

Parameter passing

The parameter passing is an assignment of the value of the actual parameter to the formal parameter

The move and copy concepts apply as well!!

On a call, ownership passes from:

- argument to called function's parameter
- returned value to caller's receiver

```
fn main() {  
    let s1 = String::from("hello");  
    let s2 = id(s1); //s1 moved to arg  
    println!("{}", s2); //id's result moved to s2  
    println!("{}", s1); //fails  
}  
fn id(s:String) -> String {  
    s // s moved to caller, on return  
}
```

RUN ►

DEBUG ▾

STABLE ▾

...

```
1 fn main() {
2     let s1 = String::from("hello");
3     let s2 = id(s1);
4     println!("{}" ,s1);
5 }
6
7 fn id(s:String) -> String {
8     s
9 }
```

⋮⋮⋮

Execution

```
|  
= note: `#[warn(unused_variables)]` on by default  
  
error[E0382]: borrow of moved value: `s1`  
--> src/main.rs:4:15  
|  
2 |     let s1 = String::from("hello");  
|         -- move occurs because `s1` has type `String`, which does not implement the `Copy` trait  
3 |     let s2 = id(s1);  
|             -- value moved here  
4 |     println!("{}" ,s1);  
|             ^^ value borrowed here after move
```

```
fn main() {
    let s = String::from("hello"); // s comes into scope

    takes_ownership(s);        // s's value moves into the function...
                                // ... and so is no longer valid here

    let x = 5;                 // x comes into scope

    makes_copy(x);            // x would move into the function,
                            // but i32 is Copy, so it's okay to still
                            // use x afterward

} // Here, x goes out of scope, then s. But because s's value was moved, nothing
  // special happens.

fn takes_ownership(some_string: String) {
    println!("{}", some_string);
} // Here, some_string goes out of scope and `drop` is called. The backing
  // memory is freed.

fn makes_copy(some_integer: i32) {
    println!("{}", some_integer);
} // Here, some_integer goes out of scope. Nothing special happens.
```

```
fn main() {  
    let s1 = gives_ownership();      // gives_ownership moves its return value into s1  
    let s2 = String::from("hello");  // s2 comes into scope  
    let s3 = takes_and_gives_back(s2); // s2 is moved into takes_and_gives_back, which also  
                                    // moves its return value into s3  
} // Here, s3 goes out of scope and is dropped. s2 goes out of scope but was  
// moved, so nothing happens. s1 goes out of scope and is dropped.  
  
fn gives_ownership() -> String {      // gives_ownership will move its return value into the calling function  
  
    let some_string = String::from("hello"); // some_string comes into scope  
  
    some_string                      // some_string is returned  
}  
  
fn takes_and_gives_back(a_string: String) -> String {  
  
    a_string // a_string is returned and moves out to the calling function  
}
```

An illustrative example

```
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 of a String
    (s, length)
}
```

adesso però andiamo a vedere un po' pragmaticamente, un esempio, un po più articolato, cioè supponiamo di voler calcolare sostanzialmente il valore della lunghezza di una stringa e vedete, quindi abbiamo questa funzione, la cosa che fa prende una stringa e restituisce come risultato la stringa e il valore visto come intero senza segno del calcolato e questo perché lo fa in questo modo? La funzione prende il parametro e quindi vuol dire che il chiamante gli ha trasferito l' ownership allora questo punto poi la funzione dei ritrasferire indietro la ownership gli esempi che abbiamo visto prima, ad esempio la give_ownership o la takes_and_give_back fanno vedere esattamente questo trasferimento indietro dal chiamante al chiamato della ownership allora questo punto bisogna quindi, non solo restituire il valore, ma bisogna anche restituire la ownership per fare del valore ovviamente della stringa, parametro attuale per fare in modo che poi venga utilizzata dal chiamante per gli affaretti del chiamante. Allora andiamo a vedere come è fatto il main, quindi il main vedete quello che fa crea una stringa a questo punto chiama la funzione calculate_length, quindi perde l' ownership allora a questo punto deve crearsi la tupla risultato, che è fatta della coppia, la stringa che in realtà è il riferimento al valore che gli era già in possesso del chiamante e la lunghezza che ha un valore intero nuovo e a questo punto riesce a fare operazioni che nell'esempio qui è spiegato dalla stampa perché è un'operazione che viene fatta su questa coppia di valori su questa tupla di valori. E' evidente che c'è un passaggio di ownership avanti e indietro e uno potrebbe pensare di poter operare meglio relativamente a la programmazione di questo semplice programma.

Parameter passing: borrowing

functions may take a reference to an object as a parameter instead of taking ownership of the value.

L'idea è di fare un passaggio per parametri e che sostanzialmente è modello nuovo di passaggio per parametro voi quando avete visto i linguaggi di programmazione vi è stato detto che, ad esempio in c avete il passaggio per valore, in ml avete il passaggio per valore, in ocaml avete il passaggio per valore, in quasi tutti i linguaggi di programmazione vecchi, come ad esempio anche Java il passaggio è per valore. Esisteva anche un altro passaggio dei parametri, che è il passaggio per riferimento, voleva dire che quando io definivo la segnatura della funzione del metodo, dicevo che il valore che mi aspettavo era un riferimento a un altro valore e mettevo della notazione opportuna. Questo voleva dire che il chiamante e chiamato condividevano tramite il cammino d'accesso nome del parametro formale o cammino di accesso nome del parametro attuale condividevano il valore. Ma non c'era questa nozione di ownership che fa la differenza rispetto al vecchio passaggio dei parametri. Molto spesso uno dice erroneamente che java, c, c sharp hanno anche il passaggio per riferimento in realtà non lo hanno, non lo hanno perché il meccanismo di passaggio è sempre il passaggio per nome, è solo che ci possono essere delle situazioni in cui al nome, al valore associato al nome è un riferimento ad un oggetto, a una locazione di memoria, allora come effetto del fatto che valori sono riferimenti abbiamo una simulazione del passaggio per riferimento, ma non abbiamo il passaggio per riferimento. Quello che qui i progettisti di Rust hanno fatto è introdurre un meccanismo di passaggio dei parametri che permetteva di avere un riferimento all'oggetto, quindi di prendere in prestito l'accesso all'oggetto, ma senza che il chiamante perdesse l'ownership del valore, quindi, è proprio un prestito in questo senso borrow.

