Rust 2023

clase 7

Temario

- Smart pointers
- Tests
- Linters
- Pending issues
- Programación concurrente
- Macros
- Características avanzadas del lenguaje

Smart Pointers

Smart Pointers

- Como vimos `&` y `&mut` son referencias(punteros) a un dato donde se quiere tomar prestado y no tienen ninguna característica más que lo que vimos y no tienen costo.

- Los Smart Pointers en cambio son estructuras de datos que actúan como referencia pero también contienen metadatos y características especiales.

- Hay una diferencia, con el concepto de ownership y borrowing, entre las referencias y los smart pointers: mientras que las referencias solo toman prestados datos, en muchos casos, los smart pointers poseen los datos a los que apuntan.

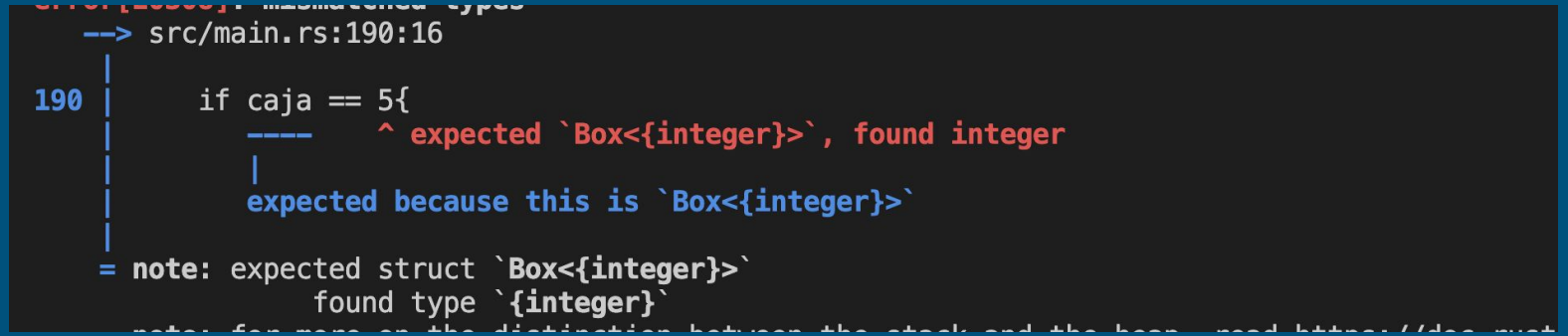
Smart Pointers

- Implementan los traits Drop y Deref.

- Hasta el momento implícitamente vimos 2 smart pointers: String y Vec, cuentan como tales ya que poseen algo de memoria y permiten manipularlos. También tienen metadatos y características especiales.

Smart Pointers: Box<T>

```
fn main() {
    let caja = Box::new(5);
    if caja == 5{
        println!("es cinco!");
    }
}
```



Smart Pointers: Box<T>

```
fn main() {
    let caja = Box::new(5);
    if *caja == 5{
        println!("es cinco!");
    }
}
```

Smart Pointers: Box<T>

```
enum MiLista{
    Nodo(i32, MiLista),
    Nada
}
fn main() {
//nodo1(1) -> nodo2(2) -> nodo3(3) -> Nada
    use MiLista::*;
    let n4 = Nada;
    let n3 = Nodo(3, n4);
    let n2 = Nodo(2, n3);
    let n1 = Nodo(1, n2);
}
```

```
error[E0072]: recursive type `MiLista` has infinite size
   --> src/main.rs:185:1
    |
185 | enum MiLista{
    | ^^^^^^^^^^^^
186 |     Node(i32, MiLista),
    |               ------- recursive without indirection
```

Smart Pointers: Box<T>

```
enum MiLista{
    Nodo(i32, Box<MiLista>),
    Nada
}
fn main() {
//nodo1(1) -> nodo2(2) -> nodo3(3) -> Nada
    use MiLista::*;
    let n4 = Nada;
    let n3 = Nodo(3,Box::new(n4));
    let n2 = Nodo(2, Box::new(n3));
    let n1 = Nodo(1, Box::new(n2));
}
```

