How Rust Achieves Thread Safety

This is why types are so damn coo

Sorry, I'm sick of AR

(This is way cooler anyway)

What is a trait?

What is a marker trait?

Spawning a thread

```
pub fn spawn<F>(f: F) -> JoinHandle
where F: FnOnce() + Send + 'static
```

'static

- Value must outlive the scope it was created in
- For closures, this means they have to own all data that they access
- No shared references

Send

- A value that can safely be sent from one thread to another
- Any data which is made only of `Send` values is automatically `Send`
- Types which can be used for sharing data are not `Send` (like pointers).

Sync

- A value that can safely be shared between threads.
- Not explicitly referenced by the Thread signature.

```
unsafe impl<T> Send for &T where T: Sync
```

What makes a `Fn()` be `Send`?

If you were to make a closure be a struct, it'd look something like this:

```
struct Closure {
  args: Args,
  env: Environment,
}

struct Args {
  # field for each argument type
}

struct Environment {
  # field for each value closed over
}
```

How do you share data in Rust anyway?

- Cannot share data by default in Rust, only borrow it.
- Borrows work well for straightforward lifetimes, does not work for complex
  ones

How do you share data in Rust anyway?

Enter `Rc` (short for "reference counted")

```
pub struct Rc<T: ?Sized> {
    _ptr: NonZero<*mut RcBox<T>>,
}
```

How do you share data in Rust anyway?

- `*mut` is a pointer
- Pointers are not `Send`
- `Rc` is not `Send`

Ok no seriously how do you share data across threads, though?

Enter `Arc` (short for atomic reference counted)

- Similar structure to `Rc`
- Implementation is atomic
- Still uses a pointer, so must be explicitly declared as `Send`

```
unsafe impl<T> for Arc<T> where T: Send + Sync
```

How does that prevent data races?

- `Arc` can only give you a shared (immutable) reference to it's data.
- The `Arc` owns the value, no way to reclaim ownership.
- Need a type that is `Sync` and also exposes a mutable reference to it's data

Enter `Mutex`

- This is the *only* type that gives a mutable reference, and is `Sync`
- The only way to share data across threads (in safe Rust anyway)
- The primitives `Mutex` uses are still accessible but your type would not be
  `Sync` or `Send` by default

```rust
// This is our marker trait
trait Foo { }

// This specifies it is implemented by default
impl Foo for .. { }

// Only take "foo safe" functions here
fn do_foo_safe<F>(f: F) where F: FnOnce() + Foo
```

```
// This compiles because `i32` is "foo safe"
fn main() {
  let x = 1;
  do_foo_safe({|| x });
}

// We can opt out of i32 being "foo safe"
impl !Foo for i32 { }

// This now fails to compile
fn main() {
  let x = 1;
  do_foo_safe({|| x });
}
```

These are already best practices

The compiler isn't forcing anything new

It just won't let you write a program that is incorrect