

# Possesso

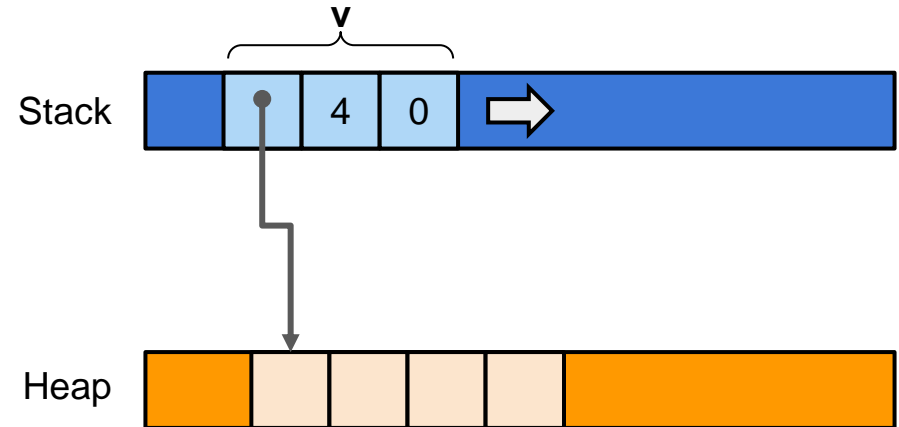
2023-24

# Possesso

- In Rust, ogni valore introdotto nel programma è **posseduto** da una ed una sola variabile
  - Un particolare blocco logico contenuto nel compilatore, detto **borrow checker**, verifica formalmente questo fatto, per ogni punto dell'esecuzione del programma
  - Ogni violazione porta ad un **fallimento della compilazione**
- Possedere un valore significa essere responsabili del suo **rilascio**
  - Se il valore contiene una risorsa (puntatore ad memoria dinamica, handle di file o socket, ...), questa deve essere liberata
  - Dopodiché occorre restituire al sistema operativo la memoria in cui il valore è memorizzato
- Il rilascio avviene quando la variabile che lo possiede **esce** dal proprio *scope* sintattico o quando le viene assegnato un nuovo valore
  - Rust offre un meccanismo (drop) mediante il quale è possibile associare azioni arbitrarie da eseguire prima che sia liberata la memoria
  - Il rilascio può essere rimandato a dopo, se il contenuto della variabile viene trasferito (mosso) in un'altra variabile: in questo caso la nuova variabile diventa responsabile del suo rilascio

# Possesso e rilascio

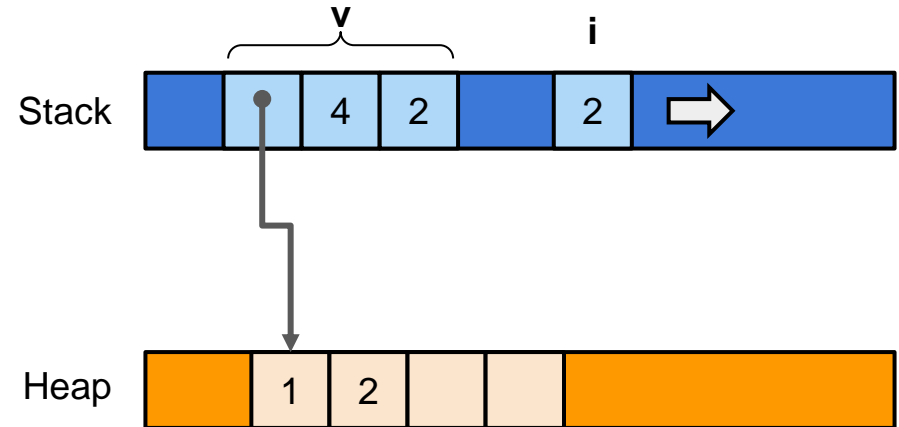
```
fn main() {  
    let mut v = Vec::with_capacity(4);  
    // v possiede il vec  
  
    for i in 1..=5 {  
        v.push(i);  
    }  
  
    println!("{:?}",v);  
}
```



All'atto della creazione, viene acquisito un blocco sullo heap, il cui puntatore è memorizzato all'interno della struttura dati sullo stack

# Possesso e rilascio

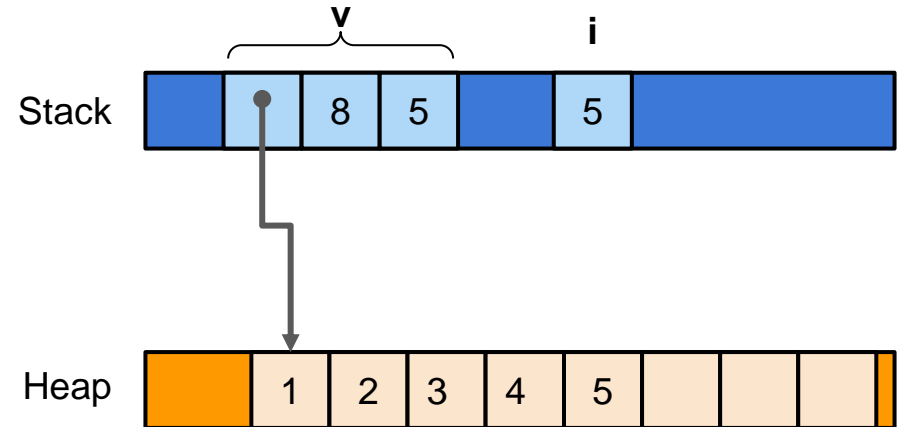
```
fn main() {  
    let mut v = Vec::with_capacity(4);  
    // v possiede il vec  
  
    for i in 1..=5 {  
        v.push(i);  
    }  
  
    println!("{:?}",v);  
}
```



All'atto della creazione, viene acquisito un blocco sullo heap, il cui puntatore è memorizzato all'interno della struttura dati sullo stack

# Possesso e rilascio

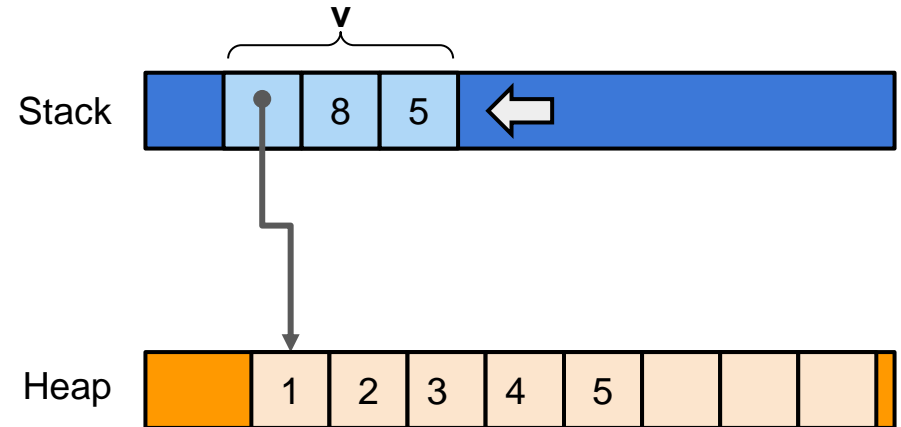
```
fn main() {  
    let mut v = Vec::with_capacity(4);  
    // v possiede il vec  
  
    for i in 1..=5 {  
        v.push(i);  
    }  
  
    println!("{:?}", v);  
}
```