Allora andiamo a vederlo nell'esempio che abbiamo visto in precedenza come funziona e creiamo la solita stringa e a questo punto abbiamo la funzione calculate semplicemente restituisce il valore della lunghezza della stringa, ma la cosa importante è andare a vedere le annotazioni che a questo punto abbiamo sulla segnatura della funzione e sulla invocazione della funzione stessa all'interno del main. Nella segnatura della funzione abbiamo subito la cosa ovvia, quindi viene restituito il valore usize quindi è un valore senza segno e vedete che il parametro, è l'&String vuol dire che si aspetta un riferimento a un oggetto di tipo stringa. Quando lo chiamo gli do esattamente il riferimento all'oggetto di tipo stringa, bene per chi conosce il linguaggio di programmazione, questo qui è esattamente il meccanismo di passaggio che avevamo in ALGOL, solo che al posto del mettersi l'& uno ci metteva reference quindi esattamente un'idea rivitalizzata.

```
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()
}
```

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

    let len = calculate_length(&s1);

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

references: allow one to refer to some value without taking ownership of it

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

allora questo punto, quando noi vediamo nel dettaglio questa operazione, abbiamo un riferimento che viene trasmesso, quindi chiamante e chiamato condividono lo stesso valore, il punto è che quando il chiamante chiama il chiamato con un valore per riferimento, gli passa il valore ma non perde l'ownership del valore e questo non perdere l'ownership del valore vuol dire che qui, ad esempio, può continuare in seguito a utilizzare s1, vedete la differenza rispetto al codice precedente è che a quel punto gli trasmette, fa il calcolo, non perde l'ownership e quindi lo può utilizzare in seguito. questo è l'aspetto che lo differenzia rispetto a questa che era un po più involuta come pattern di programmazione

allora a questo punto possiamo esemplificare il pass by borrowing, quindi la notazione sintattica che viene messa quando facciamo la chiamata o quando facciamo la definizione della segnatura della funzione è l'and commerciale e il nome del parametro, quindi questo vuol dire che quello è un passaggio per riferimento in cui chiamante e chiamato condividono questa struttura ma non viene perso l'ownership da parte del chiamante. Quando nello scopo viene perso, si può ovviamente cancellare come prima, però soltanto la perdita alla fine dello scope, quindi questo è quello che permette in questa funzione per quanto riguarda s1 di utilizzarlo dopo perché qui se non avessi messo l'& a questo punto avrei trasferito anche l'ownership, mettendola vuol dire che lo posso continuare a utilizzare fintanto che lo scope è attivo quindi fintanto che non arriva sino a questo punto e quindi lo utilizzo e similmente nella segnatura della funzione quando io uso l' annotation &s sto indicando che quello è un passaggio per riferimento,

The syntactic notation `&s1` lets us create a reference that *refers* to the value of `s1` but does not own it.

s1 does not own the value it points to, hence the value will not be dropped when the reference goes out of scope.

The signature of the function uses the notation `&s` to indicate that the type of the parameter is a reference.

Borrowing and mutability

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

    change(&s);
}

fn change(some_string: &String) {
    some_string.push_str(", world");
}
```

adesso dobbiamo andare a vedere che cosa succede con la borrowing e la mutabilità quando facciamo il passaggio dei parametri in modo particolare, andiamo a vedere un esempio semplice, cioè, se abbiamo questa funzione change che prende una stringa e usa il passaggio per borrowing perchè c'è un &String poi la cosa che fa è che vuole fare un operazione di push e che fa l' append della stringa che ha ricevuta come parametro con la stringa world, sostanzialmente si vuol creare il solito hello word e quello che viene fatto è creare nel main la s che contiene hello la passiamo per parametro e lo diamo in pasto alla funzione change e ci aspettiamo di creare hello word, dato che non perdiamo l'ownership.

RUN ▶

...

DEBUG ▾

STABLE ▾

...

```
1 fn main() {
2     let s = String::from("hello");
3
4     change(&s);
5 }
6
7 fn change(some_string: &String) {
8     some_string.push_str(", world");
9 }
10
11
```

beh, se lo mandiamo in esecuzione, in realtà il compilatore di rust ha qualcosa da ridire perchè voi state facendo un passaggio di un parametro per riferimento, ma lo state modificando all'interno della funzione allora se lo state modificando all'interno della funzione dovete indicare che lo volette modificare, non potete farlo semplicemente con un passaggio per riferimento, vuol dire che se voi lo passate in questo modo, con questa notazione sintattica non potete fare un'operazione di modifica proprio per il fatto che la s è dichiarata per default immutable

Execution

Standard Error

```
Compiling playground v0.0.1 (/playground)
error[E0596]: cannot borrow `*some_string` as mutable, as it is behind a `&` reference
--> src/main.rs:8:5
7 | fn change(some_string: &String) {
   |     ----- help: consider changing this to be a mutable reference: `&mut String`
8 |     some_string.push_str(", world");
   |     ^^^^^^^^^ `some_string` is a `&` reference, so the data it refers to cannot be borrowed as mutable
error: aborting due to previous error
```

Easy to fix: mutable references

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

    change(&mut s);
    println!("The value of s is '{}'", s);
}

fn change(some_string: &mut String) {
    some_string.push_str(", world");
}
```

RUN ▶

...