Smart Pointers: Deref

```
struct Caja<T>{
    d:T
}
impl<T> Caja<T> {
    fn new(d:T) -> Caja<T>{
        Caja{d}
    }
}
fn main() {
    let caja = Caja::new(5);
    if caja == 5 {
        println!("es cinco!");
    }
}
```

```
error[E0369]: binary operation `==` cannot be applied to type `Caja<{integer}>`
   --> src/main.rs:195:13
    |
195 |     if caja == 5 {
    |        ---- ^^ - {integer}
    |        |
    |        Caja<{integer}>
    |
```

Smart Pointers: Deref

```
fn main() {
    let caja = Caja::new(5);
    if *caja == 5 {
        println!("es cinco!");
    }
}
```

```
error[E0614]: type `Caja<{integer}>` cannot be dereferenced
   --> src/main.rs:195:8
    |
195 |     if *caja == 5 {
    |        ^^^^^
```

Smart Pointers: Deref

```
struct Caja<T>{
    d:T
}
impl<T> Deref for Caja<T>{
    type Target = T;
    fn deref(&self) -> &Self::Target {
        &self.d
    }
}
fn main() {
    let caja = Caja::new(5);
    if *caja == 5 {
        println!("es cinco!");
    }
}
```

✅

Smart Pointers: Drop

```
impl<T> Drop for Caja<T> {
    fn drop(&mut self) {
        println!("Adios!!!");
    }
}
fn main() {
    let caja = Caja::new(5);
    if *caja == 5 {
        println!("es cinco!");
    }
    {
        let caja = Caja::new(5);
    }
    println!("Terminando el main");
}
```

Smart Pointers: Drop

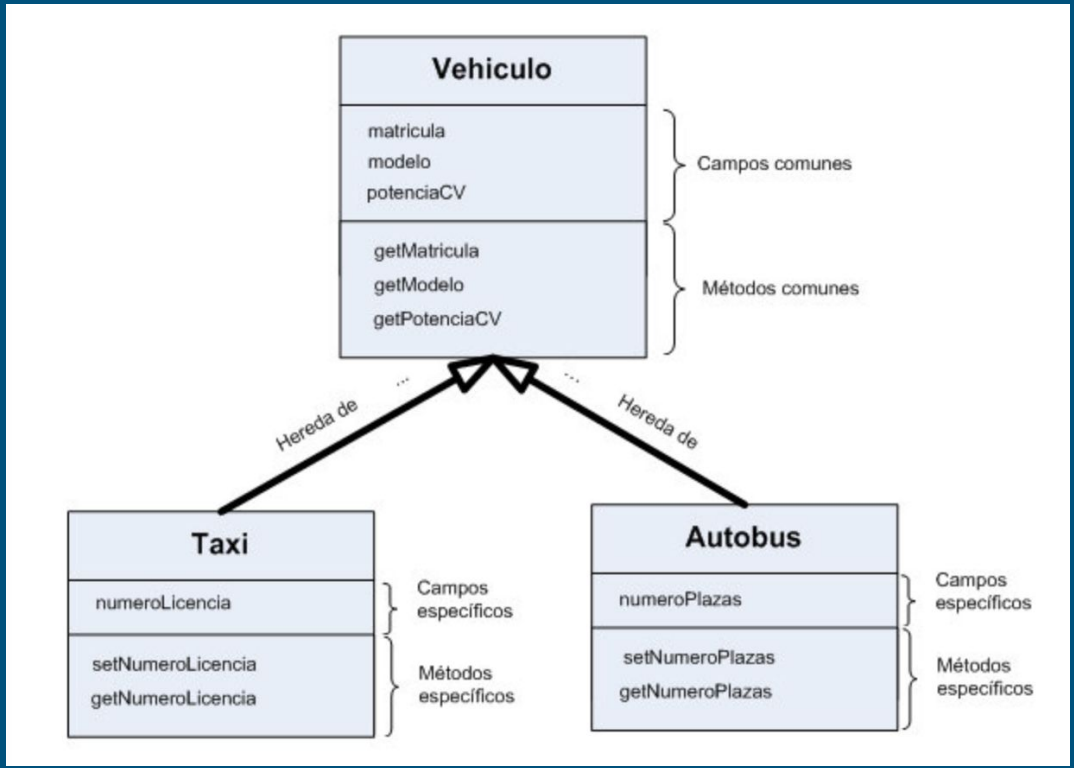
```
fn main() {
    let mut caja = Caja::new(5);
    if *caja == 5 {
        println!("es cinco!");
        caja.drop();
    }
    println!("Terminando el main");
}
```

```
error[E0040]: explicit use of destructor method
   --> src/main.rs:210:14
    |
210 |         caja.drop();
    |         -----^^^^--
    |         |    |
    |         |    explicit destructor calls not allowed
    |         help: consider using `drop` function: `drop(caja)`
```