Se necessario, il blocco viene riallocato, per fare spazio ad un maggior numero di elementi

# Possesso e rilascio

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



Finché la variabile è in scope, le risorse che possiede sono accessibili

# Possesso e rilascio

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



Quando `v` esce dal proprio scope sintattico, si occupa di rilasciare le risorse che possiede :  
(l'array allocato sullo heap, con tutto il suo contenuto)

# Movimento

- Quando una variabile viene inizializzata, prende possesso del relativo valore
  - Se ad una variabile (mutabile) è assegnato un nuovo valore, quello precedentemente posseduto viene rilasciato e la variabile diventa proprietaria del nuovo valore
- Se una variabile viene assegnata ad un'altra variabile oppure passata come argomento ad una funzione, il suo contenuto viene **MOSSO** nella destinazione
  - La variabile originale cessa di possedere il valore (non ne è più responsabile) ed il possesso passa alla variabile destinazione (o al parametro della funzione invocata)
  - La variabile originale resta allocata fino a quando non termina la sua visibilità (chiusura del blocco in cui è stata definita)
  - Eventuali accessi in **lettura** alla variabile originale porteranno ad **errori di compilazione**
  - Eventuali accessi in **scrittura** alla variabile originale **avranno successo** e ne riabiliteranno la lettura
  - La variabile destinazione conterrà una **copia** bit a bit del valore originale (ammesso che il compilatore non riesca a riusare i dati originali al loro posto)



# Movimento

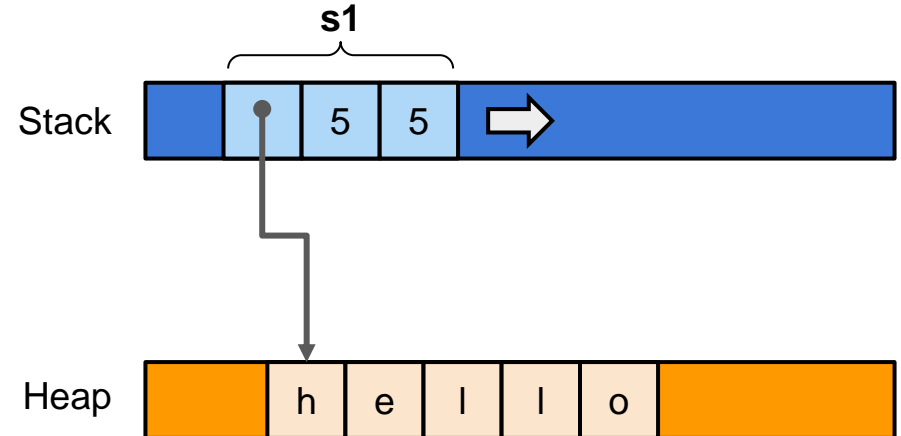
```
let mut s1 = "hello".to_string();
```

```
println!("s1: {}", s1);
```

```
let s2 = s1;
```

```
println!("s2: {}", s2);
```

```
//s1 non è più accessibile
```



# Movimento

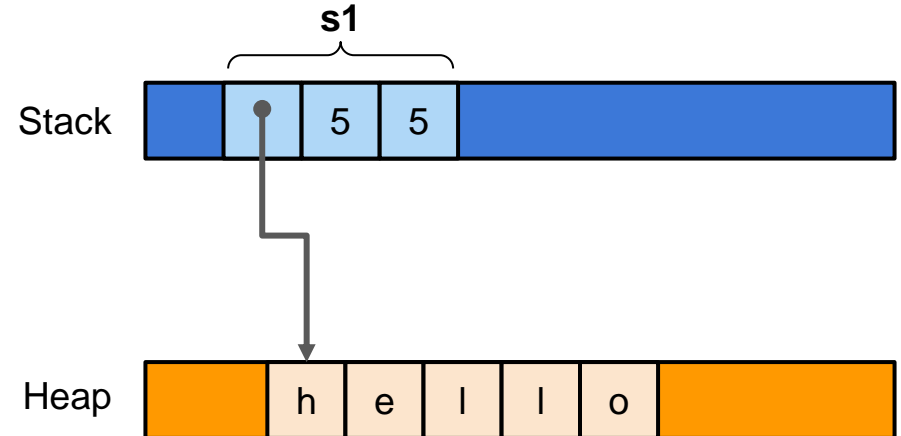
```
let mut s1 = "hello".to_string();
```

```
println!("s1: {}", s1);
```

```
let s2 = s1;
```

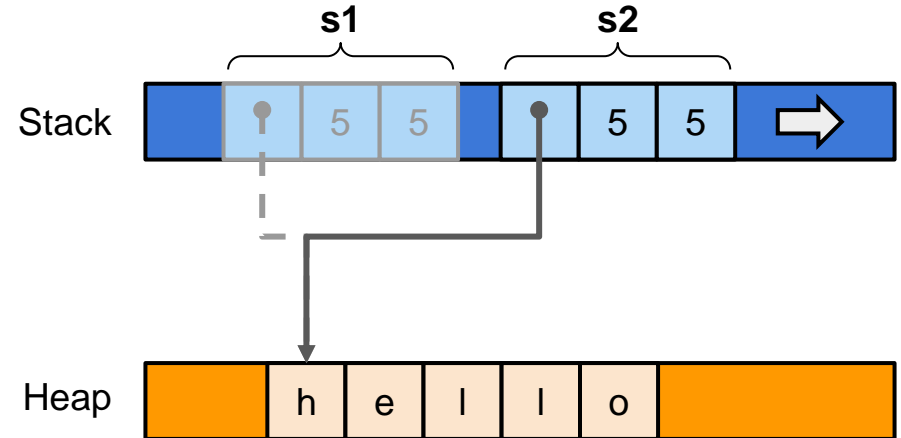
```
println!("s2: {}", s2);
```

```
//s1 non è più accessibile
```