DEBUG ▾

STABLE ▾

...

```
1 fn main() {
2     let mut s = String::from("hello");
3
4     change(&mut s);
5     println!("The value of s is '{}'", s);
6 }
7
8 fn change(some_string: &mut String) {
9     some_string.push_str(", world");
10}
11
12
13
```

dobbiamo rendere la stringa mutable, una volta che abbiamo reso la stringa mutable dobbiamo chiamare la funzione change, notate anche nella funzione change dobbiamo dirle, guarda, passiamo un riferimento e questo riferimento fa riferimento a un qualcosa che può essere modificato, quindi la notazione è esattamente una notazione che dice quali sono i vincoli e le caratteristiche dell'operazione che si possono fare tramite il passaggio dei parametri allora questo punto gli passiamo il parametro, che è un riferimento modificabile e poi a quel punto, dopo possiamo continuare a utilizzare la s per tutto quello che abbiamo detto prima prima potevamo fare del borrowing con dei riferimenti immutable, ora possiamo fare anche del borrowing con riferimenti mutable

```
Compiling playground v0.0.1 (/playground)
Finished dev [unoptimized + debuginfo] target(s) in 3.15s
Running `target/debug/playground`
```

Standard Output

```
The value of s is 'hello, world'
```

```
fn main() {  
    let mut s = String::from("hello");  
  
    let r1 = &mut s;  
    let r2 = &mut s;  
  
    println!("{}{}, {}", r1, r2);  
}
```

**Constraint on
mutable references:
we can have only
one mutable
reference to a
particular piece of
data in a particular
scope**

ci sono degli aspetti di rust che vanno studiati e analizzati con particolare attenzione qui non abbiamo nemmeno bisogno di avere la funzione ausiliaria per poter fare l'esempio vedete, creiamo una stringa mutabile poi quello che facciamo creiamo due riferimenti r1 e r2 che sono dei riferimenti mutable alla stringa s bene, rust ci dice che noi possiamo avere soltanto un mutable reference a un particolare valore all'interno ovviamente di un particolare scope, tutte le osservazioni che vi sto facendo sono sempre all'interno di un particolare scope, infatti se io lancio il playground su quel programma il compilatore dice esattamente guarda questo primo borrowo mi va bene, il secondo no, ho il vincolo di averne uno solo e quindi ti attacchi e infatti dà errore si la compilazione non va avanti quindi vuol dire che ci sono dei vincoli

RUN ►

...

DEBUG ▾

STABLE ▾

...

```
1 fn main() {
2     let mut s = String::from("hello");
3
4     let r1 = &mut s;
5     let r2 = &mut s;
6
7     println!("{} , {}", r1, r2);
8 }
9
10
11
12
```

⋮⋮⋮

Execution

Standard Error

```
Compiling playground v0.0.1 (/playground)
error[E0499]: cannot borrow `s` as mutable more than once at a time
--> src/main.rs:5:14
```

```
4     let r1 = &mut s;
5         ----- first mutable borrow occurs here
6     let r2 = &mut s;
7         ^^^^^^ second mutable borrow occurs here
8
9     println!("{} , {}", r1, r2);
```

Mutable and Immutable references

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

    let r1 = &s; // no problem
    let r2 = &s; // no problem
    let r3 = &mut s; // BIG PROBLEM

    println!("{}, {}, and {}", r1, r2, r3);
```

prima il problema era che avevamo dei vincoli su dei borrow immutabili e se le prendiamo non immutable, cioè se noi avevamo una stringa mutable, la solita hello poi a questo punto vedete abbiamo dei borrow dei riferimenti immutable r1 e r2 di fatto non stiamo facendo delle operazioni non particolarmente significativo perché non possiamo modificare, abbiamo solo l'accesso, non perdiamo l'ownership da parte di s, poi quello che invece facciamo è un borrow mutable a questo punto, sempre la solita s e li ho scritto big problem perché vedete se noi lo mandiamo in esecuzione il primo passa, abbiamo un immutabile borrow, il secondo passa abbiamo un immutable borrow e qui abbiamo un mutable borrow, quindi abbiamo ancora una volta una situazione dove all'interno dello stesso scope abbiamo questo vincolo e in realtà la cosa che uno potrebbe fare potrebbe giocare sulle caratteristiche dello scope,

RUN ►

...

DEBUG ▾

STABLE ▾

...

```
1 fn main() {  
2     let mut s = String::from("hello");  
3  
4     let r1 = &s; // no problem  
5     let r2 = &s; // no problem  
6     let r3 = &mut s; // BIG PROBLEM  
7  
8     println!("{} , {} , and {}", r1, r2, r3);  
9 }  
10  
11  
12  
13
```

⋮⋮⋮

Execution

Standard Error

```
Compiling playground v0.0.1 (/playground)  
error[E0502]: cannot borrow `s` as mutable because it is also borrowed as immutable  
--> src/main.rs:6:14
```

```
4     let r1 = &s; // no problem  
      -- immutable borrow occurs here  
5     let r2 = &s; // no problem  
6     let r3 = &mut s; // BIG PROBLEM  
          ^^^^^^ mutable borrow occurs here  
7
```

RUN ►

...

DEBUG ▾

STABLE ▾

...

