.UBAfiuba
FACULTAD DE INGENIERÍA

Taller de Programación I

Ownership / Lifetimes / Traits / Generics

Ing. Pablo A. Deymonnaz
pdeymon@fi.uba.ar

Facultad de Ingeniería
Universidad de Buenos Aires

Por qué *ownership*?

ownership= tiene que ver con las referencia y prestamos. Quien es el dueño de la variable cando se le pasa a una funcion o cuando pierdo la propiedad de dueño

Motivación:

- eliminar categorías completas de bugs (punteros nulos, uso despues de liberación, liberación doble, buffer overruns, invalidación de iteratores, condiciones de carrera por datos)
- al restringir los programas que son considerados válidos,
- a la vez que no se paga un costo en tiempo de ejecución.

Por qué *ownership*?

Cómo surgen estos bugs?

- **Aliasing**: mas de un camino para alcanzar la misma memoria.
- **Mutación**: cambiar memoria en común.

Cualquiera de las das, por sí misma, no causa problemas.

- **Solución**: usar una o la otra, pero no ambas a la vez.
- **Inspiración**: C++ RAII
- **Formalización**: sistema de tipos subestructural linear.

Qué es *ownership*?

Inspiración: C++ RAII

- Un recurso se inicializa cuando se asigna (allocate).
- La inicialización es realizada por los constructores de clase.
- La limpieza es realizada por los destructores de clase.
- Los destructores se invocan automáticamente cuando un objeto sale de alcance.

En C o C++, una instancia de una clase es dueña de otro objeto al que apunta.
Esto en general significa que el objeto que adueña puede decidir cuando liberar al objeto adueñado: cuando el dueño es destruido, a su vez destruye todas sus posesiones.

Reglas de *ownership*

1. Cada valor en Rust tiene una variable que es su dueña.
2. Solo puede existir una dueña a la vez.
3. Cuando la dueña sale de alcance, el valor sera liberado.

- Cada valor tiene un unico dueño que determina su *lifetime*. Cuando el dueño es liberado (*dropped*), el valor adueñado también.
- Una variable es dueña de su valor. Cuando el control sale del bloque en el cual la variable esta declarada, la variable se destruye, por lo que su valor tambien se destruye.

Taxonomia de *ownership*

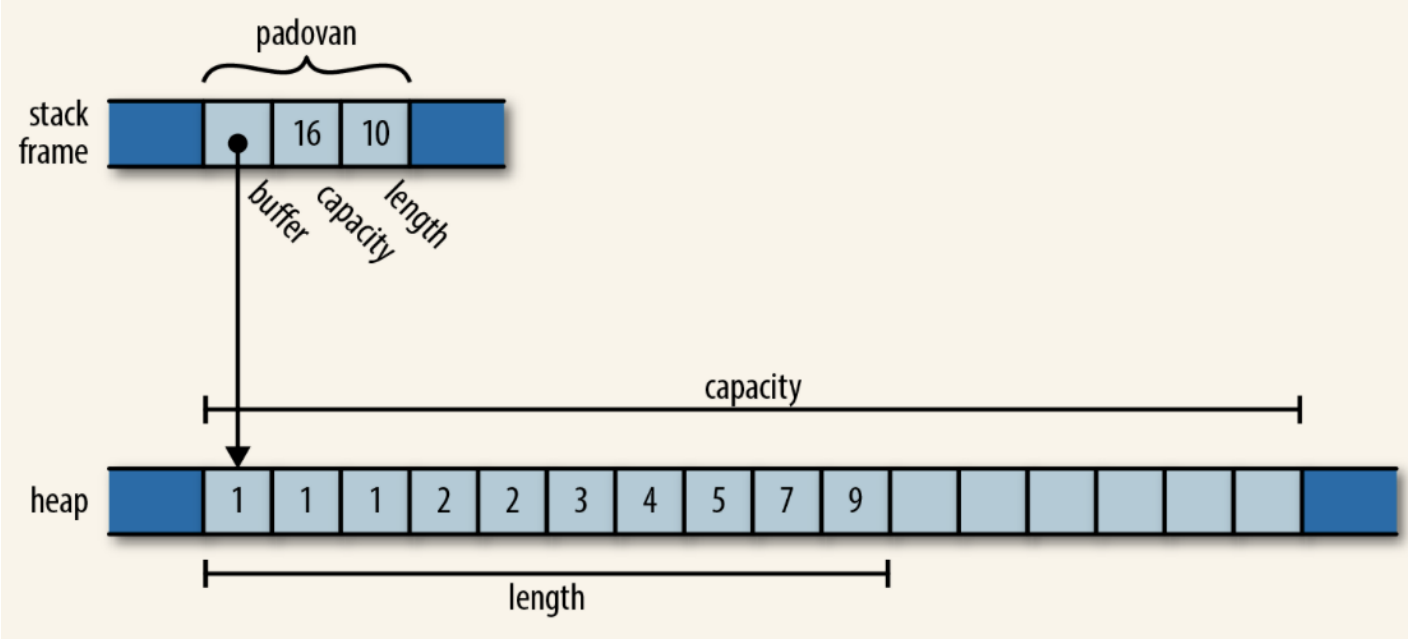
Se puede usar el sistema de tipos para validar que algo sea mutable o compartido, pero no ambas a la vez.

