POSSESSO

In Rust, ogni valore è posseduto da una ed una sola variabile; in caso di violazione -> non compila

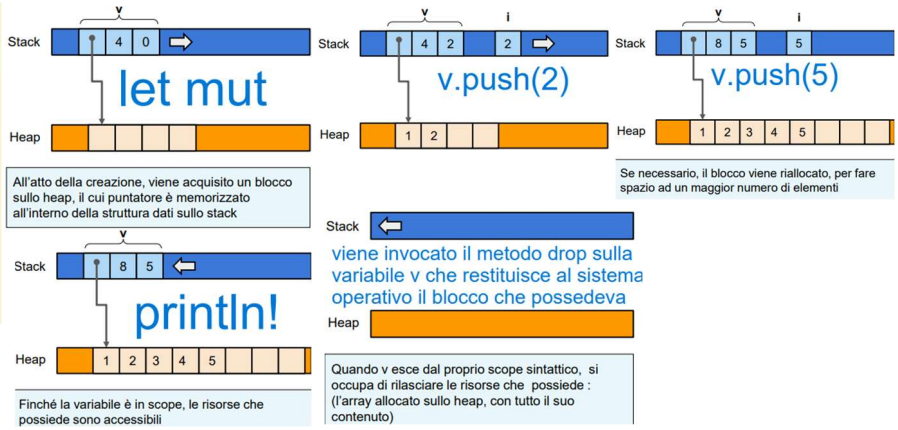
Possedere un valore → essere i responsabili del rilascio

- Liberare memoria
- Se il valore contiene una risorsa (puntatore a memoria dinamica, socket,..)→ devo liberare anche questa

RILASCIO:
Avviene quando una variabile esce dal suo scope sintattico;

MOVIMENTO1
Se il valore che possedeva la mia variabile, l'ho dato a qualcun altro, la situazione è diversa. (se in rust faccio v2=v1 → il valore è ora posseduto da v2 e sarà v2 a doversi occupare di tutto → MOVIMENTO)
Stessa cosa accede quando passo un valore ad un parametro di una funzione → il parametro diventa il nuovo possessore e la variabile passata ha perso diritto di accedere e dovere di rilascio.

```
fn main() {
  let mut v = Vec::with_capacity(4);
                // v possiede il vec
  for i in 1..=5 {
    v.push(i);
  }
  println!("{:?}",v);
}
```

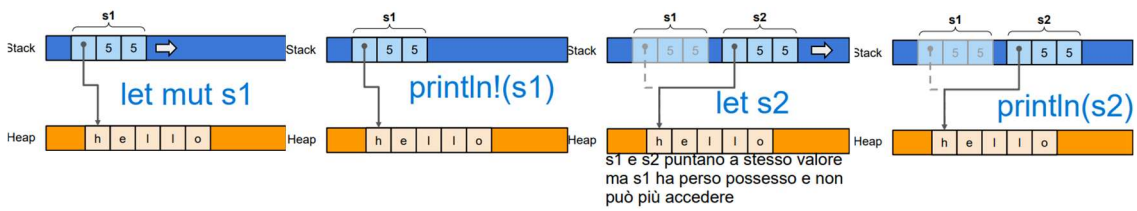


MOVIMENTO2
Dunque, il possesso di una variabile inizia alla sua inizializzazione.
Se ad una variabile mutabile assegno un nuovo valore, prima di accettarlo, la variabile rilascia il vecchio valore.
Se:

- Una variabile viene assegnata ad un'altra variabile
- Una variabile è passata come argomento di una funzione
- Il suo contenuto e il possesso viene MOSSO nella destinazione
  - La variabile originale non ha più possesso e non dovrà fare nulla quando uscirà dallo scope
  - Non si possono fare accessi in lettura alla variabile originale → non compila
  - Si può scrivere sulla variabile originale (se mut) → le sto assegnando un nuovo valore → variabile possiede un nuovo valore → si riabilita la lettura su questa variabile

