

I dettagli del linguaggio

2023-24

Variabili e tipi

- Una variabile lega un valore ad un nome
 - Viene introdotta dalla parola chiave **let**
- Rust favorisce l'**immutabilità**
 - Di base, una variabile può essere legata ad UN SOLO VALORE, per tutta la sua esistenza
 - E' possibile indicare che la variabile potrà essere legata, in futuro, ad un altro valore (dello stesso tipo) aggiungendo alla parola chiave **let** il modificatore **mut**
- Ad ogni variabile è associato **staticamente** (cioè, per tutta la durata del programma) un tipo
 - Il tipo definisce l'insieme dei valori che possono essere memorizzati in una variabile, così come l'insieme delle operazioni che possono essere effettuate su tali valori
- Il tipo associato ad una variabile può essere esplicitamente definito nella clausola **let** che introduce la variabile
 - Il motore di inferenza dei tipi consente, nella maggior parte dei casi, di dedurre il tipo di una variabile dal valore con cui è stata inizializzata, rendendo opzionale la dichiarazione esplicita

Variabili e tipi

```
let v: i32 = 123; // v è immutabile e ha tipo i32 (intero a 32 bit con segno)
// v = -5;           // ERRORE: Non è possibile riassegnare il valore
```

```
let mut w = v;      // w può essere riassegnata, ha lo stesso tipo di v (i32)
w = -5;             // OK. Ora w vale -5
```

```
let x = 1.3278;     // x è immutabile di tipo f64 (floating point a 64 bit)
```

```
let y = 1.3278f32;  // y è immutabile di tipo f32 (floating point a 32 bit)
```

```
let one_million = 1_000_000 // si possono usare '_' per separare le cifre
```

Shadow

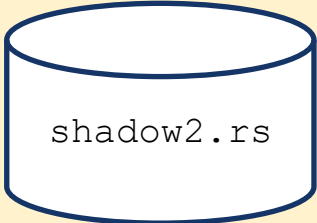
```
fn main() {  
    let i:i32 = -12;  
  
    println!("Numero {i}");  
  
    let i = "ciao mamma";    // la presenza della seconda variabile i nasconde  
                             // la prima  
  
    println!("Stringa {i}");  
}
```



Shadow (II)

- Con lo shadowing si crea una nuova variabile che nasconde la copia precedente, ma non la distrugge.

```
fn main() {  
  
    let mut x = 5;  
  
    x = x + 1;  
  
    {  
        let x = x * 2;  
        println!("The value of x in the inner scope is: {x}");  
    }  
  
    println!("The value of x is: {x}");  
}
```

A diagram of a cylinder, representing a file or a module, with the text "shadow2.rs" written inside it. The cylinder is drawn with a blue outline and a light blue fill.

Shadow (III)

- Con lo *shadowing* si crea una nuova variabile che nasconde la copia precedente, ma non la distrugge.

```
fn main() {  
  
    let mut x = 5;  
  
    x = x + 1;  
  
    {  
        let x = x * 2;  
        println!("The value of x in the inner scope is: {x}");  
    }  
  
    println!("The value of x is: {x}");  
}
```

```
The value of x in the inner scope is: 12  
The value of x is: 6
```

Shadow vs. Mut

- Con shadow si può cambiare tipo

```
fn main() {  
  
let spaces = "  ";  
let spaces = spaces.len();  
}
```

- Con mut non si può cambiare tipo

```
fn main() {  
  
let mut spaces = "  ";  
spaces = spaces.len();  
}
```

```
3 | let mut spaces = "  ";  
   |                   ----- expected due to this value  
4 | spaces = spaces.len();  
   |                   ^^^^^^^^^^^^^ expected `&str`, found `usize`
```

Valore iniziale

- Ad una variabile dobbiamo assegnare un **valore iniziale** che può essere:
 - un valore immediato
let x = 2;
 - il risultato di una espressione
let x = a * 2;

