Smart Pointers(cont.)

Smart Pointers: Rc -> Reference Counted

```
use std::rc::Rc;
enum MiLista{
    Nodo(i32, Rc<MiLista>),
    Nada
}
impl MiLista{
    fn get_siguiente(&self) -> Option<&Rc<MiLista>>{
        match self {
            MiLista::Nodo(v, s) => {Some(s)},
            MiLista::Nada =>{None},
        }
    }
    fn get_dato(&self) -> Option<&i32>{
        match self {
            MiLista::Nodo(v, s) => {Some(&v)},
            MiLista::Nada =>{None},
        }
    }
```

Smart Pointers: Rc -> Reference Counted

```
fn psudo_dfs(a:&Rc<MiLista>){
    if !a.get_dato().is_none(){
        if a.get_dato().unwrap() == &1{
            //hacer logica con 1
        }
        println!("{}", a.get_dato().unwrap());
    }
    if let Some(sig) = a.get_siguiente(){
        psudo_dfs(sig);
    }
}
```

Smart Pointers: Rc -> Reference Counted

```
fn main() {
    let mut n4 = Nada;
    let n4_rc = Rc::new(n4);
    let n3 = Nodo(3,n4_rc.clone());
    let n3_rc = Rc::new(n3);
    let n2 = Nodo(2, n3_rc.clone());
    let n2_rc = Rc::new(n2);
    let n1 = Nodo(1, n2_rc.clone());
    let n5 = Nodo(5, n2_rc.clone());
    let n5_rc = Rc::new(n5);
    let n1_rc = Rc::new(n1);
...
```

Smart Pointers: Rc -> Reference Counted

```
    let v = [ n1_rc.clone(), n2_rc.clone(),
        n3_rc.clone(),n4_rc.clone(),n5_rc.clone()
    ];
    for i in v{
        if let Some(dato)=i.get_dato(){
            println!("dfs para: {}", dato);
            psudo_dfs(&i);
        }
        println!("");
    }
}
```

Smart Pointers: RefCell -> patrón de mutabilidad interior

- Permite mutar datos incluso cuando hay referencias inmutables a esos datos. Normalmente esta acción no está permitida por la regla de borrowing.
- Para mutar los datos el patrón utiliza código no seguro dentro de una estructura de datos para modificar las reglas de Rust.
- El código inseguro le indica al compilador que haremos la verificación de forma manual.
- Se pueden usar tipos que usan el patrón de mutabilidad interior solo cuando podemos asegurar de que se seguirá las reglas de préstamo en tiempo de ejecución, aunque el compilador no puede garantizarlo.
- El código no seguro involucrado se envuelve en una API segura y el tipo externo sigue siendo inmutable.

Smart Pointers: RefCell -> mutabilidad interior

```
use std::rc::Rc;
use std::cell::RefCell;
enum MiLista{
    Nodo(RefCell<i32>, Rc<MiLista>),
    Nada
}
```

Smart Pointers: RefCell -> mutabilidad interior

```
impl MiLista{
    fn get_siguiente(&self) -> Option<&Rc<MiLista>>{
        match self {
            MiLista::Nodo(v, s) => {Some(s)},
            MiLista::Nada =>{None},
        }
    }
    fn get_dato(&self) -> Option<&RefCell<i32>>{
        match self {
            MiLista::Nodo(v, s) => {Some(&v)},
            MiLista::Nada =>{None},
        }
    }
}
```

Smart Pointers: RefCell -> mutabilidad interior

```
use MiLista::*;
fn main() {
    let mut n4 = Nada;
    let n4_rc = Rc::new(n4);
    let n3 = Nodo(RefCell::new(3),n4_rc.clone());
    let n3_rc = Rc::new(n3);
    let n2 = Nodo(RefCell::new(2), n3_rc.clone());
    let n2_rc = Rc::new(n2);
    let n1 = Nodo(RefCell::new(1), n2_rc.clone());
    let n5 = Nodo(RefCell::new(5), n2_rc.clone());
    let n5_rc = Rc::new(n5);
    let n1_rc = Rc::new(n1);
    let v = [ n1_rc.clone(), n2_rc.clone(),
        n3_rc.clone(),n4_rc.clone(),n5_rc.clone()
    ];
...
```

Smart Pointers: RefCell -> mutabilidad interior

```
    for i in v{
        if !i.get_dato().is_none(){
            let d = i.get_dato().unwrap();
            println!("dfs para: {}", *d.borrow());
            psudo_dfs(&i);
        }
        println!("");
    }
    let mut n = n1_rc.get_dato().unwrap();
    println!("{:?}", n.borrow());
}
```

