# 02\_3 LIFETIME

Un valore inzia ad esistere all’interno del blocco in cui è definitito e, se non viene mosso, cessa di esistere insieme al suo blocco. Se di questo valore recuperiamo il riferimento, il riferimento potrà esistere e essere acceduto in vari modi solo nell’intervallo di tempo in cui il valore, a cui il riferimento si riferisce, esiste.

Se recupero il **riferimento di un valore e lo passo ad una funzione**:

* Riferimento ha una durata
  + Internamente il compilatore si segna i vincoli legati al tempo di vita
* Se funzione riceve 2+ riferimenti, occorre che le azioni svolte dalla funzione siano compatibili ai tempi di vita dei vari riferimenti.

Quando la funzione generica viene usata, il compilatore costruisce delle versioni specifiche per i tipi con cui la funzione è stata utilizzata 🡪 **MONOMORFIZAZIONE**.

**Nota**: gli indicatori del tempo di vita non partecipano alla monomorfizzazione, sono solo utili affinchè il borrow checker possa capire quali sono le **conseguenze dei puntatori** che stiamo passando e possa quindi verificare che non facciamo schifezze.

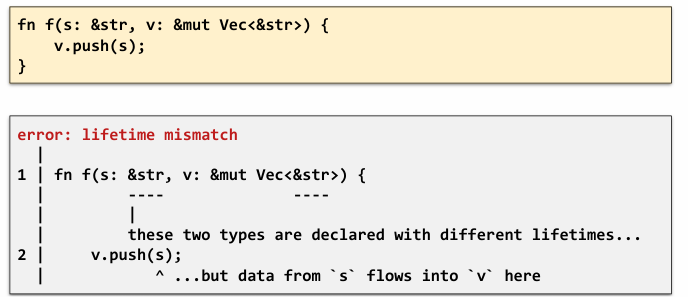
Se la **funzione ritorna un riferimento**, il compilatore non riesce a dedurre il tempo di vita. In questo caso è essenziale dire da dove è preso questo intero di cui è tornato il riferimento.

*In questo caso specifichiamo che il tempo di vita è legato al secodo parametro → così il compilatore comprende quanto potrebbe durare questo prestito.*

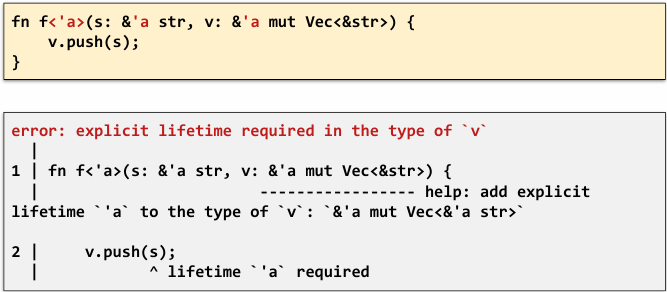
Se invece di distinguere a’ e b’, avessi a’ e a’ allora il tempo di vita sarebbe ridotto al minore dei due a’.

Se il compilatore si accorge che il risultato della funzione è usato, ci blocca.

Se la funzione memorizza il riferimento ricevuto in ingresso in una struttura dati, il compilatore deduce che il **tempo di vita della struttura in cui il riferimento è memorizzato deve essere incluso o coincidente con il tempo di vita del riferimento** → Se questo non avviene, il compilatore identifica l’errore e impedisce alla compilazione di avere successo

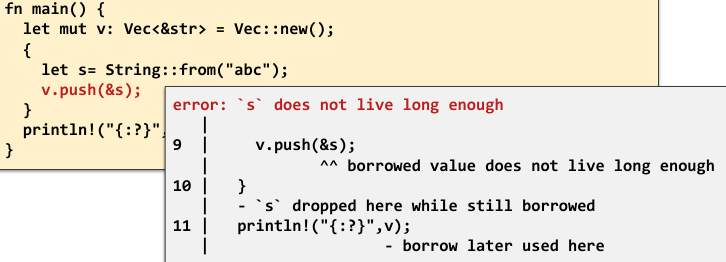
**Esempi**:

*Iil problema è che la stringa potrebbe essere non più valida e il vettore lo sia ancora → per evitare questo problema bisogna garantire che la stringa viva più del vettore.*



*Gli sto dicendo che il vettore esiste per il tempo di quella stringa ma il compilatore vuole garanzie che il contenuto del vettore sia anche valido per tutto quel tempo di vita.*

**Modo corretto:**

Ma **problema**:

*Il vettore dura più di s, non si può mettere s dentro.*

Nel caso in cui ci sono **due riferimenti in ingresso e uno solo in uscita**, **bisogna specificare** al compilatore a quale tempo di vita è legato il riferimento ritornato.

*Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente*

*Non posso accedere al risultato oltre la durata di v2.*

*Non posso modificare v2 fintanto che faccio accesso al risultato (penultima riga da cancellare altrimenti non compila)*

Lo **scopo degli identificatori** relativi al ciclo di vita è duplice:

* X codice che **invoca la funzione** → indicano su quale indirizzo è basato il risultato in uscita
* X codice interno alla **funzione** → garantiscono che vengano restituiti solo indirizzi a cui è lecito accedere per il tempo di vita indicato.

*Quando avviene l’invocazione, il compilatore verifica che gli identificatori che abbiamo passato esistano per almeno un tempo di vita minimo.*

**Immagine che contiene testo, Carattere, calligrafia, schermata

Descrizione generata automaticamenteStrutture dati**: se io salvo un riferimento con un tempo di vita all’interno di una struttura dati, questa struttura dati eredita il tempo di vita più breve tra quelli dei riferimenti che presenta al suo interno.

*Nota: nel caso in cui una struttura diventa parte di una struttura più grande, la più grande sarà limitata dal tempo di vita di quella inserita al suo interno.*

*Immagine che contiene testo, schermata, Carattere, linea

Descrizione generata automaticamente***Elisione dei tempi di vita**: se la funzione prende un solo riferimento in ingresso e un solo riferimento in uscita, il compilatore deduce in automatico il tempo di vita.  
*Se sono presenti più riferimenti e non si specifica il tempo di vita, il compilatore immagina che i tempi di vita siano indipendenti tra di loro →il programmatore dovrebbe specificare.*

***Per non avere problemi****:*

Se una funzione restituisce un riferimento o un tipo che contiene un riferimento, **bisogna specificare** il tempo di vita del riferimento.

Se una funzione prendere mut sel o self tramite riferimento, per il compilatore il tempo di vità è quello del riferimento a self (Se non specificato). *(Questo parte dal presupposto che, se un metodo di un oggetto restituisce un dato preso a prestito (borrow), questo sia stato preso dai dati posseduti dall’oggetto stesso)*

**Nota aggiuntiva**: esiste il **lifetime static** → riferimento che dura certamente quanto l’intero programma (es: costanti). Inoltre, se una variabile posside interamente il suo valore (non ha riferimenti) → quella variabile ha come tempo di vita static.

## CHIUSURE *(funzione lambda)*

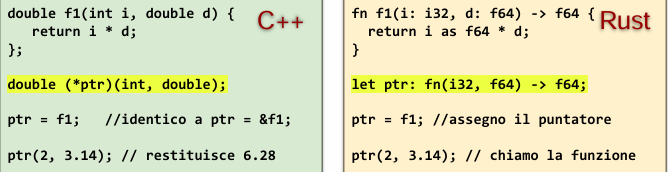
**Nota**: programmazione può essere di tipo imperativo (azioni hanno effetti collaterali che si propagano sulle nostre variabili) oppure funzionale (pone le radici in matematica → non ha effetti collaterali).

Una funzione mangia dei dati e produce un risultato:

* Operano su valori
* Operano su altre funzioni
  + Per fare ciò linguaggio deve permettere di salvare all’interno delle variabili una funzione.

Sui dati si possono fare diverse operazioni, mentre le funzioni posso semplicemente chiamarle.

**Let f= alpha** 🡪 f diventa un puntatore alla funzione alpha**, f è diventato un alias di alpha**; *volendo si può fare in modo che f possa assumere alpha o beta in base ad un evento → decide cosa salvare dentro f e indipendentemente chiamo f (alpha e beta devono avere lo stesso numero e tipo di parametri e ritorno).*

Ad una variabile, oltre che le funzioni**, posso asegnare dei tratti** che danno all’oggetto che gli implementa il comportamento di una funzione (FnOnce, FnMut, Fn)

A questo punto chiamare ptr o f1 è la stessa cosa

Posso assegnare un valore differente in base ad una condizione presente nel codice → a questo punto chiamo sempre la mia variabile e chiama la funzione desiderata.

#### **C++**

Immagine che contiene testo, Carattere, schermata, numero

Descrizione generata automaticamenteNota sul c++: quando introduco una classe, posso inserire tra i suoi metodi anche operator(). Creo una cosa che uso come fosse una funziona senza che sia una funzione.

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente All’interno della mia classe posso definire più operator per diversi tipi.

Siccome è una classe, può avere delle **variabili membro** → possono essere usate per ricordare il comportamento della mia funzione → posso trasformare la mia funzione in impura: funzione con uno stato, rciorda la sua storia. Es: Accumulatore.

**Nota**: se creo due oggetti di tipo accumulatore questi saranno distinti tra loro → posso avere più storie in contemporanea cosa che non sarebbe possibile usando variabili globali.

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamentePosso anche usare il mio stato per cambiare il valore di ritorno.

Le funzioni lambda non hanno un nome, lo prendono dalla variabile a cui sono assegnate.