La variabile destinazione contiene una copia bit a bit del valore originale (sullo stack), la parte sullo heap resta dov'è, cambia solo chi ha il puntatore.

```
let mut s1 = "hello".to_string();
println!("s1: {}", s1);
let s2 = s1;
println!("s2: {}", s2);

//s1 non è più accessibile
```



***NOTA**: Nel momento in cui dovessi copiare il mio v in un v1, il possesso si trasferisce.*

COPIA → tratto copy

Alcuni tipi sono definiti copiabili (es: numeri) → implementano il tratto **Copy**:

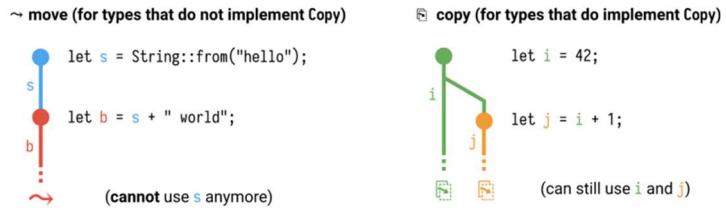
- Quando avviene un'assegnazione o un passaggio di parametro, viene assegnata una **copia bit a bit ma non perdo il diritto** di andare a guardare dentro perché in realtà il tratto copy mi garantisce che non avevo doveri *(NOTA: se ho il tratto copy non ho il tratto drop → variabile non ha doveri)*
- Possibile quando il valore contenuto non costituisce una risorsa che richiede rilascio o altro (a differenza dei file descriptor o altre informazioni importanti)

Tipi semplici e le loro combinazione (tuple, array di tipi semplici) sono copiabili

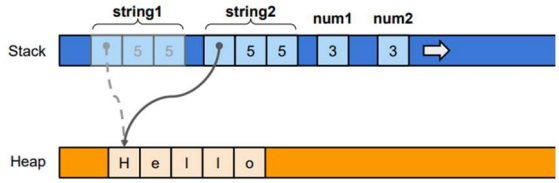
→ **NOTA**: riferimenti a valori non mutabili sono copiabili

→ **NOTA**: riferimenti a valori mutabili non sono copiabili

NOTA *(12:10):*




```
let string1 = "Hello".to_string();
let string2 = string1; //da qui in poi, string1 è inaccessibile in lettura
                       //a meno che non venga riassegnato
let num1: i32 = 3;
let num2 = num1;       //nessun vincolo su num1!
```



CLONAZIONE

I metodi che implementano il tratto Clone possono essere duplicati invocando il metodo **clone():**

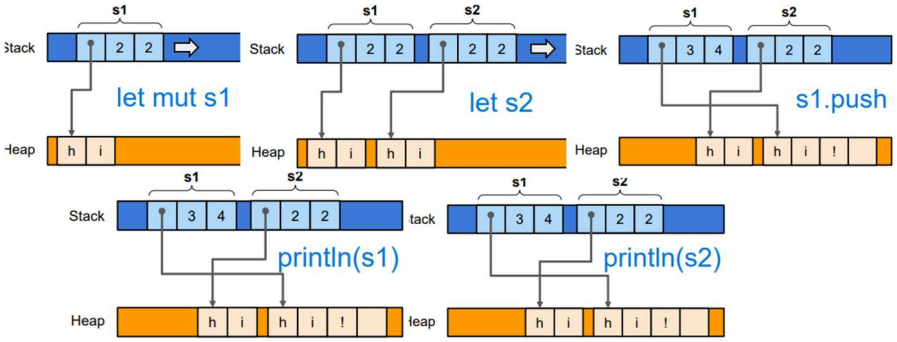
- **Copia in profondità** dei valori
  - Sia stack che heap
- Modifiche sui cloni, non impattano l'originale

Il programmatore può **modificare l'implementazione** della clonazione (a differenza di quanto avveniva per assegnazione e copia).