```
1 fn main() {  
2     let mut s = String::from("hello");  
3  
4     let r1 = &s; // no problem  
5     let r2 = &s; // no problem  
6     println!("{} and {}", r1, r2);  
7     // r1 and r2 are no longer used after this point  
8  
9     let r3 = &mut s; // no problem  
10    println!("{}", r3);  
11 }
```

infatti vedete basta questo semplice accorgimento che utilizziamo r1 e r2 il compilatore si accorge che dopo averle utilizzate una delle due non sono più utilizzata, allora essendo non più utilizzata a questo punto possiamo fare un borrow mutable con r3 il vincolo ce l'abbiamo perché questo punto è un solo mutable borrow che stiamo utilizzando a quello scope e lo ribadisco, lo scope è quella porzione di spazio del codice che è dopo l'utilizzo di r1 e r2 a questo punto, infatti lo stampiamo e vedete stampa esattamente quello che uno si aspetta, la visibilità e il tempo in cui l'accesso tramite delle variabili all'interno di una scope è significativo ha in realtà un altro parametro dietro che, come vedremo tra un attimo, si chiama il lifetime, cioè il tempo di vita di quando all'interno di questo scope vale ancora l'assegnamento tra il nome e il valore .

:::

Execution

Standard Error

```
Compiling playground v0.0.1 (/playground)  
Finished dev [unoptimized + debuginfo] target(s) in 1.89s  
Running `target/debug/playground`
```

Standard Output

```
hello and hello  
hello
```

sono partiti dalla progettazione di Rust con l'idea di vedere se era possibile continuare ad avere un controllo a basso livello tipo del c cercando di avere la memory safety senza aver garbage collector tutto questo discorso del borrow vi serve esattamente per avere un unico accesso al dato è evidente che a questo punto posso operare con la drop in modo safe, allora a questo punto la domanda che uno si pone, siamo sicuri che non possiamo ancora scrivere del meccanismo di dangling reference? ricordate sono quelle operazioni che voi potete fare quando all'interno di un linguaggio di programmazione che permette di avere dal programma una free esattamente l'equivalente della drop chi programma libera la memoria, ma in realtà c'è un altro ancora cammino d'accesso attivo, quindi questo cosa vuol dire che libera memoria sullo heap ma avendo ancora un cammino attivo poi quell'altro cammino ad accesso rimane senza il valore associato allora uno si domanda questa è ancora possibile?

Dangling references are still possible?

RUN ▶

...

DEBUG ▼

STABLE ▼

...

```
1 fn main() {
2     let reference_to_nothing = dangle();
3 }
4
5 fn dangle() -> &String {
6     let s = String::from("hello");
7
8     &s
9 }
10
11
12
```

andiamo a vedere questo esempio quello che faccio definisco una funzione dangle, restituisce come valore, un riferimento ad una stringa e come è fatta al suo interno la cosa, crea una stringa, restituisce il valore e poi chiude lo scope quindi viene chiamato il drop e quindi vuol dire che il valore associato a quel riferimento a questo punto viene perso. Nel main, creiamo un riferimento a nothin, quindi esattamente un dangling reference che otteniamo andando a chiamare la funzione dangle e questo è il modo immediato di scrivere un riferimento pendente. Andiamo a vedere il compilatore che cosa ci dice? Si accorge staticamente che questo è un problema e ci dice che c'è un problema, nel senso che stiamo utilizzando il valore del riferimento s con un tempo di vita che non corrisponde al tempo di vita di s, perché il tempo di vita di s corrisponde alla funzione, mentre la stiamo utilizzando in uno scope più grande, quindi con un tempo di vita più grande, ci dice anche guardate per ottenere questa cosa dovresti dargli un tempo di vita che viene chiamato statico che vale per tutta l'esecuzione del programma. Chiaramente fa sempre riferimento a questa nozione che per il momento rimane misteriosa e che volutamente rimane misteriosa perché la introduciamo tra un attimo prima volevo introdurmi il problema.

Execution

Standard Error

```
Compiling playground v0.0.1 (/playground)
error[E0106]: missing lifetime specifier
--> src/main.rs:5:16
5 | fn dangle() -> &String {
   |         ^ expected named lifetime parameter
   |
   = help: this function's return type contains a borrowed value, but there is no value for it to be borrowed from
help: consider using the `<static>` lifetime
|
```

Use after free

```
use std::mem::drop; // equivalent to free()
```

```
fn main() {
    let x = "Hello".to_string();
    drop(x);
    println!("{}", x);
}
```

Quando cerchiamo di riutilizzare un qualcosa dopo averla liberata, allora qui semplicemente stiamo dicendo che vogliamo prendere dalle librerie standard a quelle che sia come sono memoria con la drop, equivalente alla free abbiamo un main, vedete, creiamo dal letterale lo creiamo una stringa, questo il modo di scriverlo, la cancelliamo e poi la vogliamo utilizzare questo sarebbe se questo passasse uno direbbe, sì buonanotte, cioè hai cambiato un casino che comprende le borrowing e questa cosa ancora poco chiara del life time, però puoi fare queste cose bene in realtà il compilatore si accorge anche di quello vedete ti dice, guarda, tu stai facendo un'operazione dato che l'operazione di copia delle stringhe è una shallow copy, quindi a questo punto non ho una copia effettiva, a questo punto il valore viene cancellato e a questo punto stai cercando di accedere a un valore dopo aver perso l'ownership e quindi anche in questo caso il compilatore staticamente riesce a evitare il fatto di usare un valore dopo averlo liberato

RUN ►

...

DEBUG ▾

STABLE ▾

...