Smart Pointers: RefCell -> mutabilidad interior

```
fn psudo_dfs(a:&Rc<MiLista>){
    if !a.get_dato().is_none(){
        let mut d = a.get_dato().unwrap().borrow_mut();
        if *d == 1{
            *d=10;
        }
        println!("{}", *d);
    }
    if let Some(sig) = a.get_siguiente(){
        psudo_dfs(sig);
    }
}
```

Smart Pointers: Repaso

- Rc<T> permite múltiples propietarios de los mismos datos.
- Box<T> y RefCell<T> tienen propietarios únicos.
- Box<T> permite préstamos inmutables o mutables verificados en tiempo de compilación.
- Rc<T> permite sólo préstamos inmutables verificados en tiempo de compilación.
- RefCell<T> permite préstamos inmutables o mutables verificados en tiempo de ejecución.
- Dado que RefCell<T> permite la verificación de préstamos mutables en tiempo de ejecución, puede mutar el valor dentro de RefCell<T> incluso cuando RefCell<T> es inmutable.

Tests

Unit testing

En desarrollo de software es la práctica en la cual se crean pruebas automatizadas para verificar el correcto funcionamiento individual de las unidades de código más pequeñas, como funciones, métodos o clases. Estas pruebas se enfocan en aislar y probar una unidad de código de forma independiente, sin depender de otras partes del sistema.

Unit testing: algunas ventajas

- **Detección temprana de errores**: permiten identificar y corregir errores en una etapa temprana del desarrollo, lo que ayuda a evitar que se propaguen y se conviertan en problemas más difíciles y costosos de solucionar en etapas posteriores.

- **Mejora de la calidad del código:** Al escribir pruebas unitarias, los desarrolladores deben pensar en cómo utilizar y probar sus propias funciones y clases. Esto promueve la escritura de código más limpio, modular y de alta calidad, lo que facilita su mantenimiento y extensión.

Unit testing: ventajas

- ★ **Facilita la refactorización:** Las pruebas unitarias proporcionan un nivel de seguridad al refactorizar el código. Si las pruebas pasan correctamente después de realizar cambios, se tiene la confianza de que las funcionalidades previamente probadas siguen intactas.

- ★ **Documentación viva:** Las pruebas unitarias actúan como una forma de documentación viva del código. Al leer las pruebas, se obtiene una comprensión clara de cómo se espera que funcione cada unidad de código.

Unit testing: importante

Unit testing no asegura que nuestro código no tenga errores sino que es una buena práctica para reducirlos

Unit testing: en rust

```
#[test]
```

```
assert!(expresion);
```

```
assert_eq!(v1, v2);
```

```
assert_ne!(v1, v2);
```

Unit testing: en rust

```
comandos:
    cargo test
    cargo test nombre_del_test
    cargo test nombre_con_el_que_empieza


#[ignore]
#[should_be_panic(expected="mensaje del panic")]
```

Unit testing: coverage

Es una métrica utilizada en el contexto de pruebas de software que indica el porcentaje de código fuente que ha sido ejecutado durante la ejecución de las pruebas. Se utiliza para evaluar la efectividad de las pruebas en términos de qué tan bien cubren el código y qué áreas del código no están siendo probadas.

La cobertura de código tiene como objetivo identificar las áreas del código que no han sido probadas y que podrían contener errores o comportamientos inesperados. Una alta cobertura de código no garantiza la ausencia de errores, pero proporciona una mayor confianza en la calidad del software, ya que implica que se han realizado esfuerzos para probar exhaustivamente el código.

Unit testing: ventajas de coverage

- ★ **Identificación de código no probado:** La cobertura de código permite identificar las partes del código que no han sido ejecutadas durante las pruebas, lo que indica áreas de riesgo potencial donde los errores podrían estar presentes.
- ★ **Guía para la creación de pruebas:** La cobertura de código puede ayudar a guiar la creación de pruebas adicionales al revelar las áreas que requieren una mayor cobertura. Esto asegura que las pruebas se enfoquen en las partes críticas del código.
- ★ **Medición de la calidad de las pruebas:** La cobertura de código se utiliza como una métrica para evaluar la calidad de las pruebas. Una alta cobertura indica que se han realizado esfuerzos para probar exhaustivamente el código y puede indicar una mayor confiabilidad del software.

Unit testing: coverage en rust

tarpaulin: https://crates.io/crates/cargo-tarpaulin

comandos:

```
cargo tarpaulin --target-dir src/coverage --skip-clean --exclude-files=target/debug/*
cargo tarpaulin --target-dir src/coverage --skip-clean --exclude-files=target/debug/*
--out html
```