Valori ed espressioni

- Un'**espressione** è un costrutto sintattico la cui esecuzione produce un **valore** di un dato **tipo**
 - `4 + (3 * 2)`
 - Le espressioni sono valutate bottom-up, si possono usare le parentesi tonde per modificare l'ordine con cui un'espressione viene valutata
 - Una variabile è un modo comodo per catturare il risultato di un'espressione attribuendogli un nome, per poterlo usare in seguito
- **Tutte le espressioni producono un valore**
 - A differenza di quanto avviene in C e in C++, la maggior parte delle istruzioni vengono viste come espressioni (e, di conseguenza, producono un valore)
 - Ma, al contrario, alcuni costrutti che in C e C++ restituiscono un valore (come le assegnazioni), in Rust hanno tipo `()` - che corrisponde al tipo `void` del C/C++

Assegnazione

- In C/C++ l'assegnazione è vista come una funzione e restituisce un valore.

```
int a = b = c = 12;
```



- In RUST non è possibile perché vincola al principio di transitività tra tipi che non è detto che sia valido.

Tipi

- Rust offre un insieme di tipi predefiniti (tipi elementari, tuple, stringhe, array, slice, never, vari tipi di puntatori, ...)
 - E' possibile definire tipi ulteriori sotto forma di struct, enum, funzioni, chiusure,
- A differenza di quanto succede in altri linguaggi, i tipi **NON** sono organizzati in una gerarchia di ereditarietà, ma sono tutti indipendenti tra di loro.

Tipi elementari

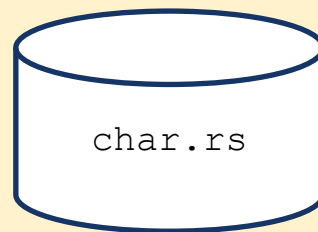
- Numeri interi con segno
 - `i8, i16, i32, i64, i128, isize`
- Numerici interi senza segno
 - `u8, u16, u32, u64, u128, usize`
- In virgola mobile (IEEE 754)
 - `f32, f64`
- Logici
 - `bool`
- Caratteri (32 bit, Unicode Scalar Value)
 - `char`
- Unit
 - `()` - rappresenta una tupla di 0 elementi, `()` indica sia il tipo che il suo unico possibile valore.
 - Serve per descrivere il comportamento di quelle funzioni che non ritornano un valore esplicito e corrisponde al tipo `void` in C/C++

Le stringhe contengono array di caratteri codificati con UTF-8

Tipo char

- Carattere: il tipo char è separato da singoli apici e può contenere un valore nella rappresentazione Unicode.

```
fn main() {  
    let c = 'z';  
    let z = 'Z';  
    let heart_eyed_cat = '😻';  
    println!("The value of this char is: {heart_eyed_cat}");  
}
```



Tratto

- Un **tratto** descrive un insieme di comportamenti (**metodi**) che un dato **tipo implementa**
 - Il compilatore utilizza l'informazione che un certo tipo implementa un dato tratto per governare il codice che viene generato manipolando un'espressione che consuma o genera quel particolare tipo
 - I tratti assomigliano a quelle che in altri linguaggi sono chiamate interfacce: insiemi di metodi privi di implementazione o con un'implementazione di default (sovrascrivibile)
- Qualsiasi tipo, predefinito o meno, può implementare zero o più tratti
- Ad esempio il tipo `int`
 - gode del tratto `Copy`, significa che permette di effettuare una copia di un dato di tipo intero a fronte di un'assegnazione (il dato intero di destra viene duplicato nel dato di sinistra)
 - gode del tratto `Eq`, significa che un dato intero è confrontabile con un altro intero e posso applicare l'operatore `==`
 - Gode del tratto `Ord`, significa che posso valutare la relazione d'ordine e applicare gli operatori relazionali `>`, `<`, `>=`, `<=`.

Dipendenza tra tratti

- I tratti sono legati tra di loro in una relazione di dipendenza che può essere:
 - Negativa, ad esempio chi gode del tratto Drop non può avere il tratto Copy e viceversa
 - Positiva, ad esempio chi ha il tratto Copy deve avere il tratto Clone.
- Esistono anche tratti privi di metodi (tratti **marker**) che definisce una proprietà del tipo. Ad esempio i tratti della programmazione concorrente Sync e Send sono tratti privi di metodi e i tipi possono godere del tratto Sync oppure Send oppure di entrambi oppure di nessuno dei due.