```
1 use std::mem::drop; // equivalent to free()
2
3 fn main() {
4     let x = "Hello".to_string();
5     drop(x);
6     println!("{}", x);
7 }
8
9
```

:::

Execution

Standard Error

Compiling playground v0.0.1 (/playground)

error[E0382]: borrow of moved value: `x`

--> src/main.rs:6:18

```
4     let x = "Hello".to_string();
5         - move occurs because `x` has type `String`, which does not implement the `Copy` trait
6     drop(x);
7         - value moved here
8     println!("{}", x);
9             ^ value borrowed here after move
```

Vuol dire che l'idea del sistema dei tipi di rust con il meccanismo della ownership sui valori evita che se tu hai ceduto all'interno del programma la ownership di un valore, poi cerchi di riutilizzare quello stesso nome ti da un errore di tipo, quindi l'ownership è estremamente efficace ed estremamente potente, proprio perché è collegata al tipo dei valori è questo l'aspetto significativo del sistema dei tipi di rust che questa nozione di ownership e lifetime sono dei tipi opportuni che vincolano l'uso dei valori e quindi il fatto che il sistema dei tipi, controllando staticamente questa informazione, lo rende estremamente potente

Use after free

use std::mem::drop; // equivalent to free()

```
fn main() {  
    let x = "Hello".to_string();  
    drop(x);  
    println!("{}", x);  
}
```

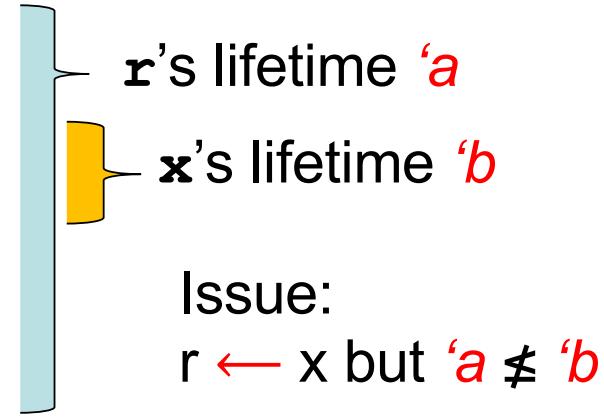
Rust's ownership: when we give up ownership on a value, then subsequent attempts to use the value are no longer valid.

This also protects against double frees, since two calls to drop would encounter a similar ownership type error.

Io quando creo una scope gli do anche un tempo di vita e il life time è un tipo, allora vedete, lì sono due tipi generici, a e b, notate che usiamo la stessa notazione dei generici di ocaml per dire che sono un tipo generico, allora ci dice che lifetime di r è sostanzialmente quella cosa in celeste, mentre il lifetime di x è quella cosa in giallino, sono un lifetime più piccolo dell'altro a questo punto qual è il problema? Il problema è che quando io faccio un assegnamento dico: vado a prendere la parte destra dell'assegnamento, ottengo un valore che assegno al left value, cioè il riferimento nella memoria della variabile. Ma la regola dell'assegnamento, ci dice anche un'altra cosa, il valore della parte destra deve essere compatibile con il tipo aspettato nella parte sinistra dell'assegnamento, x prende y è banale da un punto di vista di un' implementazione, ma non banale dal punto di vista della regola dei tipi. Allora se da una parte abbiamo intero dall'altra parte abbiamo un intero non è un problema, ma già nel linguaggio oggetti con un meccanismo di polimorfismo per sottotipo, la compatibilità ha a che vedere con la gerarchia quindi non è una cosa che si può sorvolare. In questo caso di Rust il fatto di fare un assegnamento da variabili non solo ci dice che hanno lo stesso tipo, ma il tipo deve essere compatibile anche rispetto al suo tempo di vita ho scritto, è vero, si x può andare in r ma quello che abbiamo è che il lifetime di a è più piccolo del lifetime di b non è compatibile con quello di b, cioè io sto cercando di associare a una variabile un lifetime molto grande, un valore che ha un tempo di vita più piccolo, quindi questo vuol dire che i due valori non sono compatibili e infatti il sistema controlla il tipo staticamente e ci dice quando poi dopo vogliamo utilizzare r successivamente ci dice che fallisce perché il lifetime di r è maggiore del lifetime di x.

Lifetime

```
{  
    let r; // deferred init  
    {  
        let x = 5;  
        r = &x;  
    }  
    println!("r: {}", r); //fails  
}
```



- The Rust type checker observes that **x** goes out of scope while **r** still exists
 - A **lifetime** is a *type variable* that identifies a scope
 - **r**'s lifetime '**a**' exceeds **x**'s lifetime '**b**'

Lifetime

Lifetimes are **named** regions of code that a reference must be valid for.

Those regions may be fairly complex, as they correspond to paths of execution in the program

Il lifetime è sostanzialmente un meccanismo che identifica delle regioni che rappresentano delle porzioni di codice.

```
fn main() {  
    let x = 0;  
    let y = &x;  
    let z = &y;  
}
```

LIFETIME VIEW –DONE BY RUST

```
'a: {  
    let x: i32 = 0;  
    'b: {  
        let y: &'b i32 = &'b x;  
        'c: {  
            let z: &'c &'b i32 = &'c y;  
        }  
    }  
}
```

Noi vediamo questa sintassi zuccherata, quello che succede nel retrobottega e per questo qui ci ho messo done by rust nel senso, nel retrobottega quello che succede è che vengono messe nel linguaggio intermedio in evidenza quali sono le regioni del codice, in modo particolare, questo è un lifetime 'a e viene associato a questo blocco più esterno che è x. Il valore associato è una costante, che vale per tutta la durata del programma, quando vado a creare la y ha un life time' b che corrisponde al blocco più interno, notate nella sintassi non ci sono i blocchi più interni cioè nel senso, proprio per il fatto che io ho messo questa struttura a blocchi per indicare porzioni di codice per renderle significative con una nozione sintattica. Quando io vado ad associare a y e gli dico che ha un tempo di vita gli do anche un tempo di vita al valore della x che qui sto mettendo e stessa cosa fatta per la z, vuol dire che tutte le variabili hanno associato un loro tempo di vita qui abbiamo questo z associata a b, abbiamo un c vedete che prende una y e vedete che qui la dipendenza dato che z non so qui non ho dichiarato il suo valore allora questo punto vedete che al suo interno il tempo è esattamente il life time è b perché non so bene ancora che cosa è associato a z perché qui non ho fatto alcun assegnamento. All'interno della rappresentazione intermedia ho rappresentata esattamente il life time di tutti gli elementi che sono presenti nel brano, quindi per ogni variabile sono in grado di poter dire qual è il suo life time e descriverlo internamente

