Interior Mutability | *Cell & RefCell*

Interior Mutability

- Mutable data is sometimes problematic
  - Compiler errors, ownership issues, etc.
- Possible to create permanently mutable memory
  - Less restrictive than compiler
    - Trade-offs in implementation & performance

Cell

- Permanently mutable memory location
  - Can always be mutated, even if the containing structure is immutable
- Accessing ***Cell*** data always results in a <u>move</u> or <u>copy</u>
- Data should be copy-able
  - ***#[derive(Clone, Copy)]***
  - Inefficient for large data types
    - Limit to numbers and booleans
- Prefer ***mut***

Example

```rust
use std::cell::Cell;

#[derive(Debug)]
struct Book {
    signed: Cell<bool>,
}


let my_book = Book {
    signed: Cell::new(false),
};

println!("signed: {}", my_book.signed());
my_book.sign();
println!("signed: {}", my_book.signed());
```

```rust
impl Book {
    fn sign(&self) {
        self.signed.set(true);
    }
    fn signed(&self) -> bool {
        self.signed.get()
    }
}
```

```
signed: false
signed: true
```

RefCell

- Permanently mutable memory location
  - Can always be mutated, even if the containing structure is immutable
- Accessing ***RefCell*** data always results in a <u>borrow</u>
  - Efficient data access (compared to ***Cell***)
  - Borrow checked at runtime
    - Will panic at runtime if rules are broken
    - Only one mutable borrow at a time
- Prefer ***&mut***
- Not thread-safe

Example – Borrow

```rust
use std::cell::RefCell;

struct Person {
    name: RefCell<String>,
}
```

```rust
let name = "Amy".to_owned();
let person = Person {
    name: RefCell::new(name),
};
```

```rust
let name = person.name.borrow();
```

Example - Mutation

```rust
use std::cell::RefCell;

struct Person {
    name: RefCell<String>,
}
```

```rust
let name = "Amy".to_owned();
let person = Person {
    name: RefCell::new(name),
};
```

```rust
let mut name = person.name.borrow_mut();
*name = "Tim".to_owned();
```

```rust
person.name.replace("Tim".to_owned());
```

Example - Mutation

```rust
use std::cell::RefCell;

struct Person {
    name: RefCell<String>,
}
```

```rust
let name = "Amy".to_owned();
let person = Person {
    name: RefCell::new(name),
};
```

```rust
{
    let mut name = person.name.borrow_mut();
    *name = "Tim".to_owned();
}

{
    person.name.replace("Tim".to_owned());
}
```

Example - Checked Borrow

```rust
use std::cell::RefCell;

struct Person {
    name: RefCell<String>,
}
```

```rust
let name = "Amy".to_owned();
let person = Person {
    name: RefCell::new(name),
};
```

```rust
let name: Result<_, _> = person.name.try_borrow();
let name: Result<_, _> = person.name.try_borrow_mut();
```

Recap

- ***Cell*** & ***RefCell*** allow permanent mutation
  - ***Cell*** returns owned data
  - ***RefCell*** returns borrowed data
- ***RefCell*** borrowing can panic at runtime
  - ***try_borrow*** and ***try_borrow_mut*** are non-panicking versions
- Prefer to use ***mut*** and ***&mut***
  - Use ***Cell*** & ***RefCell*** only when it’s not possible to express intentions otherwise
- Not thread-safe