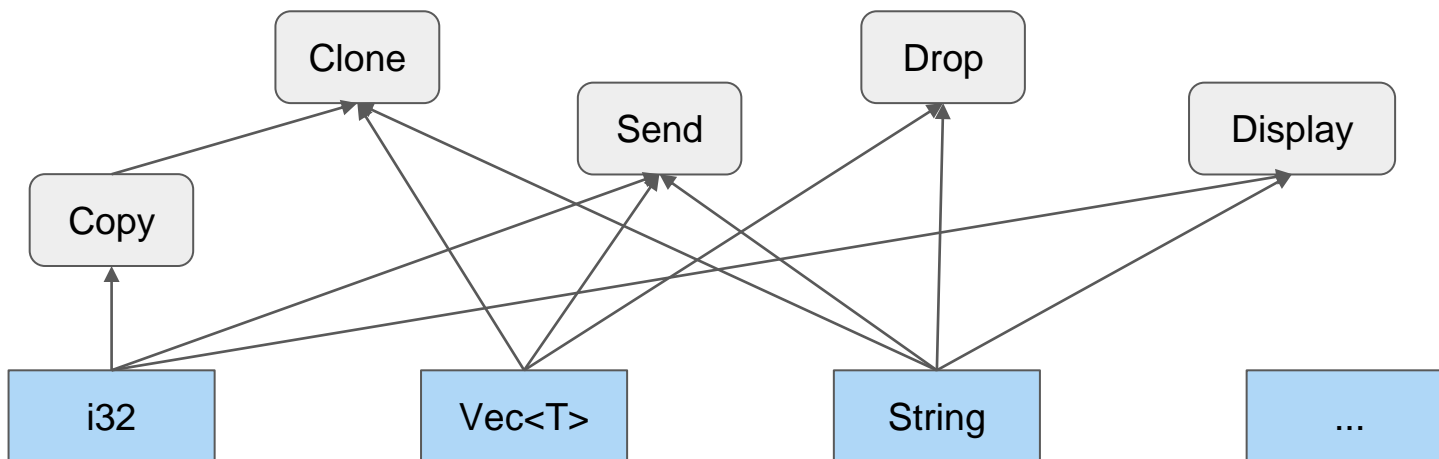
Il compilatore verifica che var1 e var2 siano dello stesso tipo che implementi il tratto Ord o PartialOrd

```
...  
var1 < var2  
...
```

Se non è verificato il
compilatore segnala Errore

Tipi e tratti

- Poiché tipi diversi possono implementare tratti comuni, si viene a creare una forma di “parentela” alquanto articolata tra tipi
 - Rust introduce una ventina di tratti predefiniti, cui il compilatore associa un particolare significato, e permette al programmatore di aggiungerne altri a piacere, al fine di estendere tale comportamento



Tuple

Accediamo in modo posizionale, ma non è un Array

- Rappresentano collezioni ordinate di valori eterogenei
 - Vengono costruite racchiudendo i valori in parentesi tonde
 - Una tupla ha tipo (T_1, T_2, \dots, T_n) , dove T_1, T_2, \dots, T_n sono i tipi dei singoli valori membro
- Si accede al contenuto di una tupla utilizzando la notazione `.0`, `.1`, ...
 - Una tupla può contenere un numero arbitrario di valori
 - Simili a `std::tuple` del C++ ma con accesso semplificato (`.<numero campo>` al posto di `std::get< <numero campo> >(tupla)`)

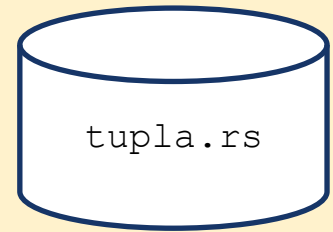
```
let t: (i32, bool) = (123, false); // t è una tupla formata da un intero
                                   // e da un booleano

let mut u = (3.14, 2.71);           // u è una tupla riassegnabile formata
                                   // da due double
let i = t.0;                        // i contiene il valore 123

u.1 = 0.0;                          // adesso u contiene (3.14, 0.0)
```

Destrutturare una tupla

```
fn main() {  
    let tup = (500, 6.4, 1);  
  
    let (x, y, z) = tup;  
  
    println!("The value of y is: {y}");  
}
```