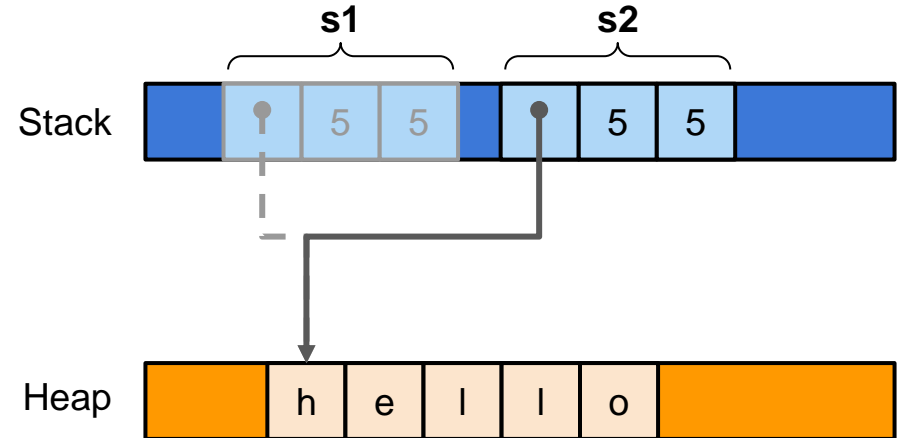
# Movimento

```
let mut s1 = "hello".to_string();  
println!("s1: {}", s1);  
let s2 = s1;  
println!("s2: {}", s2);  
  
//s1 non è più accessibile
```



# Movimento

```
let mut s1 = "hello".to_string();  
println!("s1: {}", s1);  
let s2 = s1;  
println!("s2: {}", s2);  
  
//s1 non è più accessibile
```



# Movimento

```
let mut s1 = "hello".to_string();  
println!("s1: {}", s1);           // s1: hello  
  
let s2 = s1;  
println!("s2: {}", s2);           // s2: hello, in s1  
c'è la stessa cosa:  
                                   // ma NON è più accessibile  
  
// s1 = "world".to_string();       // de-commentando  
questa riga, s1 torna  
                                   //  
a possedere un valore  
  
println!("s1.to_uppercase(): {}", s1.to_uppercase());  
  
                                   // NON possibile senza de-commentare  
                                   // la riga precedente
```

# Movimento

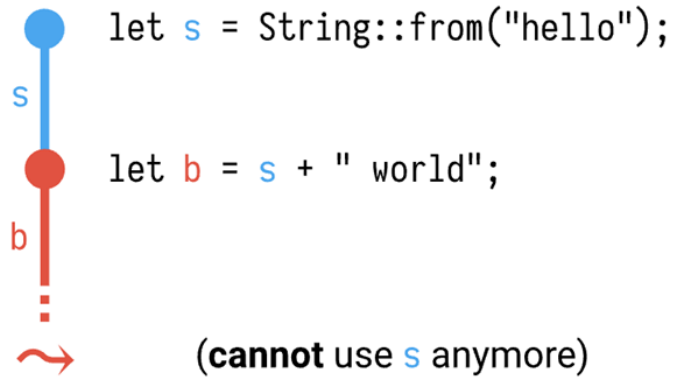
```
18 | let mut s1 = "hello".to_string();
   |          ----- move occurs because `s1` has type `String`, which does
not implement the `Copy` trait
19 |     println!("s1: {}", s1);
20 |     let s2 = s1;
   |             -- value moved here
...
23 |     println!("s1.to_uppercase(): {}", s1.to_uppercase());
   |                                     ^^^^^^^^^^^^^^^^^^^^^ value
borrowed here after move
```

# Copia

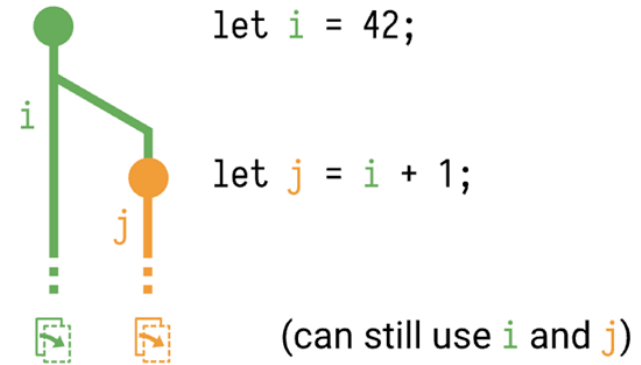
- Alcuni tipi, tra cui quelli numerici, sono definiti **copiabili**
  - Implementano il tratto **Copy**
  - Quando un valore viene assegnato ad un'altra variabile o usato come argomento in una chiamata a funzione, il valore originale rimane accessibile in lettura
  - Questo è possibile quando il valore contenuto non costituisce una “risorsa” che richiede ulteriori azioni di rilascio
- I tipi semplici e le loro combinazioni (tuple e array di numeri, ad esempio) sono copiabili
  - Così come sono copiabili i riferimenti a valori non mutabili
  - I riferimenti a valori mutabili NON sono copiabili
- Da un punto di vista del codice generato, **non cambia nulla** tra copia e movimento
  - L'istruzione di assegnazione o il passaggio come argomento comporta la duplicazione (bit a bit) del valore originale
  - Semplicemente, in caso di copia, il borrow checker non impedisce l'ulteriore accesso in lettura al dato originale

# Copia e movimento

→ **move** (for types that do not implement Copy)



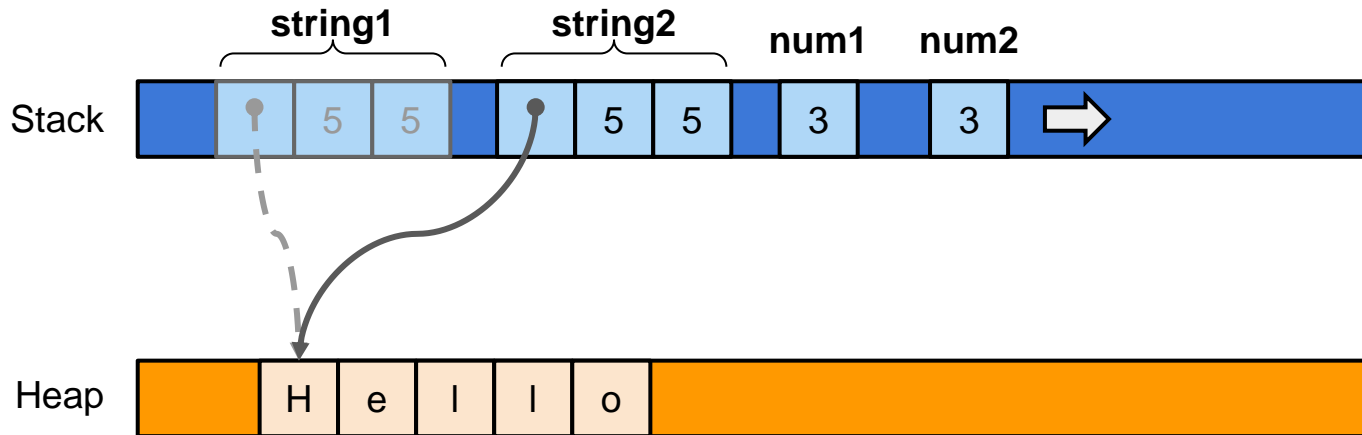
 **copy** (for types that do implement Copy)





