Smart Pointers

Cells

interlinked

The Equation

`Arc<RwLock<T>>` is `Rc<RefCell<T>>`,

`RwLock<T>` is `RefCell<T>`

`Cell<T>`

Interior mutability by moving values in and out of the cell

Should be used for types what are not resource intensive - cheap copying and moving

```rust
fn normal_mut(num: &mut i32) {
    *num += 10;
}

fn cell_mut(num: &Cell<i32>) {
    let inner = num.get();
    num.set(inner + 12);
}
```

It is also possible with non-copy types

```
fn non_copy_mut(x: &Cell<String>) {
    let inner = x.take();
    x.set(inner + " bonk")
}
```

A pain to work with them though

```
println!("non copy mut {:?}", z);
```

`Cell` is quite chill - it doesn't panic

`RefCell<T>`

Dynamic borrowing

Not that chill, runtime panic when violating the borrow checker

```rust
fn ref_cell_mut(x: &RefCell<String>) {
    let mut reference_x = x.borrow_mut();
    *reference_x = "word".to_string();
}

fn ref_cell_mut_no_scam(x: &RefCell<String>) {
    let ref_x_1 = x.borrow_mut();
    let ref_x_2 = x.borrow_mut();
    println!("wow this surely does not panic {ref_x_1:?} {ref_x_2:?}");
}
```

`OnceCell<T>`

It is there! It is just a mix of `RefCell` (we get a shared reference with no copy) and `Cell` (no runtime borrow check) and it generally should be only set once. Good for lazily computing/initiaing statics.

Async

`RefCell` and `Cell` are `!Sync`

But we have `Mutex` and `RwLock`!

```rust
pub enum Cow<'a, B>
where
    B: 'a + ToOwned + ?Sized,
{
    Borrowed(&'a B),
    Owned(<B as ToOwned>::Owned),
}
```

Why?

Just like `Option<T>` is *maybe* `T`, `Cow` is *maybe* *owned* which is *maybe* useful