Puntatori e memoria

- Rust offre vari modi per rappresentare indirizzi in memoria
 - Riferimenti (condivisi e mutabili)
 - Box
 - Puntatori nativi (costanti e mutabili)
- A differenza di quanto capita in C e C++, l'uso dei puntatori è abbondantemente semplificato grazie alle garanzie offerte dal compilatore del linguaggio
 - Che verifica il **possesso** ed il **tempo di vita** delle variabili e garantisce che possano avvenire solo accessi che è possibile dimostrare essere leciti (oppure forza il programmatore ad assumersi la responsabilità della correttezza del proprio operato racchiudendo il codice in un blocco **unsafe** {...})

Riferimenti

- L'espressione **let r1 = &v;**, dove **v** è un qualsiasi valore o espressione, definisce ed inizializza il **riferimento r1**
 - La variabile **r1** prende in prestito (*borrows*) il valore **v** e potrà accedervi (**in sola lettura**) con l'espressione ***r1**
 - Un riferimento viene rappresentato internamente come un blocco di memoria contenente l'indirizzo di memoria in cui il valore è memorizzato
 - I riferimenti in sola lettura possono essere copiati, assegnandoli ad un'altra variabile o passandoli come parametro ad una funzione: ma fino a che esiste almeno un riferimento ed è in uso, il valore originale non è modificabile
- L'espressione **let r2 = &mut v;** definisce ed inizializza il riferimento mutabile **r2**
 - La variabile **r2** prende in a prestito, **in modo esclusivo**, il valore **v** e permette di modificarlo (ad esempio, scrivendo ***r2 = ...;**)
 - Finché un riferimento mutabile esiste ed è in uso, non è possibile né creare altri riferimenti (mutabili o meno) al valore originale, né accedere in alcun modo al valore originale (**mutuamente esclusivo**).

Riferimenti

Si legge Ref

- L'espressione **let r1 = &v;**, dove **v** è un qualsiasi valore o espressione, definisce ed inizializza il **riferimento r1**
 - La variabile **r1** prende in prestito (**borrows**) il valore **v** e potrà accedervi (**in sola lettura**) con l'espressione ***r1**
 - Un riferimento viene rappresentato internamente come un blocco di memoria contenente l'indirizzo di memoria in cui il valore è memorizzato
 - I riferimenti in sola lettura possono essere usati sia come variabile o passandoli come parametro ad una funzione: ma fino a quel momento ed è in uso, il valore originale non è modificabile
- L'espressione **let r2 = &mut v;** definisce ed inizializza il riferimento mutabile **r2**
 - La variabile **r2** prende in prestito, **in modo esclusivo**, il valore **v** e permette di modificarlo (ad esempio, scrivendo ***r2 = ...;**)
 - Finché un riferimento mutabile esiste ed è in uso, non è possibile né creare altri riferimenti (mutabili o meno) al valore originale, né accedere in alcun modo al valore originale (**mutuamente esclusivo**).

Si legge RefMut

Ref e RefMut

- Sono puntatori privi di possesso
- Li posso ricavare soltanto da una variabile che già esiste
- Chi li riceve non ha la responsabilità del rilascio che rimane al possessore originale
- Se voglio operare sul dato puntato posso invocare un metodo senza dereferenziare il Ref. Il compilatore va a vedere che cosa c'è prima del punto e se è un Ref lo dereferenzia automaticamente:
 - se fosse un Ref ad un Ref viene dereferenziato 2 volte
 - Se fosse un Ref ad un Ref ad un Ref viene dereferenziato 3 volte
 - e così via fino ad arrivare al tipo base a cui applica il metodo.



```
fn main() {  
    let v = 32;  
    let p = &v;  
    let pp = &p;  
    let ppp = &pp;  
  
    let str = ppp.to_string();  
  
    println!("{str}");  
}
```

Riferimenti

- Sebbene possano apparire simili ai puntatori in C/C++, i riferimenti Rust non possono mai essere nulli
 - Né, tantomeno, contenere l'indirizzo di un valore che è stato già rilasciato o non è stato inizializzato
- I riferimenti implementano una logica *single writer or multiple readers*
 - Il compilatore, attraverso un apposito modulo detto *borrow checker* si fa carico di garantire questa regola che costituisce la base delle regole di sanità all'interno di Rust
- Un riferimento Rust è profondamente diverso dall'equivalente C++, pur avendo una notazione vagamente simile
 - In C++, è lecito costruire solo riferimenti a variabili, non al risultato di un'espressione temporanea: un riferimento costituisce un alias alla variabile a cui è stato inizializzato
 - Poiché il C++ non gestisce in modo specifico l'esistenza in vita né tiene conto delle duplicazioni, i riferimenti C++ possono diventare invalidi e dare origine a comportamenti non definiti