Tipo	Ownership	Alias	Mutable?
`T`	Adueñado		`X`
`&T`	Referencia compartida	`X`	
`&mut T`	Referencia mutable		`X`

Bases de *ownership*

```
fn print_padovan() {
    let mut padovan = vec![1,1,1]; // allocated here
    for i in 3..10 {
        let next = padovan[i-3] + padovan[i-2];
        padovan.push(next);
    }
    println!("P(1..10) = {:?}", padovan);
} // padovan here
```

Bases de *ownership*



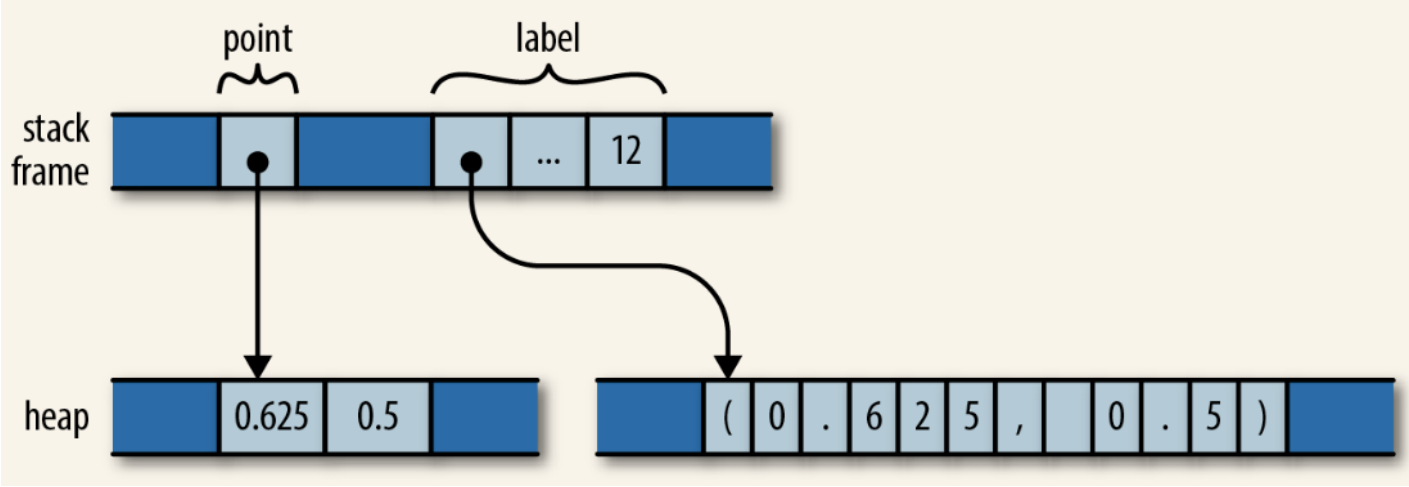
Bases de *ownership*

```
{
    let point = Box::new((0.625, 0.5)); // point asignado
    let label = format!("{:?}", point); // label asignado
    assert_eq!(label, "(0.625, 0.5)");
}                                       // ambos liberados
```

Bases de *ownership*

- Un `Box<T>` es un puntero a un valor de tipo T almacenado en *heap*.
- Invocando `Box::new(v)` asigna espacio en *heap*, mueve el valor `v` a ese espacio, y retorna un `Box` que apunta a ese espacio.
- Un `Box` es dueño del espacio al que apunta, de tal manera que cuando el `Box` es destruido, se libera su espacio tambien.

