Rust 2023

clase 6

Temario

- Closures
- Iterators
- Manejo de errores
- Archivos

Closures

Closures: ¿Qué son?

Los closures son funciones anónimas que se pueden guardar en una variable o pasar como argumentos a otras funciones.

Se puede crear el closure en un lugar y luego llamarlo en otro para evaluar algo en un contexto diferente.

Permiten la reutilización de código

Closures: en el contexto

```
#[derive(Debug, PartialEq, Copy, Clone)]
enum Color {
    Rojo,
    Azul,
}
struct Inventario {
    remeras: Vec<Color>,
}
```

Closures: en el contexto

```
impl Inventario {
    fn get_color(&self, color_favorito: Option<Color>) -> Color {
        color_favorito.unwrap_or_else(|| self.mas_stockeado())
    }
    fn mas_stockeado(&self) -> Color {
        let mut num_rojo = 0;
        let mut num_azul = 0;
        for color in &self.remeras {
            match color {
                Color::Rojo => num_rojo += 1,
                Color::Azul => num_azul += 1,
            }
        }
        if num_rojo> num_azul {
            Color::Rojo
        } else {
            Color::Azul
        }
    }
}
```

Closures: en el contexto

```
fn main() {
    let store = Inventario {
        remeras: vec![Color::Azul, Color::Rojo, Color::Azul],
    };
    let u_pref1 = Some(Color::Rojo);
    let get_color1 = store.get_color(u_pref1);
    println!( "El usuario prefiere {:?} y obtiene {:?}",
        u_pref1, get_color1
    );
    let u_pref2 = None;
    let get_color2 = store.get_color(u_pref2);
    println!("El usuario prefiere: {:?} y obtiene {:?}",
        u_pref2, get_color2
    );
}
```

Closures: inferencia de tipos

```
fn main() {
    fn  sumar_v1    (x: u32, y:u32) -> u32 { x + y }
    let sumar_v2 = |x: u32, y:u32| -> u32 { x + y };
    let sumar_v3 = |x, y|                 { x + y };
    let sumar_v4 = |x, y|                   x + y  ;

    sumar_v1(3, 3);
    sumar_v2(3, 3);
    sumar_v3(3, 3);
    sumar_v4(3, 3);
}
```

Closures: inferencia de tipos

```
fn main() {
    let sumer_v3 = |x, y|              { x + y };
    let sumar_v4 = |x, y|                x + y  ;

    let c1 = Caja{dato:2};
    let c2 = Caja{dato:5};
    println!("{}", sumar_v3(c1, c2));
    println!("{}", sumar_v4(c1, c2));
}
```

Closures: inferencia de tipos

```
use std::ops::Add;
#[derive(Clone, Copy)]
struct Caja{
    dato:i32
}
impl Add for Caja{
    type Output = i32;

    fn add(self, otro: Self) -> i32 {
        self.dato + otro.dato
    }
}
```

Closures: referencias y ownership

```
fn main() {
    let list = vec!["uno".to_string(), "dos".to_string(), "tres".to_string()];
    println!("Antes de definir el closure:  {:?}", list);

    let pide_prestado = || println!("Desde el closure:  {:?}", list);

    println!("Antes de llamar al closure:  {:?}", list);
    pide_prestado();
    println!("Luego de llamar al closure  {:?}", list);
}
```

Closures: referencias y ownership

```
fn main() {
    let mut list = vec![1, 2, 3];
    println!("Antes de definir el closure:  {:?}", list);

    let mut prestado_mutable = || list.push(7);

    prestado_mutable();
    println!("Luego de llamar al closure:  {:?}", list);
}
```

Closures: referencias y ownership

```
fn main() {
    let mut list = vec![1, 2, 3];
    println!("Antes de definir el closure:  {:?}", list);

    let mut prestado_mutable = move  || {list.push(7); println!("{:?}", list)};

    prestado_mutable();
    println!("Luego de llamar al closure:  {:?}", list);
}
```

```
  --> src/main.rs:185:50
   |
179|     let mut list = vec![1, 2, 3];
   |         -------- move occurs because `list` has type `Vec<i32>`, which does not implement the `Copy` trait
...
182|     let mut prestado_mutable = move  || {list.push(7); println!("{:?}", list)};
   |                                ------- ---- variable moved due to use in closure
   |                                |
   |                                value moved into closure here
...
185|     println!("Luego de llamar al closure: {:?}", list);
   |                                            ^^^^ value borrowed here after move
```

Closures: como parámetros

```
#[derive(Debug)]
struct Rectangulo {
    ancho: u32,
    alto: u32,
}
fn main() {
    let mut arreglo = [
        Rectangulo { ancho: 10, alto: 1 },
        Rectangulo { ancho: 3, alto: 5 },
        Rectangulo { ancho: 7, alto: 12 },
    ];
    arreglo.sort_by_key(|r| r.ancho);
    println!("{:#?}", arreglo);
}
```

Iterators

Iterator: ¿Qué es?

Iterator es un patrón de diseño de comportamiento que te permite recorrer elementos de una colección sin exponer su representación subyacente (lista, pila, árbol, etc.).

Iterator en rust

Rust en sus collections implementa el trait Iterator para poder utilizarlas como tal

```
#[derive(Debug)]
struct Algo{
    d:i32,
}
impl Default for Algo{
    fn default() -> Self { Algo{d:10} }
}
fn main(){
    use std::collections::LinkedList;
    use std::collections::HashMap;
    let mut v = Vec::from([Algo::default(), Algo::default(), Algo::default(), Algo::default()]);
    let mut l = LinkedList::from([Algo::default(), Algo::default(), Algo::default(), Algo::default()]);
    let mut hm = HashMap::from([(1, Algo::default()),(2, Algo::default()),(3,Algo::default())]);
    let mut iter_v = v.iter();
    let mut iter_l = l.iter();
    let mut iter_hm = hm.iter();
```

Iterator en rust

```
..
let  iter_v_clone = iter_v.clone();
iter_v.next();
iter_v.cycle();
iter_v.enumerate();
iter_v.take(3);
iter_v.step_by(3);
iter_v.skip(2);
let otro = iter_v_clone.chain(iter_l);
for i in otro{
    println!("{:?}", i);
}
```

Iterator y closures

```
..
    iter_v.all(closure);
    iter_v.any(closure);
    iter_v.filter(closure);
    iter_v.filter_map(closure);
    iter_v.skip_while(closure);
```

Iterator en rust

```
let mut otro = iter_v_clone.chain(iter_l);
while let Some(e) = otro.next(){
    println!("{:?}", e);
}
```

Iterator: implementando en struct

```
struct Caja{
    c:i32
}
impl Default for Caja{
    fn default() -> Self {
        Caja { c: 0 }
    }
}
impl Iterator for Caja{
    type Item = i32;
    fn next(&mut self) -> Option<i32> {
        if self.c < 10{
            self.c +=1;
            return Some(self.c)
        }
        None
}
```

Iterator: implementando en struct

```
fn main(){
    let mut a = Caja::default();
    while let Some(v) = a.next(){
        println!("{}", v);
    }
}
```