Riferimenti

```
fn main() {  
    let mut i = 32;  
  
    let r = &i;  
    println!("{}", *r);  
  
    i = i+1;    // Problematico!  
    println!("{}", *r);  
}
```

```
fn main() {  
    let mut i = 32;  
  
    let r = &mut i;  
    println!("{}", i); // Problema!  
  
    *r = *r+1;  
    println!("{}", *r);  
}
```

error[E0506]: cannot assign to `i` because it is borrowed

--> [src/main.rs:11:3](#)

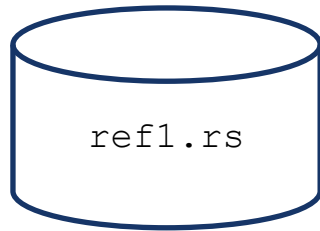
```
8 |     let r = &i;  
  |           -- borrow of `i` occurs here  
...  
11 |     i = i+1;  
   |     ^^^^^^ assignment to borrowed `i` occurs here  
12 |     println!("{}", *r);  
   |                   -- borrow later used here
```

error[E0502]: cannot borrow `i` as immutable because it is also borrowed as mutable

--> [src/main.rs:9:18](#)

```
8 |     let r = &mut i;  
  |           ----- mutable borrow occurs here  
9 |     println!("{}", i);  
  |                   ^ immutable borrow occurs here  
10 |  
11 |     *r = *r+1;  
   |         -- mutable borrow later used here
```

Riferimenti



```
fn main() {  
    let mut i = 32;  
  
    let r = &i;  
    println!("{}", *r);  
  
    i = i+1;  
    println!("{}", i);  
}
```

```
fn main() {  
    let mut i = 32;  
  
    let r = &mut i;  
  
    *r = *r+1;  
    println!("{}", *r);  
  
    println!("{}", i);  
}
```

Il tempo di vita effettivo
termina all'ultimo
accesso

Riferimenti a espressioni

```
fn main() {  
    let a = 2;  
  
    let p = &mut 12;  
    *p = *p + 1;  
  
    let p1 = &(a + 2);  
  
    let p2 = &(*p1 * 2);  
  
    println!("{}", *p, *p1, *p2);  
}
```



Box<T>

- Una variabile locale è **SEMPRE** allocata sullo stack
 - Questo si estende di una quantità pari alla dimensione del valore che deve essere memorizzato nella variabile
 - Quando la variabile esce dal proprio scope sintattico (quando, cioè, il programma raggiunge la fine del blocco in cui la variabile è stata definita), lo stack si contrae ed il valore viene rilasciato
- In alcune situazioni **non è nota la dimensione** del dato che deve essere memorizzato
 - In altri, occorre **prolungare il tempo di vita** del valore oltre quello del blocco sintattico in cui è definito
- In queste occasioni si può allocare un oggetto sullo **heap**, utilizzando il tipo generico

Box<T>

- Una variabile di questo tipo contiene il puntatore al valore e **possiede** il dato
- Il Box implementa il tratto **Drop** (distruttore) e quando il Box sparirà il dato verrà distrutto (automaticamente) e rilasciata la memoria sullo heap

Box<T>

- Si alloca un valore di tipo Box con il costrutto

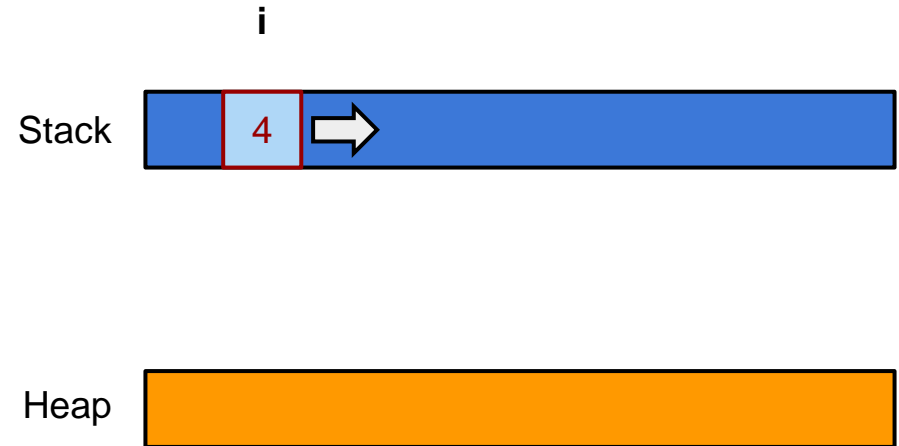
let b = Box::new(v);