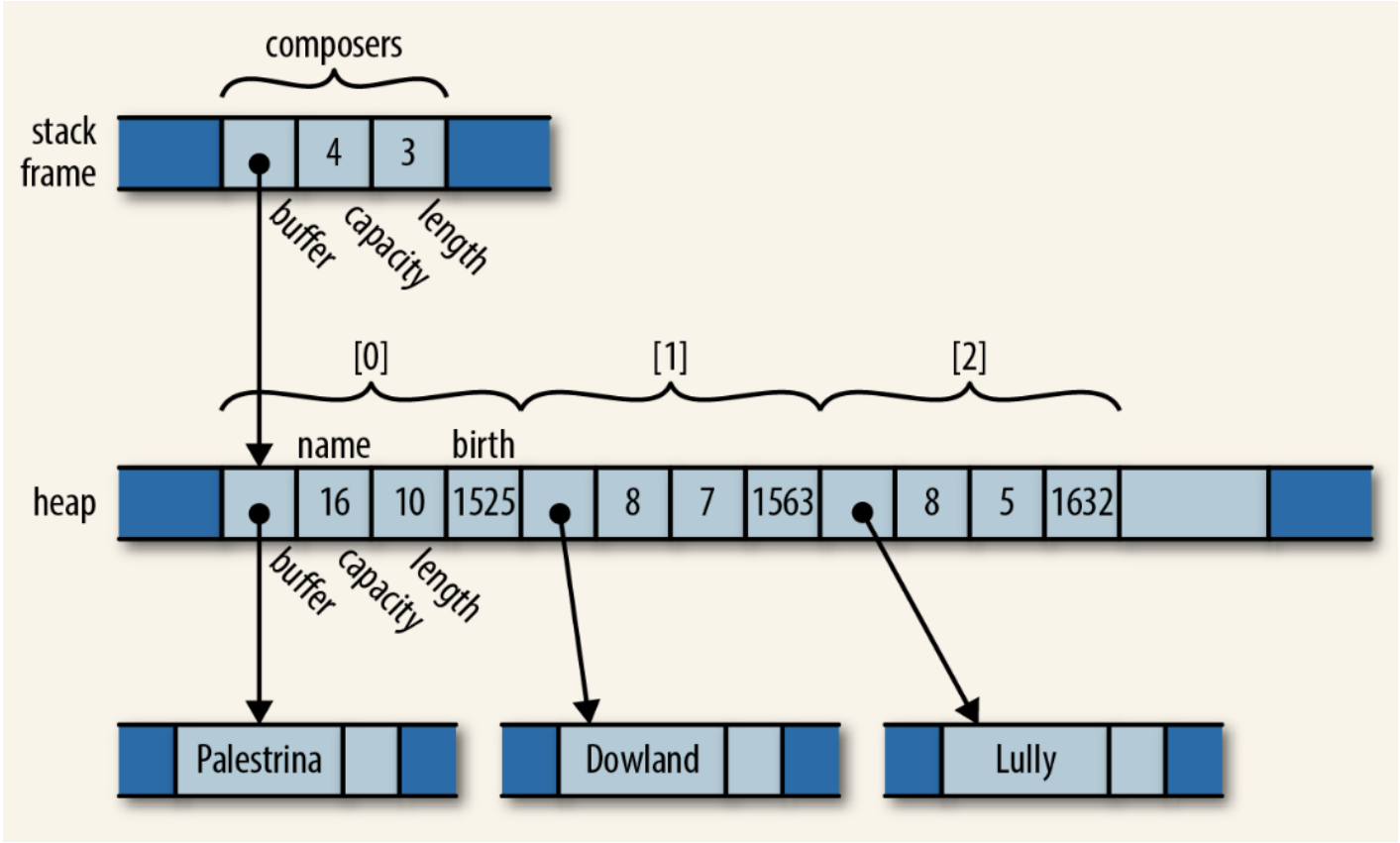
Bases de *ownership*



Bases de *ownership*

```
struct Person { name: String, birth: i32 }
let mut composers = Vec::new();
composers.push(Person {
    name: "Palestrina".to_string(),
    birth: 1525
    });
composers.push(Person {
    name: "Dowland".to_string(),
    birth: 1563
    });
composers.push(Person {
    name: "Lully".to_string(),
    birth: 1632
    });
for composer in &composers {
    println!("{}, born {}", composer.name, composer.birth);
}
```

Bases de *ownership*



Bases de *ownership*

Que implican estas reglas hasta ahora:

- Cada valor tiene un unico dueño, es fácil decidir cuando hay que destruir un valor.
- Un valor puede ser dueño de otros valores. Se sigue que los valores dueños y adueñados forman árboles.
- Los programes Rust normalmente no destruyen explícitamente valores. Destruir un valor es eliminarlo de alguna manera del árbol de pertenencia.
- Rust es menos poderoso que otros lenguajes*.

Movimientos

- Se pueden mover valores de un dueño a otro. Esto permite construir, reordenar, y tirar abajo el árbol.
- La librería estandar provee tipos de punteros que llevan conteo de referencias, como `Rc` y `Arc`. Esto les permite tener múltiples dueños, bajo ciertas restricciones.
- Se puede "tomar prestado una referencia" a un valor. Las referencias son punteros que no adueñan los valores a los que apuntan, y tienen *lifetimes* limitados.

Operaciones que mueven

- Para la mayoría de los tipos, operaciones tales como asignar un valor a una variable, pasarla a una función, o retornarla de una función no copian el valor, lo *mueven*.
- El origen cede su pertenencia del valor al destinatario, y pasa a ser *no inicializada*, el destinatario ahora controla el *lifetime* del valor.

Operaciones que mueven

- Asignación a una variable: si se mueve un valor a una variable que ya estaba inicializada, Rust destruye (*drop*) el valor anterior de la variable.

```
let mut a = Person{
    name: "Marcos".to_string(),
    birth: 1942
    };
a = Person{
    name: "Pedro".to_string(),
    birth: 1935
    }; // Marcos destruido
```

Operaciones que mueven

Otros lugares donde ocurren movimientos, ademas de inicialización y asignación:

- Retornar valores de una función.

```
let x = f(); // f's return value moved into x
```

- Construcción de nuevos valores.

```
let x = Person{name: "Mateo".to_string(), birth: 1928};
```

- Pasar valores a una función.

```
let mut x = Person{
    name: "Pedro".to_string(), birth: 1935};
f(x);
x.birth = 1942; // error, x moved into f
```

Operaciones que mueven

```
let s =
    vec![
        "udon".to_string(),
        "ramen".to_string(),
        "soba".to_string()
        ];
let t = s;
let u = s;
```

Operaciones que mueven

```
error[E0382]: use of moved value: `s`
 --> src/main.rs:9:13
  |
2 |  let s =
  |      - move occurs because `s` has type `Vec<String>`,
  |        which does not implement the `Copy` trait
...
8 |  let t = s;
  |          - value moved here
9 |  let u = s;
  |          ^ value used here after move
```

Movimientos y control de flujo

Principio General: Si es posible que a una variable se le haya movido el valor, y no ha sido asignado un valor nuevo de manera definitoria, se considera no incializada. Si una variable aun tiene un valor despues de evaluar la expresión de la condición de un if, entonces podemos usarlo en ambas ramas.

```
let x = vec![10, 20, 30];
if c {
    f(x); // ... ok to move from x here
} else {
    g(x); // ... and ok to also move from x here
}
h(x) // error: x uninitialized here if either path uses it
```

Movimientos y control de flujo

Mover una variable en un bucle esta prohibido:

```
let x = vec![10, 20, 30];
while f() {
    g(x); // error: x se mueve en la primera iteración,
          // estará no inicializada en la segunda
}
```

Movimientos y contenido indexado

```rust
// Construir un vector de strings "101", "102", ... "105"
let mut v = Vec::new();
for i in 101 .. 106 {
    v.push(i.to_string());
}

// Acceder a elementos aleatorios del vector.
let third = v[2];
let fifth = v[4];
```

Movimientos y contenido indexado

```
error[E0507]: cannot move out of index of `Vec<String>`
 --> src/main.rs:8:17
  |
8 |  let third = v[2];
  |              ^^^^
  |              |
  |              move occurs because value has type `String`
  |              which does not implement the `Copy` trait
  |              help: consider borrowing here: `&v[2]`
```

Movimientos y contenido indexado

Antes de buscar una forma de vencer el *borrow checker*, se puede considerar cambiar el patrón del código. Cada uno de estos metodos quita un elemento del vector, pero de tal manera que deja el vector en un estado tal que todos sus componente estan inicializados. Otras colecciones de la libreria estandar ofrecen patrones similares.

Movimientos y contenido indexado

1. Desapilar un valor del final del vector:

```
let fifth = v.pop().unwrap();
assert_eq!(fifth, "105");
```

2. Mover un valor del medio del vector, y reemplazarlo por el ultimo:

```
// element into its spot:
let second = v.swap_remove(1);
assert_eq!(second, "102");
```

3. Intercambiar otro valor por el que estamos moviendo:

```
let third = std::mem::replace(&mut v[2],
                "substitute".to_string());
assert_eq!(third, "103");
```

Movimientos y contenido indexado

Los tipos de colecciones como `Vec` generalmente ofrecen metodos para consumir todos sus elementos en un bucle.

```
let v = vec!["liberté".to_string(),
             "égalité".to_string(),
             "fraternité".to_string()];

for mut s in v {
    s.push('!');
    println!("{}", s);
}
```

Cuando se pasa un vector a un bucle de manera directa, `for ... in v`, esto mueve el vector de `v`, dejando a `v` no inicializada.