# Copia e movimento

```
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!
```

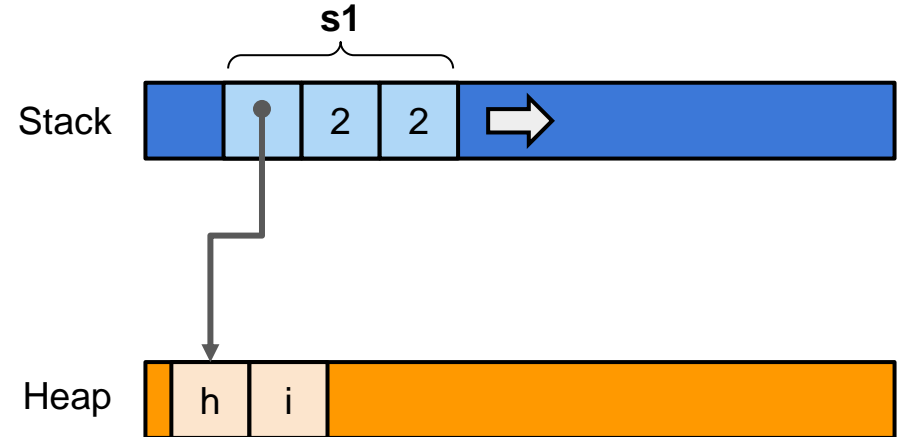


# Clonazione

- I tipi che implementano il tratto **Clone** possono essere duplicati invocando il metodo **clone()**
  - A differenza della copia e del movimento, la clonazione può comportare una **copia in profondità** dei valori
  - Di conseguenza, il costo della clonazione può essere elevato
- L'implementazione dell'operazione di clonazione è modificabile dal programmatore
  - L'implementazione di copia e movimento, invece, è sotto il controllo esclusivo del compilatore ed è basata sull'invocazione della funzione **memcpy(...)**
- Affinché un tipo possa implementare il tratto **Copy**, occorre che implementi **anche** il tratto **Clone**
  - Tuttavia, non occorre che le due implementazioni coincidano

