Sync
  - Implemented by types that can Send references between threads.
  - T is Sync if and only if &T is Send

# Quick Recap of Trait Objects

- Box<dyn Display>
  - A boxed variable that implements Display.
  - The actual type of the variable is determined at runtime via a table lookup.

# Quick Recap of Trait Objects

- **let** objects = Vec<Box<**dyn** Display>>
  - A vector of objects that all implement display.
  - objects.push(Box::new(“this implements display”))
  - objects.push(Box::new(6));
  - The objects variable can contain multiple different types now.

# Will it Compile?

```
fn main() {
    // the compiler is only aware that this variable implements Send and Display at compile time, even though we know a Box<&str> implements much more than that.
    let send: Box<dyn Send+Display> =  Box::new("this is send but not sync");


    // move keyword forces closure to take ownership of anything used in it.
    let thread = thread::spawn(move || {
        println!("{send}");
    });
    thread.join().unwrap();
}
```

# Will it Compile

```
fn main() {
    let sync: Box<dyn Sync+Display> = Box::new("this is sync but not send");

    let thread = thread::spawn(move || {
        println!("{sync}");
    });
    thread.join().unwrap();
}
```

# Will it Compile?

```
fn main() {
    let sync: &(dyn Sync+Display) = &"this is sync but not send";
    let thread = thread::spawn(move || {
        println!("{sync}");
    });
    thread.join().unwrap();
}
```

# Will it Compile?

```
fn main() {
    let some_string = String::new();
    let string_ref: &(dyn Sync + Display) = &some_string;
    let thread = thread::spawn(move || {
        println!("{string_ref}");
    });
    thread.join().unwrap();
}
```

# Easier Paradigms

- Having to either move ownership into the thread or have all references be static is inconvenient

# Solution option one: using tricks to get static references

```
fn main() {
    let some_string = String::new();
    let string_ref: &'static (dyn Sync + Display) = unsafe {
        let non_static_ptr = (&some_string) as *const String;
        &*(non_static_ptr)
    };
    let thread = thread::spawn(move || {
        println!("{string_ref}");
    });
    thread.join().unwrap();
}
```

# Solution option one: using tricks to get static references (safely)

```
fn main() {
    let some_string = String::new();
    let string_ref: &'static (dyn Sync + Display) =  {
        let boxed = Box::new(some_string);
        Box::leak(boxed)
    };
    let thread = thread::spawn(move || {
        println!("{string_ref}");
    });
    thread.join().unwrap();
}
```

# Solution option 2: Just not bothering with shared memory and references

- Consider cloning before moving into a thread, then passing that cloned value out when the thread exits.

# Solution option 2: Just not bothering with shared memory and references

- See cloning() function

# A more flexible alternative

- See thread_channel() function

# Gotchas of Thread Channels

- Thread channels have overhead.
- Back Pressure
- Message passing doesn’t work for everything.
- MPMC is complicated.

# Overcoming thread channel limitations.

- Bounded channels. (crossbeam or async)
- Crossbeam ArrayQueues
  - https://github.com/utahrobotics/lunadev-2025/blob/fusion/lunabotics/lunabot/src/apps/production/rp2040.rs

# Solution option 3: Shared Ownership.

- Pros:
  - Can feel like magic if done right.
  - Great when you don’t need mutability.
- Cons
  - Not needing mutability is rare.
  - Added complexity.
  - More overhead.
  - Potential of cyclical references.
- Take a look at the reference_counting() function.

# Gotchas of Using Arc

- Immutable
- Overhead
- For an Arc<T> to be passed between threads, T must be Send and Sync.
- Added Complexity

# Overcoming Rc/Arc limitations

- Mutex for interior mutability.
  - This does make things more complicated though.
- Is overhead actually a problem?
- Cyclical references.
  - Use Weak<T>

# Mutexes

- allows you to mutate some data through a shared "immutable" reference
- lock()
  - Blocks until a MutexGuard is acquireable
- try_lock()
  - Non blocking version

# Mutexes

- Wrapping data in a Mutex is a way to turn some data that is Send but not Sync Syncable
  - if T is Send, then Mutex<T> is Send and Sync

# Poisoning

- Happens when a thread panics when a MutexGuard is currently in scope.
- Makes all subsequent calls to lock() return Err

# RwLock

- Like mutex, but allows for multiple concurrent readers
- Call read() to get read lock
- Call write() to get write lock

# Will it Compile?

```
//immutable variable
let reference_counted = Arc::new(Mutex::new(String::new()));
// notice how string_ref is immutable
let string_ref: Arc<Mutex<String>> = reference_counted.clone();
let thread = thread::spawn(move || {
    // mutating through an immutable reference?
    (string_ref.lock().unwrap()).push_str("fromthread1");
});

let string_ref: Arc<Mutex<String>> = reference_counted.clone();
let thread2 = thread::spawn(move || {
    (string_ref.lock().unwrap()).push_str("fromthread1");
});
```

# Will it Compile?

```
let rc = Arc::new(Mutex::new(String::new()));


let string_ref: Arc<Mutex<String>> = rc.clone();
let thread = thread::spawn(move || {
    drop(*(string_ref.lock().unwrap()));
});
thread.join().unwrap();
```

# Cons of Mutexes:

- Deadlocks
- Added complexity
- Be careful of causing too much blocking.
- Overhead

# Atomics: Interior Mutability for primitive types.

- Atomicity via hardware x86-64 instructions.

# Atomics Example

- https://gist.github.com/matthewashton-k/58a16115f8f47965aa7511afe904feb9

# Atomic spinlock

- See custom_lock()

# Conclusion

- When to use memory leaks?
- When to use Reference counting + interior mutability?
- When to use thread channels?