Tipos que se copian

- Asignar un valor de un tipo que es `Copy` copia el valor, en lugar de moverlo.
- El origen de la asignacion permanece inicializado y usable, con el mismo valor que tenia antes.
- Pasar tipos `Copy` a funciones y constructores opera similarmente.
- En Rust, cada movimiento es una copia no profunda* byte a byte, que deja al origen no inicializado.
- Las copias son similares, solo que el origen permanece inicializado.

Tipos que se copian

- Los tipos estandar `Copy` incluyen a todos los tipos numericos de enteros de maquina y punto flotante, `char` y `bool`, y un algunos mas.
- Una tupla o arreglo de tamaño fijo cuyos componentes son `Copy` es, a su vez, `Copy`.
- Solo tipos para los cuales una copia bit a bit alcanza, pueden ser `Copy`.
- En general, cualquier tipo que necesita hacer algo especial cuando un valor es destruido no puede ser `Copy`.

Tipos que se copian

- Por defecto, los tipos struct y enum no son `Copy`.
- Si todos los campos de tu struct son a su vez `Copy`, puedes hacer que tu tipo lo sea tambien usando un atributo `#[derive(Copy, Clone)]` antes de la definición:

```
#[derive(Copy, Clone)]
struct Label { number: u32 }
```

Referencias - Ejemplo 1

```rust
fn main() {
    let s1 = String::from("hello");

    let len = calculate_length(s1);

    println!("The length of '{}' is {}.", s1, len);
}

fn calculate_length(s: String) -> usize {
    let length = s.len(); // len() returns the length
    length                // of a String
}
```

Referencias - Ejemplo 1

```
error[E0382]: borrow of moved value: `s1`
 --> test.rs:6:53
  |
2 |  let s1 = String::from("hello");
  |      -- move occurs because `s1` has type
  |         `std::string::String`, which does
  |         not implement the `Copy` trait
3 |
4 |  let len = calculate_length(s1);
  |                             -- value moved here
5 |
6 |  println!("The length of '{}' is {}.", s1, len);
  |                                        ^^ value
  |  borrowed here after move

error: aborting due to previous error
```

Referencias - Ejemplo 1

```
fn main() {
    let s1 = String::from("hello");

    let (s2, len) = calculate_length(s1);

    println!("The length of '{}' is {}.", s2, len);
}

fn calculate_length(s: String) -> (String, usize) {
    let length = s.len(); // len() returns the length
    (s, length)           //  of a String
}
```

Referencias - Ejemplo 1

```
fn main() {
    let s1 = String::from("hello");

    let len = calculate_length(&s1);

    println!("The length of '{}' is {}.", s1, len);
}

fn calculate_length(s: &String) -> usize {
    s.len()
}
```

Referencias

- La mayoria de los tipos de punteros vistos en ejemplos han sido puntoros que son dueños de los valores referenciados, e.g. `String`, `Vec` y `Box` son dueños de los datos a los que apuntan.
- Las referencias en Rust son otro tipo de puntero que no se adueñan del valor al que apunta, sino que "toma prestado" (*borrow*).
- Las referencias no tienen efecto sobre los *lifetimes* de sus referentes.
- Las referencias no deben *sobrevivir*a sus referentes.
- Se le dice *tomar prestado* (*borrowing*) a crear una referencia a un valor.
- Las referencias en sí no son mas que direcciones de memoria.
- Una referencia permite acceder a un valor sin afectar su pertenencia.

Referencias

Tipo	Ownership	Alias	Mutable?
`T`	Adueñado		`X`
`&T`	Referencia compartida	`X`	
`&mut T`	Referencia mutable		`X`

Referencias

- Hay dos tipos de referencias
  - Una **referencia compartida** permite leer pero no modificar su referente. Se pueden tener a la vez tantas referencias compartidas a un valor particular como se desee.
  - Si se tiene una **referencia mutable** a un valor, se puede tanto leer como modificar ese valor. Sin embargo, no se puede tener ninguna otra referencia activa de ningun tipo a la misma vez.
- Las referencias en Rust nunca son nulas:
  - No existe un valor por defecto inicial para una referencia.
  - Rust no convierte enteros en referencias.
  - Para indicar la ausencia de un valor, se usa `Option<T>`

Referencias - Ejemplo 2

```rust
use std::collections::HashMap;

type Table = HashMap<String, Vec<String>>;

fn main() {
    let mut table = Table::new();
    table.insert(
        "Gesualdo".to_string(),
        vec![
            "many madrigals".to_string(),
            "Tenebrae Responsoria".to_string(),
        ],
    );
```

Referencias - Ejemplo 2

```
    table.insert(
        "Caravaggio".to_string(),
        vec![
            "The Musicians".to_string(),
            "The Calling of St. Matthew".to_string(),
        ],
    );
    show(table);
}
```

Referencias - Ejemplo 2

```
fn show(table: Table) {
    for (artist, works) in table {
        println!("works by {}:", artist);
        for work in works {
            println!("  {}", work);
        }
    }
}
```

Referencias - Ejemplo 2

```
...
show(table);
assert_eq!(table["Gesualdo"][0], "many madrigals");
error[E0382]: use of moved value: `table`
  --> references_show_moves_table.rs:29:16
   |
28 |     show(table);
   |          ----- value moved here
29 |     assert_eq!(table["Gesualdo"][0], "many madrigals");
   |                ^^^^^ value used here after move
   |
   = note: move occurs because `table` has type
           `HashMap<String, Vec<String>>`,
           which does not implement the `Copy` trait
```

