# Referencias y préstamos en Rust

Continuando con Rust 101, tutorial de Rust en español vamos a adentrarnos en un concepto clave para entender y programar en Rust. Rust es un lenguaje seguro como hemos dicho, esto implica que no puede haber NULLs ni tampoco fugas de memoria. Es decir, tiene que gestionar la memoria de forma eficiente pero dejarla directamente al criterio del programador. Algunos lenguajes disponen de un recolector de basura (GC, *Garbage Collector*), que reduce la eficiencia del lenguaje. Go, Swift o Java usan GC. Luego existen lenguajes que dejan la memoria al descubierto, como C y C++. Rust toma un enfoque diferente, ya que no deja la memoria al descubierto ni usa GC. Para ello el compilador realiza una tarea de propietarios y préstamos que veremos a continuación.

## Las reglas:

1. Cada variable en Rust, es propietaria de un valor.
2. Un valor solo puede tener un propietario a la vez.
3. Cuando el propietario desaparece, el valor lo hace tambien,(de forma automática).

Así que cuando una variable desaparece del *entorno*, el dato del que era dueño es liberado. Estas reglas vistas así permiten poca flexibilidad en la programación. Es por ello que los dueños pueden prestar sus valores.

## La trait Copy y la trait Clone:

En primer lugar hay que distinguir los distintos tipos de comportamiento en funcion del tipo de valor. Si implementan la trait Copy (la mayoría de tipos primitivos), entonces su comportamiento por defecto es de copia. Son datos en los que la copia es barata y rápida y no influye que existan varias copias de lo mismo. Cualquier cosa que no requería *malloc* en C puede ser Copy en Rust. Se trata de valores que se almacenan en el stack.

Otra trait es Clone, que permite hacer copias de datos más complejos, por ejemplo de un vector. Veamos un ejemplo de un código sin Copy pero con Clone.

```
fn main(){
    let s1 = String::from("Adios - Xavier Cugat");
    let s2 = s1;
    println!("{}",s1); // ERRROR

}
```

Este código da error porque el tipo String no implementa Copy. En la línea *let s2 = s1;* lo que ha ocurrido en realidad ha sido que se ha movido el valor, lo que significa que se ha transferido el valor (String) el cual era propiedad de s1 a s2, pasando este a ser su propietario. Como s1 ya no es propietaria del valor no se puede operar con él. Esto pasa en los tipos que no implementan Copy, los cuales transfieren la propiedad de sus valores a otra variable. Si queremos hacer una *copia* real, tendremos que recurrir al clonado. El tipo String implementa Clone así que es posible generar otro dato String exactamente igual pero independiente del original.

```
fn main(){
    let s1 = String::from("Adios - Xavier Cugat");
    let s2 = s1.clone();
    println!("{}",s1);

}
```

## Implicaciones:

Esto tiene unas implicaciones sorprendentes, y es que pasar una variable tal cual a una función si no es del tipo Copy implica que ¡perdemos el acceso a ese valor! Pongamos un ejemplo:

```
fn main(){
    let s1 = String::from("Bolero - Maurice Ravel ");
    f(s1);
    println!("{}",s1);
}
```

Este código da error. Al hacer la llamada a la función f hemos transferido la propiedad del valor de s1 a f. Por ello, cuando intentamos hacer el print no vamos a poder ya que s1 ya no es dueña de la cadena de texto. Para solucionar estos problemas tenemos los préstamos.

## Prestando en Rust:

Rust permite prestar los valores de los que una variable es dueña de dos maneras: solo lectura o con permisos de escritura. Siempre ha de cumplirse la norma de que solo puede haber una con permiso de escritura pero puede haber infinidad con permisos de lectura. Y nunca se pueden dar las dos cosas a la vez.

El préstamo de lectura se realiza con el operador &, que lo que hace es una referencia de lectura al valor. La variable sigue siendo asi propietaria de este, solo lo ha prestado y entrega una referencia a dicho valor.

```
fn main(){
    let s1 = String::from("Bolero - Maurice Ravel ");
    f(&s1);
    println!("{}",s1);
}
```

Este código ya sí funciona y podemos ver en pantalla Bolero - Maurice Ravel.

Si queremos hacer el préstamo con permisos de escritura, hemos de usar &mut. Sin entrar muchos detalles, así funcionaría un préstamo de escritura.

```
fn f(s: &mut String){
    s.push_str(" & Adios - Xavier Cugat");
}

fn main(){
    let mut s1 = String::from("Bolero - Maurice Ravel");
    f(&mut s1);
    println!("{}",s1);
}
```

Así tenemos por pantalla la frase *Bolero - Maurice Ravel & Adios - Xavier Cugat*.

Este tipo de sintaxis, a priori compleja, nos ayudará mucho con la concurrencia pero de momento ya hemos visto suficiente.