```
fn main() {  
let x = 0;  
let z;  
let y = &x;  
z = y;  
}  
  
'a: {  
    let x: i32 = 0;  
    'b: {  
        let z: &'b i32;  
        'c: {  
            let y: &'b i32 = &'b x;  
            z = y;  
        }  
    }  
}
```

Lifetime parameters

- Each reference of type t has a lifetime parameter
 - $\&t$ (and $\&mut t$) – lifetime is implicit
 - $\&'a t$ (and $\&'a mut t$) – lifetime ' a is explicit
- Where do the lifetime names come from?
 - When left implicit, they are generated by the compiler
 - Global variables have lifetime ' $static$

I lifetime sono dei tipi denotabili vuol dire che io lo posso mettere anche come parametri, come annotazioni di tipo, ad esempio della segnatura delle funzioni allora se io scrivo in una segnatura di una funzione riferimento a un certo tipo t oppure un riferimento a un entità mutabile, sto dicendo che lifetime è implicito, sarà il compilatore a crearsi il suo lifetime esattamente nello stile che abbiamo visto in precedenza, però posso farlo esplicitamente, cioè posso vincolare nella definizione della segnatura, posso mettere delle nozioni di lifetime che mettono dei vincoli al tempo di vita delle variabili e dei parametri in modo particolare la notazione che verrà utilizzata è che l' $\&$ ' a t sta a dire che ho un riferimento di tipo t con un certo tipo di vita che è il tipo a . Come dicevo, quando sono lasciati impliciti, quindi quando uso questa notazione al compilatore che lo fa per quelle variabili globali, come esempio quei valori che abbiamo visto prima è la notazione statica

Lifetime: generic

Si può anche scrivere delle funzioni che hanno dei vincoli su lifetime. Allora andiamo a vedere come lo possiamo usare in modo generico. Stiamo definendo questa funzione longest che sostanzialmente prende due stringhe e restituisce quella che ha la lunghezza maggiore. Andiamo a vederle dal punto di vista non sono mutable, non facciamo alcune operazioni di modifica, dal punto di vista del lifetime diciamo che questa funzione longest è parametrica rispetto a lifetime 'a e quello che diciamo è che sia le stringhe che corrispondono ai parametri in ingresso che la stringa che otteniamo come risultato hanno lo stesso lifetime. Questa notazione ci dice che il tempo di vita della stringa risultato sarà lungo almeno quanto 'a.

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {  
    if x.len() > y.len() {x} else {y}  
}
```

x and **y** must have the same lifetime, and the returned reference shares it

The function signature also tells Rust that the string (slice) returned from the function will live at least as long as lifetime 'a.

```
fn main() {  
    let string1 = String::from("abcd");  
    let string2 = "xyz";  
  
    let result = longest(string1.as_str(), string2);  
    println!("The longest string is {}", result);  
}  
  
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {  
    if x.len() > y.len() { x } else { y }  
}
```

The lifetime parameters are not changing the lifetimes of any values passed in or returned.

The lifetime parameters specifies that the type system should reject any values that don't adhere to these constraints

Abbiamo questa longest definita come abbiamo visto prima e poi vedete abbiamo un main, definisce una string1 definisce una string2 come letterale e poi vedete quello che succede è che chiama longest su questi due valori e poi a questo punto dice la stringa più grande è quella che avrà come risultato. Allora a questo punto è che il life time che hanno string1 e string2 è un valore che viene trasmesso associato al valore della stringa, viene trasmessa al parametro e viene poi anche eventualmente restituito perché è quello che fa questa funzione, quindi questo vuol dire che se io trasmetto come parametro uno che ha un lifetime diverso, che non è compatibile con quello che si aspettava la segnatura della funzione, a questo punto ho un errore, esattamente come ho degli errori negli assegnamenti, questo vuol dire che la nozione di borrow era collegata al valore, che la nozione di lifetime è collegata al borrow e al valore e tutte queste informazioni sono controllate staticamente dal compilatore, infatti vedete che quando io vado a eseguire questo semplice programma, fa quello che ci aspettavamo e ci dice qual è il valore della stringa più lunga, che è abcd rispetto a xyz

RUN ►

...

DEBUG ▾

STABLE ▾

...

```
1 fn main() {
2     let string1 = String::from("abcd");
3     let string2 = "xyz";
4
5     let result = longest(string1.as_str(), string2);
6     println!("The longest string is {}", result);
7 }
8
9 fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
10    if x.len() > y.len() { x } else { y }
11 }
```

:::

Execution

Standard Error

```
Compiling playground v0.0.1 (/playground)
Finished dev [unoptimized + debuginfo] target(s) in 1.94s
Running `target/debug/playground`
```

Standard Output

The longest string is abcd

```
fn main() {  
    let string1 = String::from("abcd");  
    let string2 = "xyz";  
  
    let result = longest(string1.as_str(), string2);  
    println!("The longest string is {}", result);  
}  
  
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {  
    if x.len() > y.len() { x } else { y }  
}
```

allora è la notazione dice che il tipo del risultato, inteso in termini del lifetime associato al risultato, sarà come minimo un tempo di vita che quantomeno sarà maggiore del più piccolo del life time dei due parametri.

The generic lifetime '`'a` will get the concrete lifetime that is equal to the smaller of the lifetimes of the parameter `x` and `y`.

Because we've annotated the returned reference with the same lifetime parameter '`'a`, the returned reference will be valid for the length of the smaller of the lifetimes of `x` and `y`.

RUN ►

...