Referencias - Ejemplo 2

```
fn show(table: &Table) {
    for (artist, works) in table {
        println!("works by {}:", artist);
        for work in works {
            println!("  {}", work);
        }
    }
}

...
show(&table);
```

Lifetimes

Seguridad de referencias

- Rust intenta asignar a cada tipo de referencia en un programa un *lifetime* que satisface las restricciones impuestas por como se usa.
- Un *lifetime* es un tramo, un área, del programa para el cual esa referencia se puede usar de manera segura: un bloque lexico, una sentencia, una expresión, el alcance de una variable, etc.
- El *lifetime* de una variable debe contener o abarcar el *lifetime* de la referencia que toma prestada su valor.
- Mas alla del punto donde una variable sale de alcance, la referencia seria una puntero colgante o invalido.
- Las *lifetimes* no se declaran, definen, o modifican explicitamente. Son funcion unicamente del uso que se le da a valores y a sus referencias y las necesidades y restricciones que surgen de este uso.

Tomando prestada una variable local

```
{
    let r;
    {
        let x = 1;  // declaramos una variable `x` con un valor de `1`
        r = &x;     // tomamos una referencia a `x` y la asignamos a `r`
    } // `x` sale de su alcance y se descarta aquí
    assert_eq!(*r, 1); // error: lee memoria que
                       // solia ocupar `x`
}                      // error: intentamos acceder a `x` a través de `r`, pero `x` ya no existe
```

En este código, x es una variable local que solo existe dentro del bloque interno. Cuando ese bloque termina, x sale de su alcance y se descarta. Sin embargo, todavía tienes una referencia a x en r, y cuando intentas usar r para acceder a x, obtienes un error en tiempo de compilación. Esto es porque Rust garantiza la seguridad de la memoria al asegurarse de que no puedas acceder a una variable que ya no existe.
Este error es un ejemplo de cómo el sistema de “ownership” (propiedad) y “borrowing” (préstamo) de Rust ayuda a prevenir errores y garantizar la seguridad de la memoria.

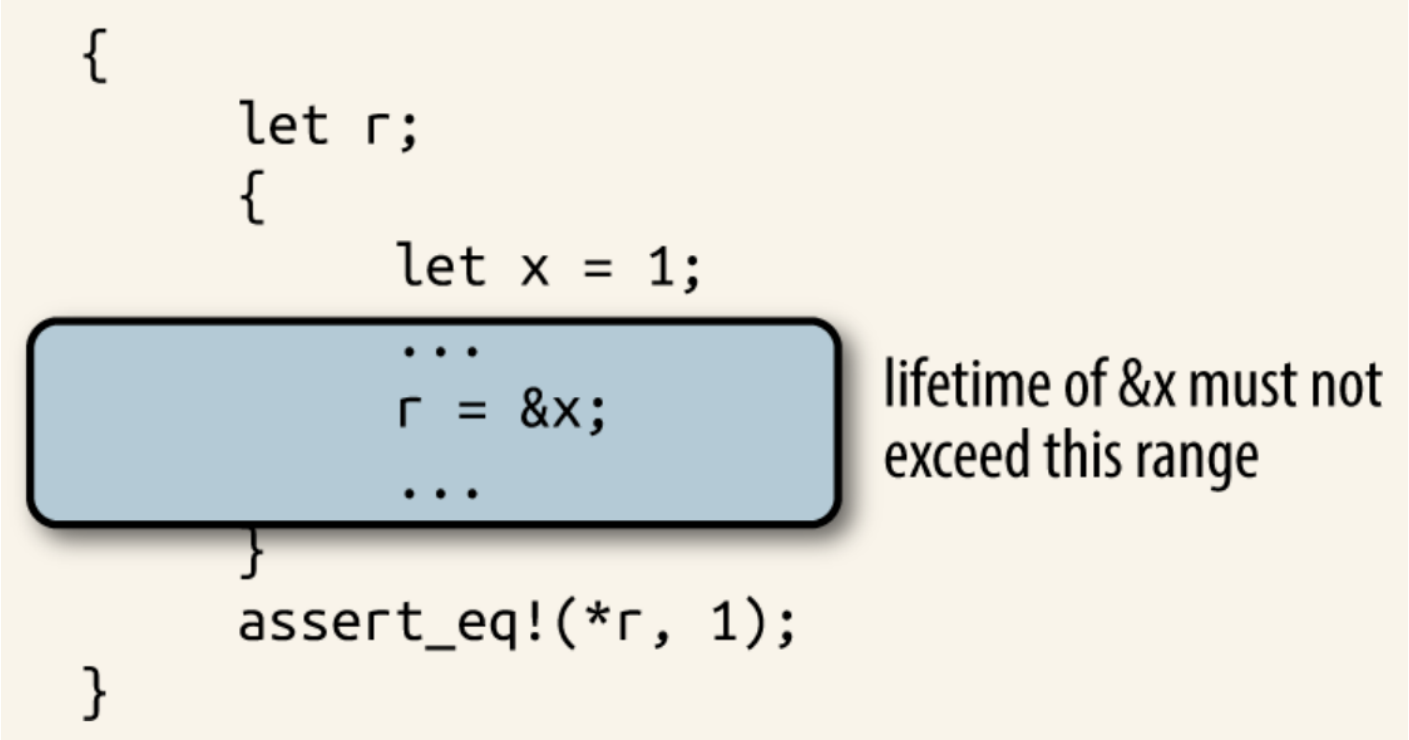
Tomando prestada una variable local

```
error[E0597]: `x` does not live long enough
 --> test2.rs:6:29
  |
5 |         r = &x;
  |         ^^^^^^ borrowed value does not live long enough
6 |     }
  |     - `x` dropped here while still borrowed
7 |     assert_eq!(*r, 1); // error: lee memoria que
  |     ------------------ borrow later used here
  |
error: aborting due to previous error
```