NOTA: un tipo che ha copy, ha sicuro anche clone (non viceversa)
NOTA: copy no insieme a drop
NOTA: clone può andare insieme a drop

```
let mut s1 = "hi".to_string();
let s2 = s1.clone();
s1.push('!');
println!("s1: {}", s1); //hi!
println!("s2: {}", s2);      //hi
```

(size e capacity di s1 sono invertiti in questo esempio per errore)



NOTA: rust clona solo su richiesta esplicita

RIFERIMENTI

Accedere temporaneamente in lettura/Scrittura (se riferimentom mutabile) ad un dato che non sto possedendo → **prestito con restituzione** quando finisco di usarla; non ho alcun obbligo.

Borrow checker → controllore dei riferimenti.

NOTA: vincoli → il riferimento può esistere solo finchè la variabile a cui faccio riferimento, esista ancora; se provo ad accedere ad un riferimento ma la avriabile originale non eisste → mi stoppa.

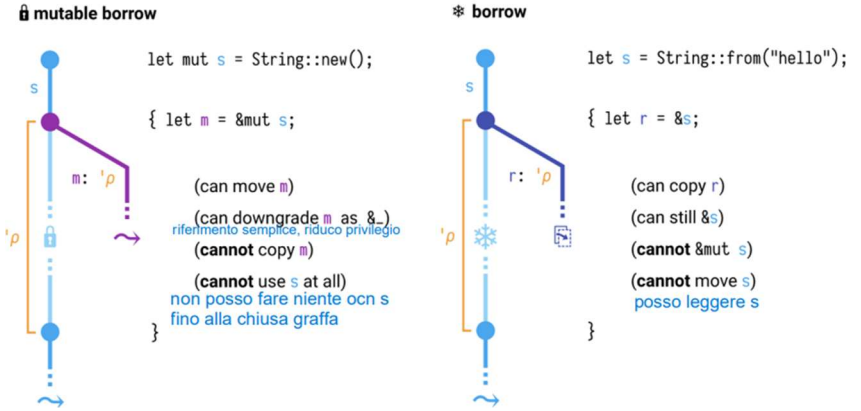
```
let point = (1.0, 0.0);      //point possiede il valore

let reference = &point;      //reference PUÒ accedere al valore in lettura
                             //finché esiste point
println!("({}, {})", reference.0, reference.1);
```

il compilatore aggiunge * in automatico se uso notazione .

NOTA: finché il riferimento è accessibile, non posso modificare il valore → possiamo leggere entrambi ma nessuno può cambiarla.
NOTA: è possibile creare ulteriori riferimenti a partire dal dato originale o altri riferimenti ad esso.
→ prestito resta in essere finché esiste almeno un riferimento (compilatore controllo quanti riferimenti ci sono)

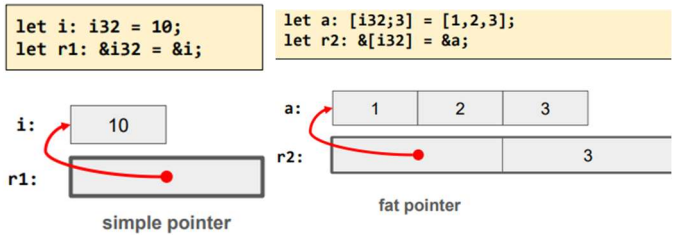
Se la variabile è mutabile, posso creare un **riferimento mutabile** → il riferimento mutabile esiste **in modo univoco** → mentre c'è un riferimento mutabile ad una variabile:

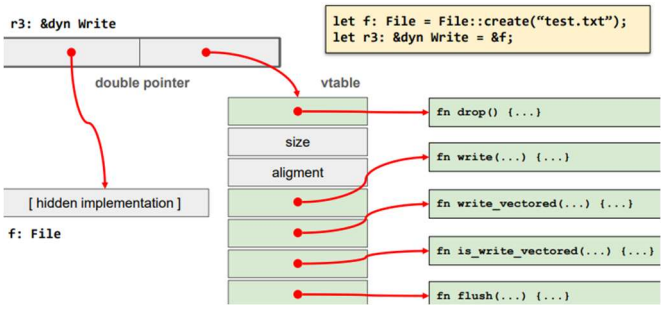
- Non ci possono essere altri riferimenti di alcun tipo
- La variabile originale non è accessibile e/o modificabile finché esiste il riferimento