DEBUG ▾

STABLE ▾

...

```
1 fn main() {
2     let x = String::from("ciao");
3     let z;
4         { let y = String::from("ciao ciao");
5             | z = longest(&x,&y);
6         }
7     println!("z = {}",z);
8 }
9
10 fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
11     if x.len() > y.len() { x } else { y }
12 }
13
```

:::

Execution

Standard Error

```
Compiling playground v0.0.1 (/playground)
error[E0597]: `y` does not live long enough
--> src/main.rs:5:23
|
5     z = longest(&x,&y);
      ^`y` dropped here while still borrowed
6 }
7     - borrow later used here
```

```
fn main() {  
    let string1 = String::from("long string is long");  
  
    {  
        let string2 = String::from("xyz");  
        let result = longest(string1.as_str(), string2.as_str());  
        println!("The longest string is {}", result);  
    }  
}
```

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {  
    if x.len() > y.len() { x } else { y }  
}
```

**string1 is valid until the end of the outer scope,
string2 is valid until the end of the inner scope,
result references something that is valid until the end
of the inner scope.**

**The type checker approves this code; it will compile
and run.**

RUN ▶

DEBUG ▾

STABLE ▾

...

```
1 fn main() {
2     let string1 = String::from("long string is long");
3
4     {
5         let string2 = String::from("xyz");
6         let result = longest(string1.as_str(), string2.as_str());
7         println!("The longest string is {}", result);
8     }
9 }
10
11 fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
12     if x.len() > y.len() {
13         x
14     } else {
```

```
Compiling playground v0.0.1 (/playground)
Finished dev [unoptimized + debuginfo] target(s) in 1.02s
Running `target/debug/playground`
```

The longest string is long string is long

andiamo a vedere come è fatto questo altro esempio, vedete creiamo una stringa che è particolarmente lunga poi siamo all'interno di un altro scope e creiamo la stringa xyz, poi chiamiamo longest e prendiamo il risultato e poi stampiamo. Andiamo a vedere dal punto di vista del lifetime qual è la caratteristica, la string1 ha un lifetime che sostanzialmente è equivalente allo scope più esterno, la string2 invece ha un life time che è equivalente a quello interno. Risultato? E' qualcosa che abbiamo detto visto come come life time è almeno grande quanto quello più piccolo quindi ha un valore che va da quello interno a quello esterno, come tempo di vita. Il compilatore fa questi controlli e ti dice che queste cose sono tutte compatibili correttamente e perché a questo punto lo compila e infatti vedete, ci dice che la stringa più lunga è esattamente la stringa long string is long, quindi gli sta bene proprio perché i lifetime sono compatibili.

SUMMARY

C++ example

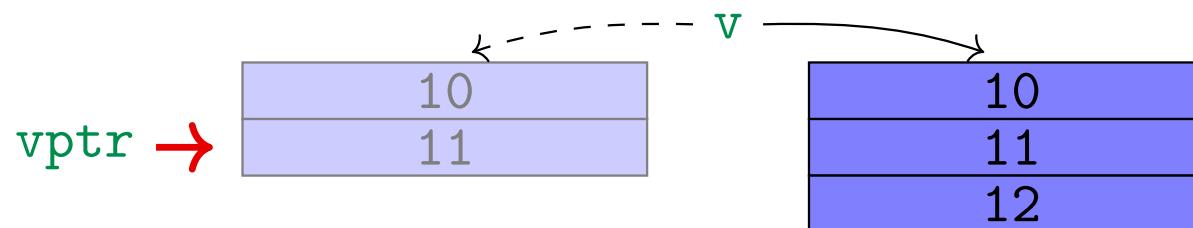
```
1 std::vector<int> v { 10, 11 };
2 int *vptr = &v[1]; // Points *into* 'v'.
3 v.push_back(12);
4 std::cout << *vptr; // Bug (use-after-free)
```

At line 3: The integer 12 is added after 11 in the buffer containing the elements of the vector.

If there is no more space for an additional element, a new buffer is allocated and all the existing elements are moved over

abbiamo poi la possibilità di andare a prendere il riferimento ad un elemento di un vector, in modo particolare diciamo che il pointer vptr è il puntatore che punta al primo elemento del vettore 1 quindi stiamo puntando all'interno del vettore. Notate che questo non è possibile in linguaggi come Java a meno che i valori del vector non siano degli oggetti e quindi caratterizzati da un riferimento, qui il valore di un vettore, in questo caso è la posizione uno è un intero che non ha un riferimento. Alla linea tre di codice andiamo a estendere il vector andando a mettere il valore 12. Alla posizione zero prima avevo 10, alla posizione zero, ora ho 10 alla posizione uno, prima avevo 11, ora ho 11, il vector finiva lì, dopo aver fatto la push ho anche una posizione tre e alla posizione tre ho il valore di 12 e poi a questo punto vado a fare un'operazione che, usa questo questo pointer in modo da fare un cout. La cosa interessante è che immaginate che il vector è memorizzato sullo heap in una determinata porzione, immaginato che è pieno, cioè non sono in grado di memorizzare un ulteriore elemento quando faccio l'operazione di resize. Vuol dire che a questo punto devo chiedere spazio per contenere il nuovo elemento quindi devo prendere i vecchi valori, spostarli e copiarli nella nuova posizione perché è più grande perché deve contenere un altro elemento.

Dangling pointer



Issue: adding a new element to the vector v has turned `vptr` into a dangling pointer: both pointers were aliasing

prima avevo il valore della `v` che puntava 10 11, il virtual pointer puntava 11 solo, ora quello che succede dopo avere inserito 12 è che ho copiato e a questo punto non ho più il riferimento alla `v` perché ho fatto un'operazione di resize e `v` punta 10 11 e 12. Allora l'effetto netto dell'operazione di resize per il fatto che la memoria era già occupata è che ho creato un dangling pointer perché virtual pointer a questo punto è dangling e questo è il motivo per cui nella linea quattro do un errore

Rust's solution