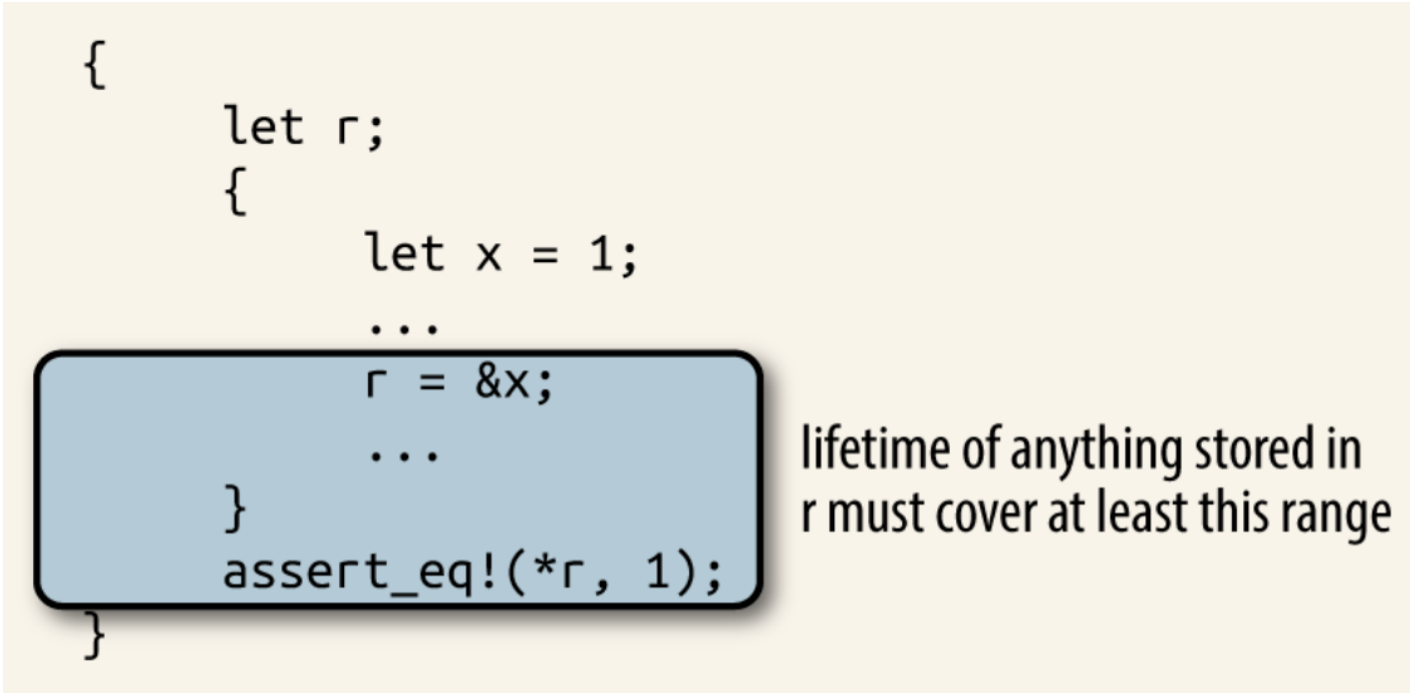
Tomando prestada una variable local

- Hay tres lifetimes cuyas relaciones tenemos que entender.
- Las variables `r` y `x` tienen cada una un lifetime, que se extiende desde el punto en cual son inicializados hasta el punto en que salen de alcance.
- La tercera lifetime es la del tipo de la referencia que toma prestado de `x` y almacenamos en `r`.