Dimensione Riferimenti

- Puntatori **semplici** *(u size -64bit)*
- Puntatore + dimensione (**fat pointer**) *(2u size –128 bit)*
  - Pta ad inizio slice
  - Dimensione slice
- Puntatore **doppio** → per dati di tipi dinamici
  - Puntatore
  - **Puntatore alla Vtable per il tratto che considero**: size-alignment-metodi di quel tratto+drop





I riferimenti hanno un **tempo di vita**: borrow checker fa in modo che gli accessi al riferimento avvengano solo nell'intervallo di tempo in cui il dato esiste. → *impedire dangling pointer*

Tempo di vita*: insieme delle righe in cui si fa accesso al riferimento*

- Informazione mantenuta, dal compilatore, insieme alle informazioni che descrivono il tipo del riferimento
- Può essere espresso scrivendo **& ' a NomeTipo**
  - do un nome al tempo di vita
  - se scrivo & ' static NomeTipo → esiste dall'inizio alla fine

Una stringa espressa in formato letterale ("some string") ha come tipo `&'static str`, in quanto la sequenza di caratteri viene allocata dal compilatore nella sezione delle costanti e non viene mai rilasciata

Il compilatore verifica che il riferimento esiste soltanto nell'intervallo di tempo in cui esiste il valore da cui è stato ottenuto (se l'originale non c'è più, non ci deve essere neanche il riferimento).

Se in una struttura inserisco un dato con un tempo di vita limitato, la struttura eredita il tempo di vita più breve.

```
{
  let r;
  {
    let x = 1;
    r = &x;
  }
  assert_eq!(*r, 1);
}
```

Chiudendo la graffa, x cessa di esistere. Dunque, r punta nel vuoto.
Facendo assert_eq, il compilatore mi blocca e mi avvisa che X non vive abbastanza.

- Le regole, ovviamente, valgono anche quando si crea un riferimento ad una parte di una struttura dati più grande
  - L'esistenza in vita del riferimento deve essere inclusa in quella della struttura a cui punta
  - `let v = vec![1, 2, 3];`
  - `let r = &v[1];` `// v deve durare più a lungo di r`
- Se, al contrario si memorizzano dei riferimenti in una struttura, tutti questi riferimenti devono avere una durata di vita maggiore della struttura dati in cui sono memorizzati

```
{
  let mut v = Vec::new();
  {
      let a = 1;
      v.push(&a);
  }
  println!("{:?}", v);
}
```

```
error[E0597]: `a` does not live long enough
 --> src/main.rs:5:14
  |
5 |       v.push(&a);
  |              ^^ borrowed value does not live long enough
6 |   }
  |   - `a` dropped here while still borrowed
7 |   println!("{:?}", v);
  |                    - borrow later used here
```

Possesso -riassunto regole:

- Ciascun valore ha un unico possessore (variabile o campo di una struttura)
  - Il valore viene rilasciato (Drop) quando il processo re esce dal proprio scope o quando al possessore viene assegnato un nuovo valore
- Può esistere un singolo riferimento mutabile ad un dato valore
- Possono esistere molti riferimenti immutabili al medesimo valore
  - Finché ne esiste uno, il valore può essere mutato
- Tutti i riferimenti devono avere una durata di vita inferiore a quella del valore a cui fanno riferimento

NOTA*: se prendo un puntatore mutabile all'elemento 3 del vettore, non potrò accedere all'intero vettore*