Smart Pointers: Drop

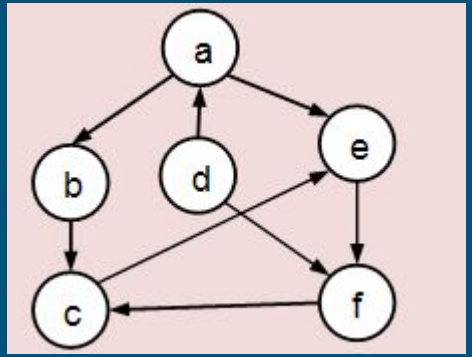
```
fn main() {
    let mut caja = Caja::new(5);
    if *caja == 5 {
        println!("es cinco!");
        drop(caja);
    }
    println!("Terminando el main");
}
```

Smart Pointers: Extra Deref



Smart Pointers: Rc -> Reference Counted

multiple ownership explícito: hay casos en los que un único valor puede tener varios propietarios, como por ejemplo en la estructura de grafo.



O en situaciones de subprocesos, como en programación concurrente.

Smart Pointers: Rc -> Reference Counted

```
enum MiLista{
    Nodo(i32, Box<MiLista>),
    Nada
}
fn main() {
    //nodo1(1) ->
    //            nodo2(2) -> nodo3(3) -> Nada
    //nodo5(5) ->
    use MiLista::*;
    let n4 = Nada;
    let n3 = Nodo(3,Box::new(n4));
    let n2 = Nodo(2, Box::new(n3));
    let n1 = Nodo(1, Box::new(n2));
    let n5 = Nodo(5, Box::new(n2));
}
```

```
error[E0382]: use of moved value: `n2`
  --> src/main.rs:17:31
   |
15 |     let n2 = Nodo(2, Box::new(n3));
   |         -- move occurs because `n2` has type `MiLista`, which does not implement the `Copy` trait
16 |     let n1 = Nodo(1, Box::new(n2));
   |                           -- value moved here
17 |     let n5 = Nodo(5, Box::new(n2));
   |                           ^^ value used here after move
```

Smart Pointers: Rc -> Reference Counted

```
use std::rc::Rc;
enum MiLista{
    Nodo(i32, Rc<MiLista>),
    Nada
}
fn main() {
    //nodo1(1) ->
    //            nodo2(2) -> nodo3(3) -> Nada
    //nodo5(5) ->
    use MiLista::*;
    let n4 = Nada;
    let n3 = Nodo(3,Rc::new(n4));
    let n2 = Nodo(2, Rc::new(n3));
    let  n2_rc = Rc::new(n2);
    let n1 = Nodo(1, Rc::clone(&n2_rc));
    let n5 = Nodo(5, Rc::clone(&n2_rc));
}
```

Smart Pointers: Rc -> Reference Counted

```
fn main() {
    //nodo1(1) ->
    //            nodo2(2) -> nodo3(3) -> Nada
    //nodo5(5) ->
    use MiLista::*;
    let n4 = Nada;
    let n3 = Nodo(3,Rc::new(n4));
    let n2 = Nodo(2, Rc::new(n3));
    let  n2_rc = Rc::new(n2);
    let n1 = Nodo(1, Rc::clone(&n2_rc));
    let n5 = Nodo(5, Rc::clone(&n2_rc));
    println!("cantidad de refs: {}", Rc::strong_count(&n2_rc));
}
```

Smart Pointers: Rc -> Reference Counted(contando refes.)

```
fn main() {
    use MiLista::*;
    let n4 = Nada;
    let n3 = Nodo(3,Rc::new(n4));
    let n2 = Nodo(2, Rc::new(n3));
    let  n2_rc = Rc::new(n2);
    {
        let n1 = Nodo(1, Rc::clone(&n2_rc));
        let n5 = Nodo(5, Rc::clone(&n2_rc));
        println!("cantidad de refs: {}", Rc::strong_count(&n2_rc));
    }
    println!("cantidad de refs: {}", Rc::strong_count(&n2_rc));
}
```