### Funzioni lambda

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamenteUna volta definita ed assegnata ad una variabile, può essere invocata trattao la variabile come fosse una funzione. Inoltre, una funzione lambda può essere passaa come argomento ad una funzione o ritornata da una funzione.

Nel corpo di questa funzione, possono essere citate delle variabili che siano visibili nel punto in cui la definizione avviene → congeliamo nel corpo della nostra funzione dei valori che prendiamo attorno a noi (nel punto in cui siao stati definiti) → la lambda acquisisce questi valori e rimangono con lei. 🡪 VARIABILI LIBERE

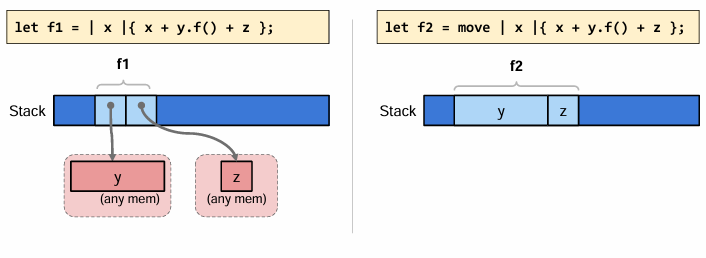
In Rust il compilatore **trasforma una chiusura in una tupla**

* Avente tanti campi quante sono le variabili libere
* Tupla che implementa uno dei tratti funzioni previsti da Rust: FnOnce, FnMut, Fn

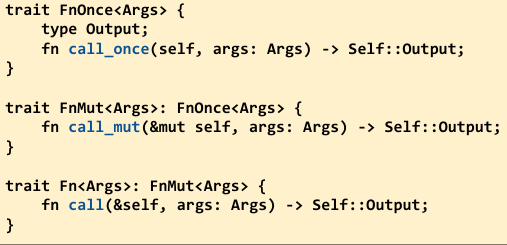
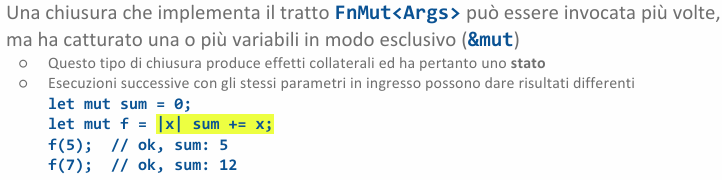
--codice rust

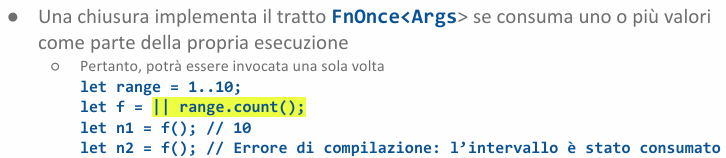
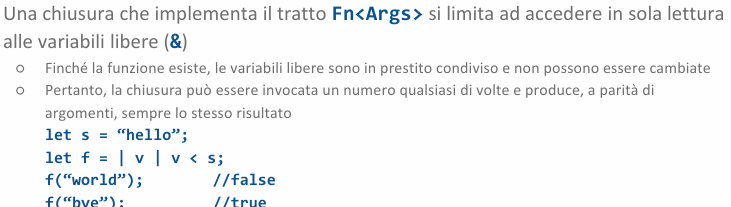
### **Cattura delle variabili in Rust:**

Di default viene inserito il riferimento semplice al dato, nel caso in cui si voglia che il dato originale abbia un tempo di vita diverso, bisogna governare in un’altra maniera → se si aggiugne la parola chiave **move**, invece di avere il reference si prende proprio la variabile → si **acquisisce il possesso alla variabile**.

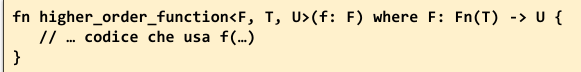


Esistono **3 tratti FUNZIONALI**: *(quelli più sotto hanno bisogno dell’implementazione di quelli più sopra)*

* **Fn once** → quando la chiusura verrà chiamata distruggerà un suo pezzo; dunque, è chiamabile una volta sola
  + Se consuma 1o+ valori nella sua esecuzione
  + Es: se chiamo self, prende possesso e restituisce altro
* **Fn mut** → ha la possibilità di essere chiamato più volte e ha la possibilità di mdoificare se stesso
  + Cattura in modo esclusivo 1+ variabili
  + Es: metodo che prende &mut self
* **Fn** → può essere chiamate infinite volte senza effetti particolari
  + Accedo in sola vettura alle variabili libere o se ne prendo possesso
    - *Si guarda ma non si modifica (come se si fa &self)*



**Funzioni di ordine superiore**

Posso implementare funzioni che accettino come parametro altre funzioni. Posso passare oggetti T che implementa il tratto Fn.

* Le funzioni lambda implementano almeno un tratto funzionale:

--codice

Slide non fatte:

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente

Immagine che contiene testo, schermata, Carattere, design

Descrizione generata automaticamente