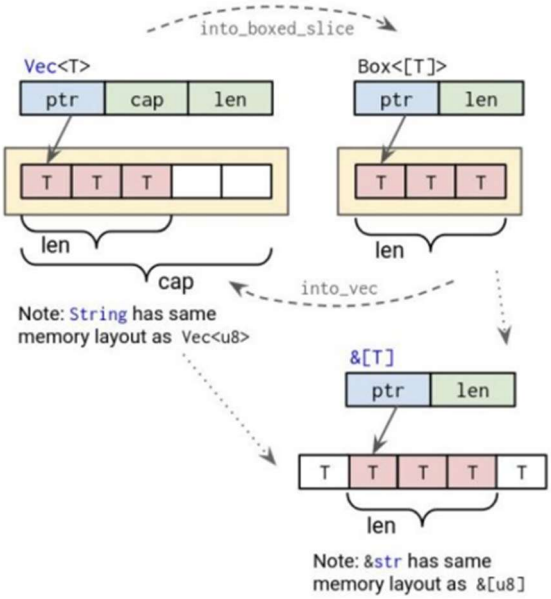
SLICE: vista di una sequenza contigua di elementi

- **lunghezza non nota** in fase di compilazione
- può leggere ma **non può scrivere**.
- Slice rappresentate da **fat pointer** (rif al primo, nr elementi)
- Slice **non possiede** il dato

Posso creare slice partendo da array o altri tipo di contenitori

NOTA: gli mstr sono delle slice.

```
let a: [i32; 5] = [1, 2, 3, 4, 5]; // a è un array di 5 interi
let s = &a[1..3];                  // s è una slice formata da 2 elementi [2,3]
let two = s[0];                                  // two contiene il valore 2
```



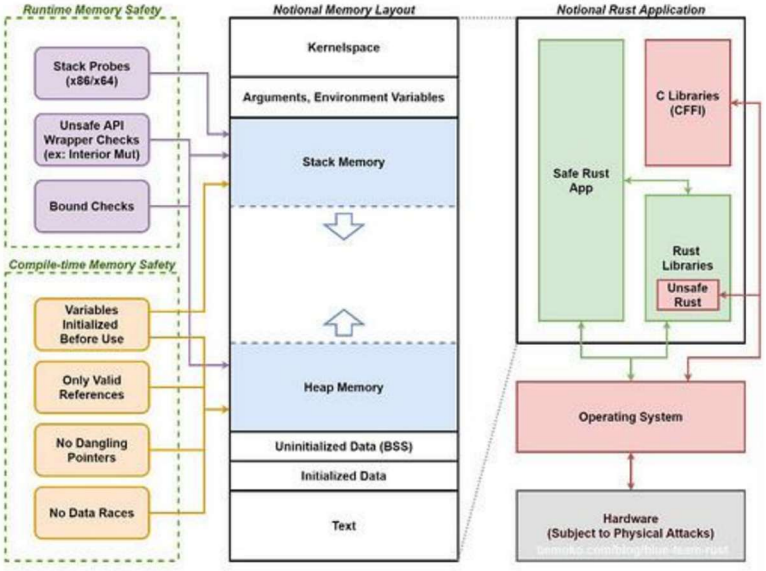
Creo un box con gli elementi contenuti in questo momento dal Vec

Creo un Vec a partire dal Box; capacity=size.

Una string non è altro che un vec di byte con più metodi.

Vantaggi introdotti dal concetto di possesso:

- **Non** esiste il concetto di riferimento nullo
- **Non** c'è il rischio di segmentazione o accesso illegale ad aree ristrette di memoria, né dangling pointer
- **Non** possono verificarsi buffer overflow o buffer underflow perché il borrow checker vede dimensione blocco
    - Iteratori offerti da Rust non eccedono mai i loro limiti
- Tutte le **variabili** sono **immutabili** per **default** e occorre dichiarazione esplicita per renderle mutabili
- Gestione di risorsa e memoria sono sotto il possesso



- *Assenza di un garbage collector impedisce comportamenti non deterministici*