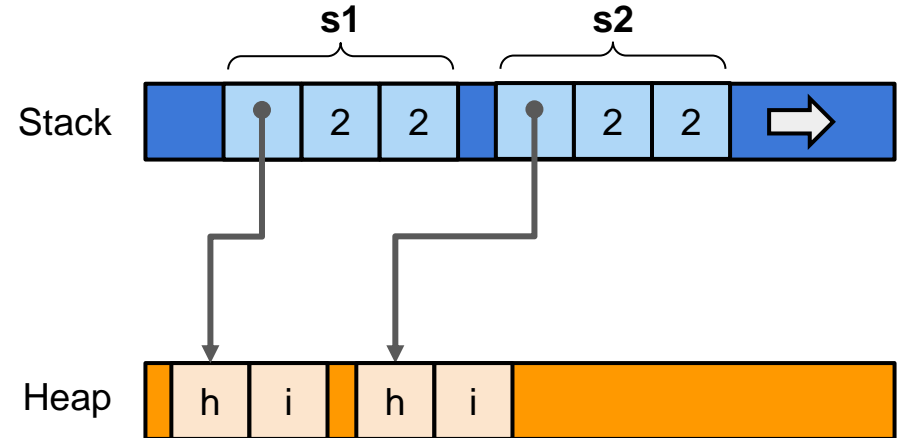
# Clonazione

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



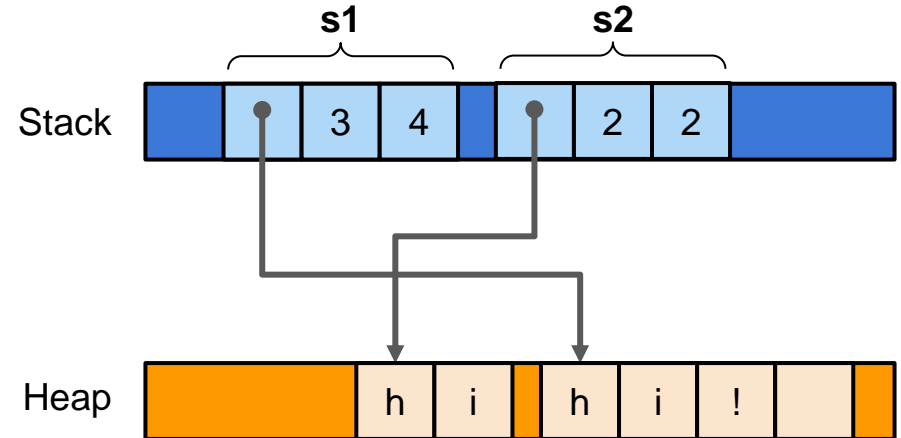
# Clonazione

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



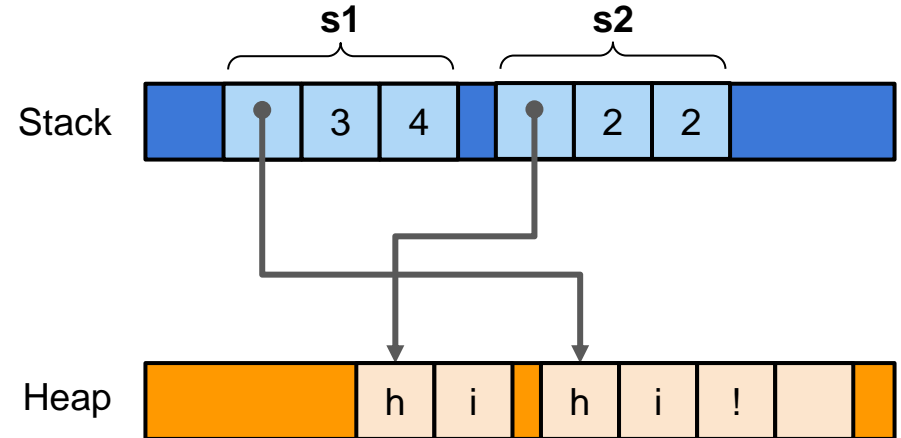
# Clonazione

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



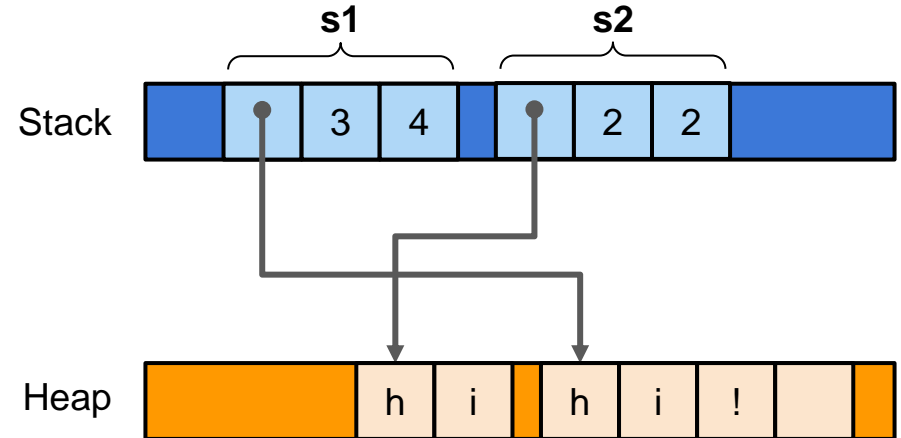
# Clonazione

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



# Clonazione

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



# Confronto con C e C++

- A differenza di quanto avviene in C e C++, in Rust il comportamento base adottato dal compilatore a fronte dell'assegnazione di una variabile o del passaggio di un argomento ad una funzione, è quello del **movimento**
  - Solo se il tipo risulta copiabile viene eseguita una copia
- In C, l'unico paradigma possibile è quello della **copia**
  - In C++ è possibile adottare il movimento a patto di invocarlo esplicitamente e che questo comportamento sia stato definito per lo specifico tipo di dato in oggetto
- Poiché in C++ il movimento è possibile, ma non c'è l'equivalente del Borrow Checker, è responsabilità del programmatore, in caso di movimento, lasciare l'oggetto di cui si è preso possesso del contenuto in uno stato coerente:
  - Eventuali accessi ai suoi campi non devono originare errori né visibilità del contenuto pregresso
  - La sua distruzione non deve originare errori né portare a fenomeni di doppio rilascio
  - Questo mette molta responsabilità nelle mani del programmatore che deve fornire implementazioni opportune per copia e movimento, distinguendo anche tra costruzione iniziale e riassegnazione!

Vedi anche: <https://radekvit.medium.com/move-semantics-in-c-and-rust-the-case-for-destructive-moves-d816891c354b>



# Riferimenti

- Per risolvere i problemi di accesso, Rust introduce diverse forme di puntatori, volte a rendere espliciti responsabilità e diritti di chi le maneggia
  - Il borrow checker si occupa di garantire che il codice complessivamente scritto sia conforme a tali regole e impedisce la compilazione in caso contrario
- Un **riferimento** è un puntatore in sola lettura ad un blocco di memoria **posseduto da un'altra variabile**
  - Permette di accedere ad un valore senza trasferirne la proprietà
  - Il riferimento può esistere solo mentre esiste la variabile che possiede il dato a cui punta
  - Il compilatore de-referenzia automaticamente il puntatore, quando si accede al valore tramite l'operatore '.'

```
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);
```

# Riferimenti e prestiti

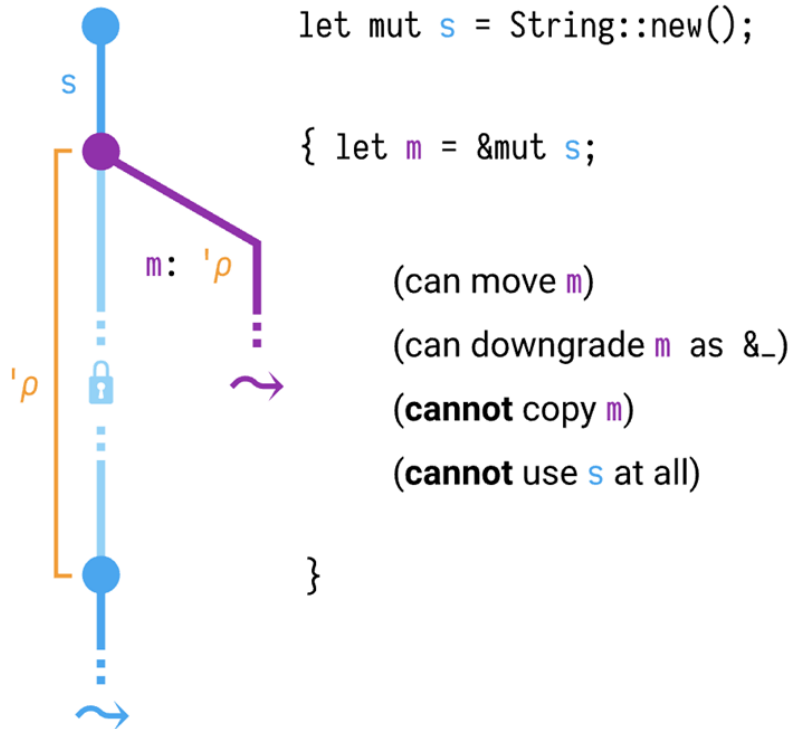
- Un riferimento **prende a prestito** (borrows) l'indirizzo di memoria in cui esiste il valore
  - Fino a che il riferimento è accessibile, non è possibile modificare il valore, né tramite il riferimento (che ha accesso in sola lettura), né tramite la variabile che possiede il valore
- E' possibile creare ulteriori riferimenti a partire dal dato originale o da altri riferimenti ad esso
  - I riferimenti sono **copiabili**: viene duplicato il puntatore
  - Il compilatore (attraverso il borrow checker) garantisce che un riferimento punti SEMPRE ad un dato valido
  - Finché esiste almeno un riferimento, il dato originale non potrà essere né modificato né distrutto

# Riferimenti mutabili

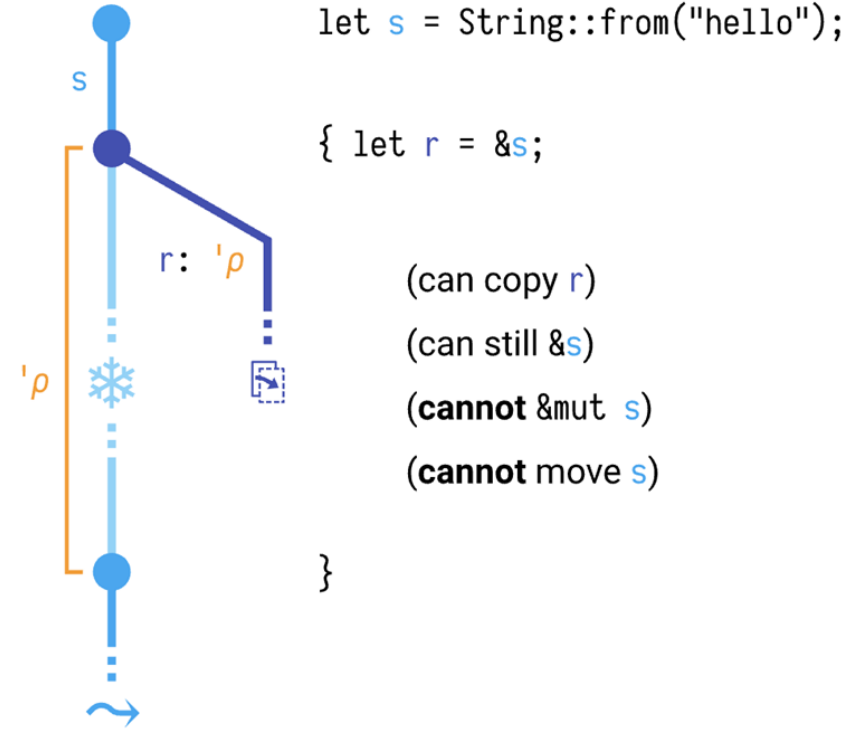
- A partire da una variabile che possiede un valore è possibile estrarre UN SOLO **riferimento mutabile** per volta
  - Si crea un riferimento mutabile con la sintassi **let r = &mut v;**
- Mentre esiste un riferimento mutabile...
  - Non possono esistere riferimenti semplici (in sola lettura)
  - Non è possibile modificare né muovere la variabile che possiede il valore
- E' possibile creare un riferimento mutabile **solo se** la variabile che possiede il dato è dichiarata mutabile a sua volta

