Funciones y closures en Rust:

Rust posee una sintaxis para definir funciones muy similar a la de otros lenguajes. Veamosla en acción.

Funciones simples:

Las funciones se definen con la palabra reservada **fn**. Posteriormente indicamos el nombre de la función y los argumentos de entrada. Si la función devuelve un valor se debe poner una flecha y el tipo del valor devuelto. Para devolver un valor se puede usar **return** o se puede dejar la última línea sin punto y coma. Funciona de forma similar a como vimos con **let** y las asignaciones complejas.

```
fn suma(a: i32, b: i32) -> i32{
    a+b
}

fn main(){
    let a = 5;
    let b = 42;
    let c = suma(a,b);
    println!("{}",c);
}
```

Rust no admite devolver varios valores a la vez, pero es posible usar tuplas y simularlo.

```
fn string_length_and_lines(txt: &String) -> (usize,usize) {
    (txt.len(),txt.lines().count())
}

fn main(){
    let s = String::from("Europe's Skies - Alexander Rybak\nSuper Strut - 
Deodato\nEl Cóndor Pasa - Uña Ramos");

    let (length,lines) = string_length_and_lines(&s);
    println!("La lista de canciones tiene una longitud de {} caracteres y {} 
líneas",length,lines);
}
```

Funciones de primer nivel:

En Rust las funciones son elementos de primer nivel, lo que quiere decir que pueden pasarse entre si como argumentos.

Veamos un ejemplo. En este caso uso un **for-in** que todavía no hemos visto, pero la idea es comprobar como se puede pasar una función como argumento.

```
fn ladrar(){
    println!("Guau")
}

fn hacer_n_veces(f: fn(), n: i64){
    for _ in 0..n{
        f();
    }
}

fn main(){
    hacer_n_veces(ladrar,42);
}
```

Genericidad:

Las funciones en Rust admiten genericidad, lo que quiere decir que la misma función es compatible con distintos tipos, siempre respetando las normas. Usualmente, para denominar al tipo genérico se usa <T>. <T> se sustituye en cada caso por el tipo en cuestión. Entonces si en el primer parámetro de la función hemos puesto un i32 y <T>también está en el segundo, este también tiene que ser i32.

```
fn mayor<T: PartialOrd + Copy>(n: &Vec<T>) -> T{
    let mut max: T = n[0];
    for &i in n.iter(){
        if i > max{
            max = i;
        }
    }
    max
}

fn main(){
    let v: Vec<i32> = vec![1,2,7,8,123,4];
    let max = mayor(&v);
    println!("{}",max);
}
```

En este ejemplo además vemos genericidad restringida, la cuál es la más habitual. Básicamente T puede ser cualquier tipo pero debe de implementar las **traits** PartialOrd (para hacer la comparación) y **Copy** (elementos que se copian, en vez de moverse). La genericidad en Rust es un campo muy extenso, no obstante para entenderla bien, tendremos que entrar en los **traits.**

Closures:

Los closures o funciones anónimas son funciones sin nombre que se definen y se consumen. En lenguajes como Python existen bajo la palabra reservada **lambda**. En Rust se definen con barras verticales.

```
fn main(){
    let potencia = |x| x*x;
    let pot_5 = potencia(5);
    println!("{}",pot_5);
}
```

Como es de imaginar podemos usar expresiones más complejas entre llaves.

```
fn main(){
    let potencia_sup = |x| {
        let y = x+1;
        y*y
    };
    let pot_5 = potencia_sup(5);
    println!("{}",pot_5);
}
```

Con esto sabemos lo suficiente de funciones y genericidad para avanzar a otros temas.