Tomando prestada una variable local

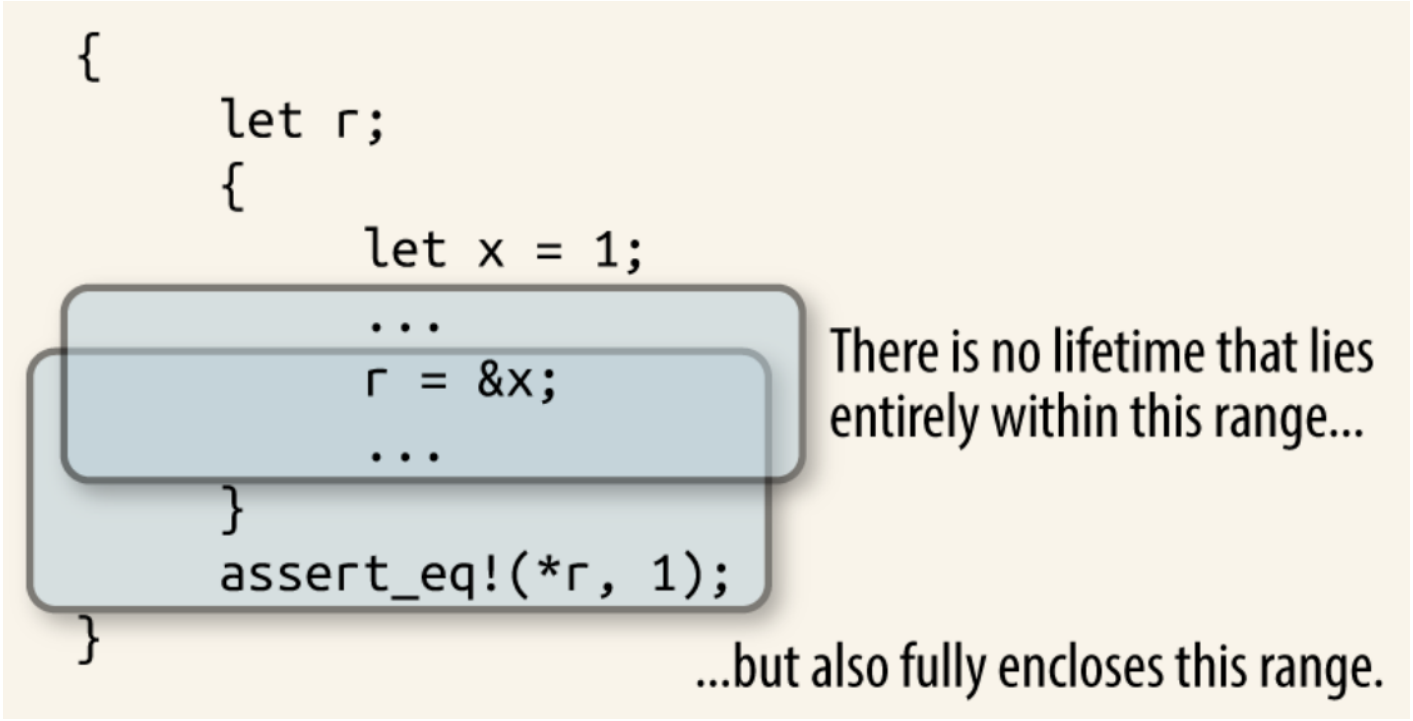


Tomando prestada una variable local



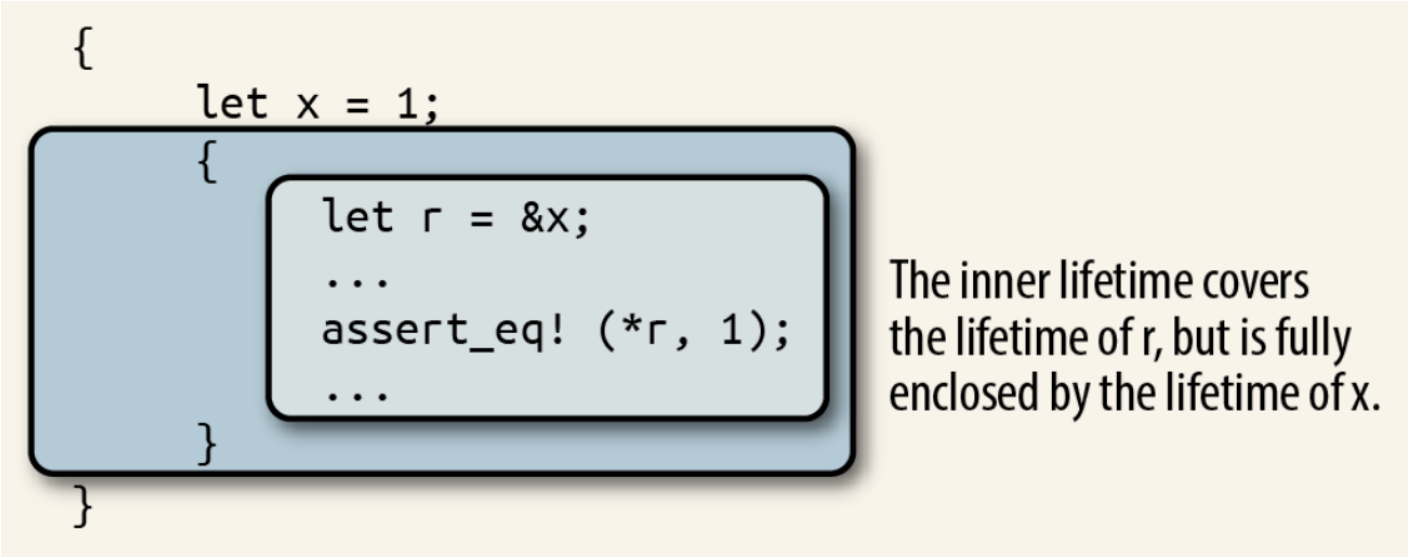
Si se almacena una referencia en una variable `r`, el tipo de la referencia debe ser valido para todo el lifetime de la variable, desde el punto en que es inicializado hasta el punto en que sale de alcance.

Tomando prestada una variable local



El primer tipo de restriccion limita cuan larga puede ser el lifetime de una referencia, mientras que el segundo tipo limita cuan chica puede ser.

Tomando prestada una variable local



Recibiendo referencias por parametro

Supongamos una funcion `f` que toma una referencia y la almacena en una variable global:

```
// Este codigo tiene varios problemas, y no compila.
static mut STASH: &i32;
fn f(p: &i32) { STASH = p; }
```

Un static es un valor que se crea cuando un programa comienza y dura hasta que termina.* (El sistema de modulos controla visibilidad, asi que son solo globales en lifetimes.)

- Todo static debe ser inicializado.
- Los statics mutables son inherentemente inseguros para threads.
- Aun en programas con un unico thread, los valores statics deben ser accedidos a traves de `unsafe`.

Recibiendo referencias por parametro

```
static mut STASH: &i32 = &128;
fn f(p: &i32) { // still not good enough
    unsafe {
        STASH = p;
    }
}
```

Recibiendo referencias por parametro

```
error[E0621]: explicit lifetime required in the type of `p`
 --> test3.rs:5:17
  |
2 | fn f(p: &i32) {
  |         ---- help: add explicit lifetime `'static`
  |                    to the type of `p`: `&'static i32`
...
5 |         STASH = p;
  |                 ^ lifetime `'static` required

error: aborting due to previous error
```

Recibiendo referencias por parametro

```
static mut STASH: &i32 = &128;
fn f<'a>(p: &'a i32) { // aun insuficiente
    unsafe {
        STASH = p;
    }
}
```

- STASH esta viva durante toda la ejecucion del programa,
- pero el tipo de la referencia que contiene debe tener un lifetime de la misma longitud (`'static`).
- Sin embargo, el lifetime de la referencia `p` es `'a`, la cual podria ser cualquiera, siempre y cuando abarca la llamada a f.

Recibiendo referencias por parametro

```
static mut STASH: &i32 = &10;

fn f(p: &'static i32) {
    unsafe {
        STASH = p;
    }
}
```

Los lifetimes en signaturas de funciones le permiten a Rust evaluar las relaciones entre las referencias que le pasas a la funcion y aquellas que la funcion retorna, y asegurarse que se estan usando de manera segura.