# Riferimenti

## 🔒 mutable borrow



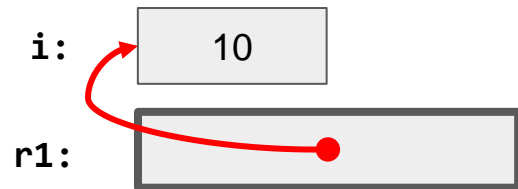
## \* borrow



# Riferimenti: disposizione in memoria

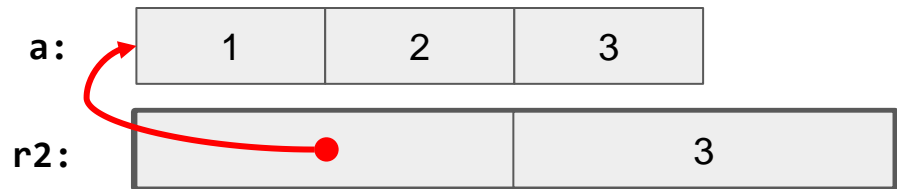
- I riferimenti sono implementati diversamente in base al tipo puntato
  - **Puntatori semplici**, se il compilatore conosce la dimensione del dato puntato
  - **Puntatore + dimensione** ( fat pointer), se la dimensione del dato è solo nota in fase di esecuzione
  - **Puntatore doppio**, se il tipo di dato puntato è noto solo per l'insieme di tratti che implementa

```
let i: i32 = 10;  
let r1: &i32 = &i;
```



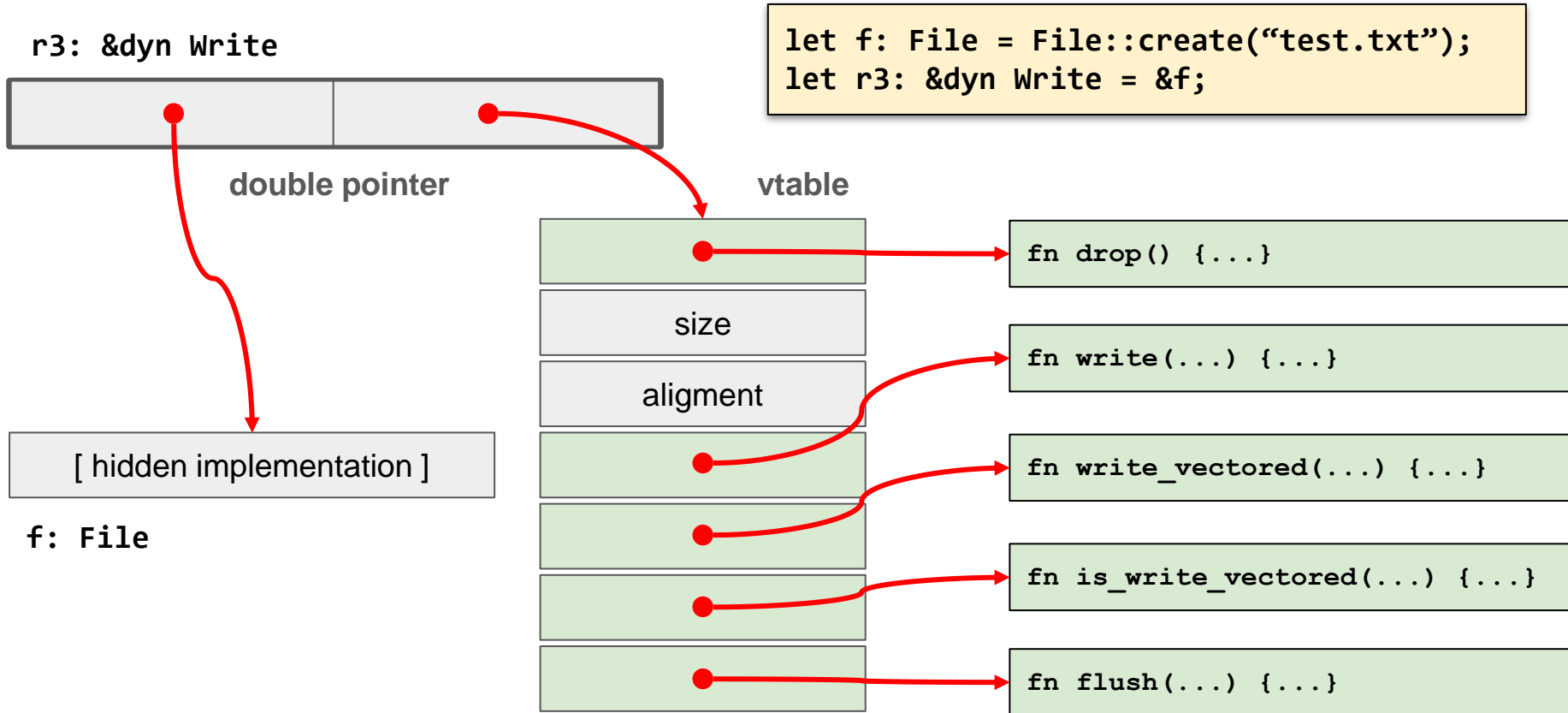
simple pointer

```
let a: [i32;3] = [1,2,3];  
let r2: &[i32] = &a;
```



fat pointer

# Riferimenti: disposizione in memoria



# Tempo di vita dei riferimenti

- Il borrow checker garantisce che tutti gli accessi ad un riferimento avvengano solo in un **intervallo di tempo** (righe) **compreso** in quelle in cui il dato esiste
  - Serve ad impedire il fenomeno dei *dangling pointer*
- L'insieme delle righe in cui si fa accesso al riferimento costituisce il suo **lifetime**
  - Tale informazione è mantenuta, dal compilatore, insieme alle informazioni che descrivono il tipo del riferimento e non ha nessuna rappresentazione in fase di esecuzione
- Sebbene in molte situazioni possa essere omesso, il tempo di vita di un riferimento può essere espresso nella firma del tipo, nel seguente modo:
  - **&'a NomeTipo**, dove **a** è un identificativo qualsiasi e il simbolo **'** (tick) serve a qualificarlo come tempo di vita
  - Tale notazione è utile in quelle situazioni in cui occorre imporre vincoli sulla durata relativa dei tempi di vita di due o più riferimenti
- Se un riferimento è valido per l'intera durata di un programma, viene indicato con la notazione **&'static NomeTipo**
  - 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