```
1 let mut v = vec![10, 11];
2 let vptr = &mut v[1]; // Points *into* 'v'.
3 v.push(12);
4 println!("{}", *vptr); // Compiler error
```

allora andiamo a vedere la soluzione di rust a questo stesso problema la soluzione di rust è creiamo un vector esattamente a parte la notazione sintattiche che è mutable, anche questo è un vector resizable creiamo un puntatore virtuale al primo elemento anche in questo caso è un entità mutabile, punto al primo elemento, esattamente quello simile, facciamo l'operazione di push, cambia un po di notazione e vedete che anche in questo caso da un compiler error non un errore a runtime che è esattamente quello che noi volevamo perché vedete quello che ho fatto ho preso questo esempio giocattolo, e l'ho eseguito e avendo fatto questa operazione di borrowing, il compilatore è in grado di andare a controllare e a questo punto abbiamo ceduto l'ownership ed è in questo caso quindi in grado di controllare che staticamente quello è un problema ma staticamente prima di mandarlo in esecuzione

RUN ►

...

DEBUG ▾

STABLE ▾

...

```
1
2 - fn main() {
3     let mut v = vec![10, 11];
4     let vptr = &mut v[1];
5     v.push(12);
6     println!("{}", *vptr); |
7 }
8
9
```

:::

Execution

Standard Error

```
Compiling playground v0.0.1 (/playground)
error[E0499]: cannot borrow `v` as mutable more than once at a time
--> src/main.rs:5:1
```

```
4 |     let vptr = &mut v[1];
|           - first mutable borrow occurs here
5 |     v.push(12);
|     ^ second mutable borrow occurs here
6 |     println!("{}", *vptr);
|           ----- first borrow later used here
```

Rust principle:

*a reference is never
both aliased and mutable at the same time.*

Rust: resource management

- When a variable holding owned memory goes out of scope the compiler automatically deallocates the memory at that point:
 - we know for sure that this memory will not be needed any more
 - the compiler transparently inserts *destructor* calls (via the drop method).
- Using destruc-tors for automatic resource management was pioneered in the form of RAII (Resource Acquisition Is Initialization) in C++
 - the key difference in Rust is that the type system can statically ensure that resources do not get used any more after they have been destructed.

a questo punto andiamo a vedere anche un'altra cosa della gestione della memoria e di tutte le risorse con rust. L'idea è che quando una variabile va fuori dallo scopo viene chiamata la drop e viene restituita la porzione di memoria che conteneva il valore allo heap. L'idea è che se vado al di fuori dello scope non ho alias, allora a quel punto non ho più altro cammino d'accesso, se vado fuori dallo scope vuol dire che non volevo più utilizzarlo con quel cammino d'accesso cioè lo restituisco allo heap. Allora il fatto di usare i distruttori, la drop non è una cosa che si sono inventati quelli che hanno progettato solo in rust ma era stata utilizzata già in c++ e veniva chiamata RAII, il passaggio in più che ha fatto lo sviluppo di Rust è che a questo punto tutto questi aspetti vengono gestiti staticamente dal compilatore da un sistema di tipo che tiene conto dell'ownership e del lifetime, questo è il passaggio, perché concettualmente è chiaro che l'idea che stava già dietro alla free, se io dico che quella variabile non la uso più allora a questo punto la posso liberare quella che il problema è che se nel frattempo ho avuto dell' alias, ho ceduto gli accessi, ho più camini di accesso al valore che avevano denotato è lì che ho casino, quindi il fatto di avere un sistema di tipi che affronta esattamente tra virgolette la confusione è la caratteristica innovativa di rust.

Borrowing

- The idea of *borrowing* takes a lot of inspiration from work on *region types*
- M. Tofte and J. Talpin. Region-based memory management. *Information and Computation*, 132(2), 1997.
- D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *PLDI*, 2002.

Borrowing and Lifetime

```
fn add_something(v: &mut Vec<i32>) {  
    v.push(11);  
}
```

```
fn main() {  
    let mut v = vec![10];  
    add_something(&mut v);  
    v.push(12);  
}
```

The compiler ensures that

- (a) the reference (`v`) only gets used during that lifetime
- (b) the referent does not get used again until the lifetime is over.

RUN ►

...

DEBUG ▾

STABLE ▾

...

```
1
2 fn add_something(v: &mut Vec<i32>) {
3     v.push(11);
4 }
5
6 fn main() {
7     let mut v = vec![10];
8     add_something(&mut v);
9     v.push(12);
10}
11
12
13
```

:::

Execution

Standard Error

```
Compiling playground v0.0.1 (/playground)
Finished dev [unoptimized + debuginfo] target(s) in 1.11s
Running `target/debug/playground`
```

To summarize:

whenever something is passed by value, Rust interprets this as ownership transfer; when something is passed by reference Rust interprets this as borrowing for a certain lifetime.

per riassumere quello che abbiamo fatto sul passaggio dei parametri, quando noi passiamo un valore rust lo interpreta come passaggio della ownership, quando viene passato per riferimento, lo intende come condivisione, ma solo per un certo lifetime che è caratterizzato dal tipo dei valori degli elementi passati come parametri.

Which is the owner of s at
HERE?

```
fn foo(s:String) -> usize {
    let x = s;
    let y = &x;
    let z = x;
    let w = &y;
    \\ HERE
}
```

- A. x
- B. y
- C. z
- D. w

Which is the owner of s at
HERE?

```
fn foo(s:String) -> usize {  
    let x = s;  
    let y = &x;  
    let z = x;  
    let w = &y;  
    \\ HERE  
}
```

- A. x
- B. y
- C. z
- D. w