Structs que contienen referencias

```
struct S { // No compila
    r: &i32
}
```

Siempre que un tipo de referencia aparece en la definicion de otro tipo, se debe escribir explicitamente su lifetime.

```
error[E0106]: missing lifetime specifier
 --> test4.rs:2:10
  |
2 |     r: &i32
  |          ^ expected lifetime parameter
```

```
struct S<'a> {
    r: &'a i32
}
```

Cada valor que se crea de tipo `S` obtiene un lifetime nuevo `'a`, el cual se restringe segun como se utilize ese valor.

Structs que contienen referencias

```
let s;
{
    let x = 10;
    s = S { r: &x };
}
assert_eq!(*s.r, 10); // bad: reads from dropped `x`
```

La expresión {S { r: &x }} crea un valor nuevo de S con un lifetime 'a.
Cuando se almacena &x en el campo r, se restringe a 'a de tal manera que debe caer enteramente dentro del lifetime de x's lifetime.

Structs con parametros de lifetime

Si un struct contiene un campo que es una referencia, se deben nombrar los lifetimes de esas referencias.

```
struct Extrema<'elt> {
    greatest: &'elt i32,
    least: &'elt i32,
}
```

Esto significa que dado un lifetime cualquiera `'elt`, se pueden crear valores `Extrema<'elt>` que contienen referencias con ese lifetime.

Structs con parametros de lifetime

```
fn find_extrema<'s>(slice: &'s [i32]) -> Extrema<'s> {
    let mut greatest = &slice[0];
    let mut least = &slice[0];

    for i in 1..slice.len() {
        if slice[i] < *least    { least    = &slice[i]; }
        if slice[i] > *greatest { greatest = &slice[i]; }
    }
    Extrema { greatest, least }
}
```

Aqui, `find_extrema` toma prestado elementos de `slice`, la cual tiene lifetime `'s`, el struct `Extrema` que se retorna tambien usa `'s` como lifetime de sus referencias.

Structs con parametros de lifetime

Rust siempre infiere los lifetime de parametros de llamadas, de tal manera que cuando se invoca `find_extrema` no se explicitan:

```
let a = [0, -3, 0, 15, 48];
let e = find_extrema(&a);
assert_eq!(*e.least, -3);
assert_eq!(*e.greatest, 48);
```

Genericidad y Traits

- Todo lenguaje tiene herramientas para manejar efectivamente la duplicacion de conceptos.
- La capacidad para escribir codigo que opera sobre valores de distintos tipos, incluso tipos que aun no han sido definidos, se llama polimorfismo.
- Rust soporte polimorfismo a traves de dos caracteristics relacionadas: genericidad, y traits.
- Traits son la version de Rust de interfaces o clases base abstractas.
- Un trait define un conjunto de metodos que pueden ser implementados para un tipo dado.

Genericidad y Traits

- Agregar un trait a un tipo no cuesta memoria, e invocar los metodos de un trait para un tipo no tiene el sobrecosto de llamadas a metodos virtuales.
- Generics es el otro tipo de polimorfismo en Rust.
- Como los templates de C++, permiten parametrizar codigo con tipos, ya sea funciones o definiciones de otros tipos como structs o enums.

Definir un trait

```
pub trait Summary {
    fn summarize(&self) -> String;
}
```

Implementar un trait sobre un tipo

```
pub struct NewsArticle {
    pub headline: String,
    pub location: String,
    pub author: String,
    pub content: String,
}

impl Summary for NewsArticle {
    fn summarize(&self) -> String {
        format!("{}, by {} ({})", self.headline,
            self.author, self.location)
    }
}
```

Implementar un trait sobre un tipo

```rust
pub struct Tweet {
    pub username: String,
    pub content: String,
    pub reply: bool,
    pub retweet: bool,
}

impl Summary for Tweet {
    fn summarize(&self) -> String {
        format!("{}: {}", self.username, self.content)
    }
}
```

Implementar un trait sobre un tipo

```
use chapter10::{self, Summary, Tweet};

fn main() {
    let tweet = Tweet {
        username: String::from("horse_ebooks"),
        content: String::from(
            "of course, as you already know, people",
        ),
        reply: false,
        retweet: false,
    };

    println!("1 new tweet: {}", tweet.summarize());
}
```

Implementar un trait sobre un tipo

- Una restriccion que tienen las implementaciones de traits es que podemos implementar un trait para un tipo solamente si o bien el trait o bien el tipo es local a nuestro crate.
- E.g. se pueden implementar traits de la libreria estandar como `Display` sobre un tipo propio como `Tweet` como parte del crate.
- Tambien se puede implementar el trait `Summary` sobre un tipo de la libreria estandar como `Vec`, porque el trait es local al crate.
- Lo que no se puede hacer es implementar traits externos sobre tipos externos.
- Esta regla asegura que el codigo de otros no puede romper el codigo del crate local, y viceversa. Sin esta regla dos crates podrian implementar el mismo trait para el mismo tipo, y el compilador no sabria cual implementacion usar.

Traits de utilidad

- Drop
- Sized
- Clone
- Copy
- Deref / DerefMut
- AsRef / AsRefMut
- From / Into
- ToOwned

Genericidad

La genericidad y traits estan relacionados porque frecuentemente, al definir una funcion con un parametro por tipo, se necesita restringir los tipos posibles a aquellos que implementen ciertas traits.

Genericidad

```
struct Point<T> {
    x: T,
    y: T,
}

fn main() {
    let integer = Point { x: 5, y: 10 };
    let float = Point { x: 1.0, y: 4.0 };
}
```

Otros temas

- Structs with lifetime parameters
- Interior mutability (Cell & RefCell)
- Lifetimes & clausuras
- Lifetimes & iteradores
- Lifetimes & concurrencia
- Lifetimes & traits

Bibliografía

- **The Rust Programming Language**, *https://doc.rust-lang.org/book/*
  - Chapter 4 Understanding Ownership
  - Chapter 10 Generic Types, Traits, Lifetimes
- **Programming Rust: Fast, Safe Systems Development**, 1st Edition, Jim Blandy, Jason Orendorff. 2017.
  - Chapter 4 Ownership
  - Chapter 5 References
  - Chapter 11 Traits and Generics