# Tempo di vita dei riferimenti

- Per essere lecito, **il tempo di vita** di un riferimento **deve essere contenuto nel** tempo di vita del **valore** a cui punta
  - Il borrow checker si fa carico di garantire tale vincolo
- Questo richiede di esplicitare il tempo di vita quando si memorizza un riferimento all'interno di una struttura dati o si usa un riferimento come valore di ritorno di una funzione
  - Questi due casi saranno trattati dettagliatamente in seguito
- Il vincolo (apparentemente ovvio) di esistenza in vita dei vincoli **dovrebbe** essere alla base anche di tutti gli usi dei puntatori in C e C++
  - Poiché, però, in questi linguaggi non viene tracciato il tempo di vita, il compilatore non è in grado di identificare la presenza di dangling pointers, né identificare eventuali problemi di non rilascio o doppio rilascio, come invece avviene in Rust



# Esistenza in vita

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

```
error: `x` does not live long enough  
    |  
9   |         r = &x;  
    |             ^^ borrowed value does not live long  
    |         enough  
10  |  
11  |     }  
    |     - `x` dropped here while still borrowed  
12  |  
13  |     assert_eq!(*r, 1);  
    |     ----- borrow later used here
```

# Esistenza in vita

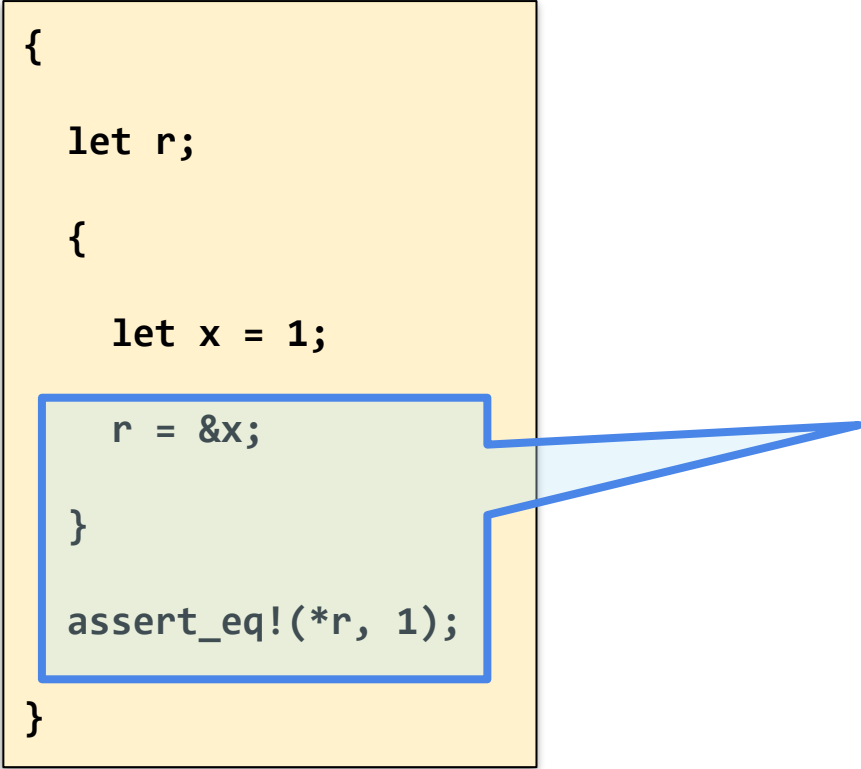
```
{  
  let r;  
  {  
    let x = 1;  
    r = &x;  
  }  
  assert_eq!(*r, 1);  
}
```

## Intervallo di esistenza di x

*qualunque riferimento ad x, non  
può eccedere questo periodo*

# Esistenza in vita

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



## Intervallo di validità di r

*il valore memorizzato al suo interno deve avere una vita che si estende per tutto questo periodo*

# Esistenza in vita

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

## Violazione dei vincoli

*Non è possibile soddisfare  
entrambi i vincoli*

# Esistenza in vita

- 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 delle regole

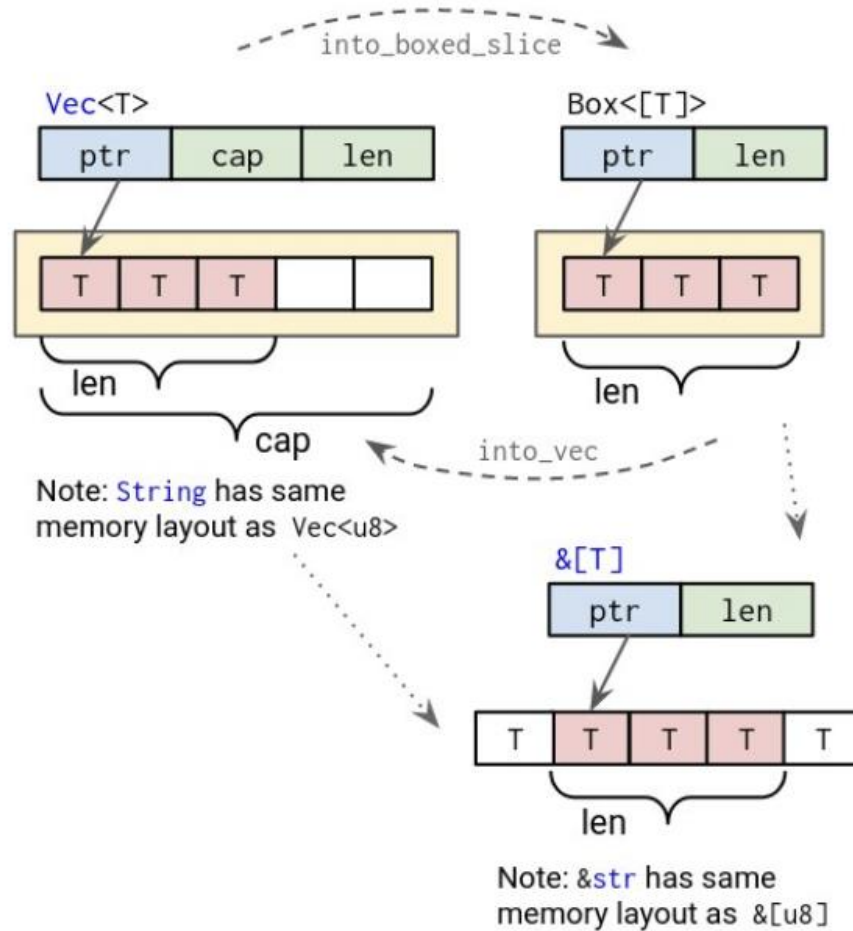
- Ciascun valore ha un **unico** possessore (variabile o campo di una struttura)
  - Il valore viene rilasciato (drop) quando il possessore esce dal proprio scope o quando al possessore viene assegnato un nuovo valore
- Può esistere, al più, un **singolo riferimento mutabile** ad un dato valore
- **Oppure**, possono esistere **molti riferimenti immutabili** al medesimo valore
  - Ma fintanto che ne esiste almeno uno, il valore non può essere mutato
- Tutti i riferimenti devono avere una **durata di vita inferiore** a quella del valore a cui fanno riferimento

# Slice

- Una slice (fetta) è una VISTA di una sequenza contigua di elementi, la cui lunghezza NON è nota in fase di compilazione, ma disponibile durante l'esecuzione
  - Internamente viene rappresentata come una **tupla di due elementi**, il primo dei quali punta al primo valore della sequenza, mentre il secondo indica il numero di elementi consecutivi
  - Una slice di elementi T ha tipo **[T]**
- Una slice **non possiede** i dati cui fa riferimento
  - Questi appartengono sempre ad un'altra variabile, da cui la slice viene derivata
  - Tutti i valori contenuti sono garantiti essere inizializzati, gli accessi sono verificati in fase di esecuzione
- E' possibile ricavare una slice a partire da un array, ma anche da altri tipi di contenitori (std::Vec<T>, String, Box<[T]>, ...)

```
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
```

# Slice





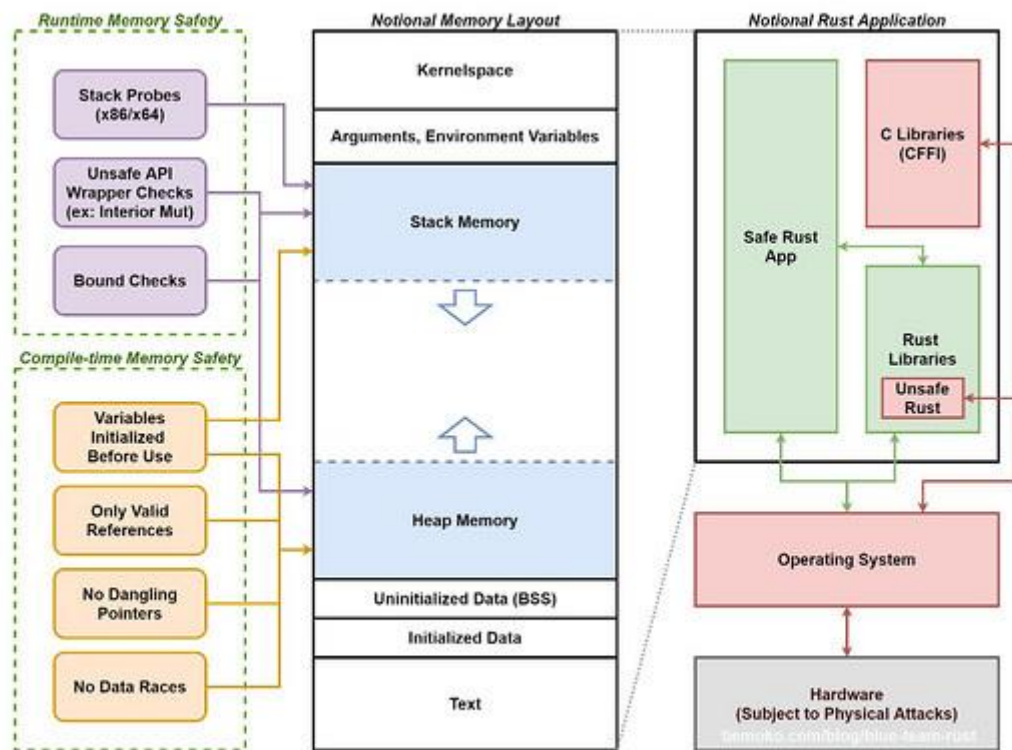
# Vantaggi introdotti dal concetto di possesso

- Un programma Rust, pur non avendo un garbage collector, offre molteplici garanzie relative alla correttezza degli accessi in memoria e del rilascio delle risorse
  - Non esiste il concetto di **riferimento nullo** e, di conseguenza, non c'è il rischio di dereferenziarlo
  - Poiché il borrow checker vigila sulle assegnazioni dei riferimenti e sugli intervalli temporali in cui essi sono effettivamente usati impedendo accessi illeciti, non c'è il rischio di originare **errori di segmentazione** o **accesso illegale** ad aree ristrette di memoria, né la possibilità di avere riferimenti ad aree già rilasciate (**dangling pointer**)
  - Poiché in ogni istante il borrow checker è in grado di determinare la dimensione di un blocco di memoria cui un riferimento punta, non possono verificarsi **buffer overflow** né **buffer underflow**
  - Per lo stesso motivo, gli **iteratori** offerti da Rust **non eccedono** mai **i loro limiti**

# Vantaggi introdotti dal concetto di possesso

- Tutte le variabili sono **immutabili** per default e occorre una dichiarazione esplicita per renderle mutabili
  - Questo obbliga il programmatore a riflettere attentamente sul come e dove i dati debbano essere modificati e su quale sia il ciclo di vita di ciascun valore
- Il modello di possesso non riguarda solo la gestione della memoria, ma anche la **gestione delle risorse** contenute in un valore
  - Come socket di rete, handle di file e database, descrittori dei dispositivi, ...
- L'assenza di un garbage collector impedisce **comportamenti non deterministici**
  - In particolare, la sospensione totale del funzionamento ogni qual volta occorra ricompattare la memoria

# Disposizione in memoria





## Per saperne di più

- Rust Ownership by Example
  - <https://depth-first.com/articles/2020/01/27/rust-ownership-by-example/>
  - Carrellata di esempi, progressivamente più articolati, per illustrare il concetto di possesso
- Understanding Ownership in Rust with Examples
  - <https://medium.com/coinmonks/understanding-ownership-in-rust-with-examples-73835ba931b1>
  - Altro articolo introduttivo che illustra i diversi aspetti collegati al possesso
- Rust Lifetimes: A Complete Guide to Ownership and Borrowing
  - <https://earthly.dev/blog/rust-lifetimes-ownership-burrowing/>
  - Approfondimento sul concetto di tempo di vita di un valore e gestione dei prestiti