dove **v** è un qualsiasi valore

- Questa istruzione definisce la variabile **b** che conterrà un puntatore ad un blocco allocato sullo heap che a sua volta contiene il valore **v**
- Si accede al valore contenuto nel blocco con l'espressione ***b**
 - Se la variabile **b** è definita come mutabile, è possibile modificare il contenuto a cui si punta con l'espressione ***b = ...;**
- Quando l'esecuzione del programma raggiungerà la fine del blocco di codice in cui la variabile **b** è stata definita (fine del sua visibilità sintattica), il blocco sarà rilasciato
 - A meno che il contenuto di **b** (il puntatore al blocco) sia stato **mosso** in un'altra variabile

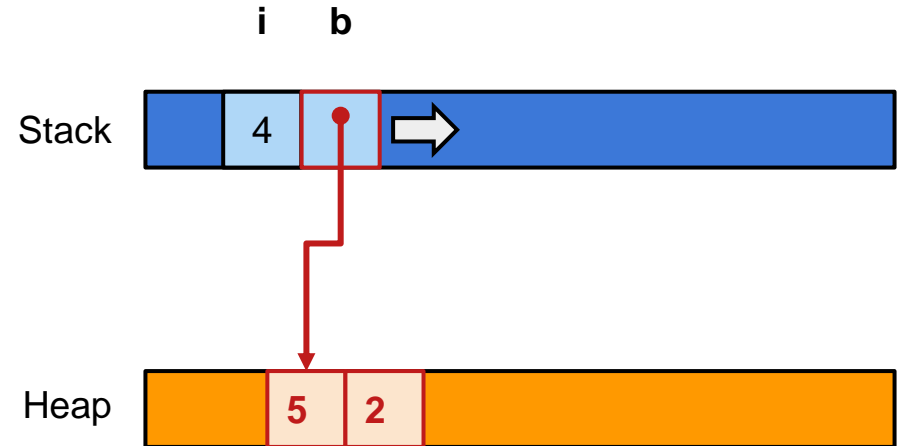
Box<T>

```
fn useBox() {  
  let i = 4;  
  let mut b = Box::new( (5, 2) );  
  
  (*b).1 = 7;  
  
  println!("{:?}", *b); // (5,7)  
  println!("{:?}", b);  // (5,7)  
}
```



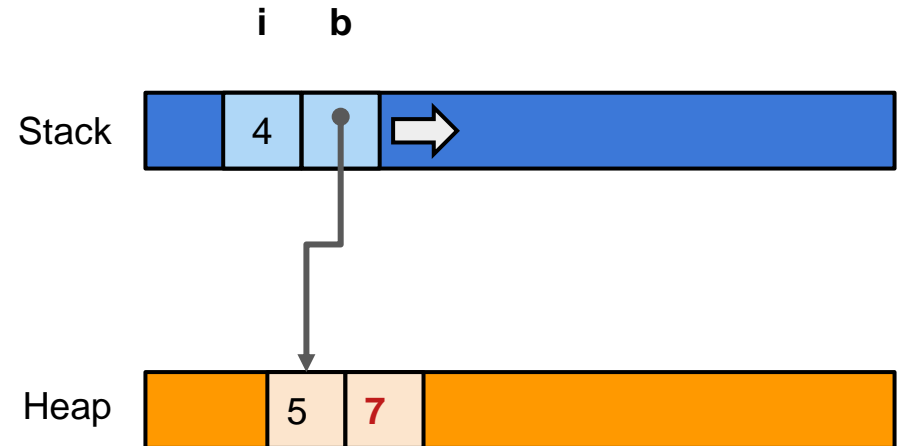
Box<T>

```
fn useBox() {  
  let i = 4;  
  let mut b = Box::new( (5, 2) );  
  
  (*b).1 = 7;  
  
  println!("{:?}", *b); // (5,7)  
  println!("{:?}", b);  // (5,7)  
}
```



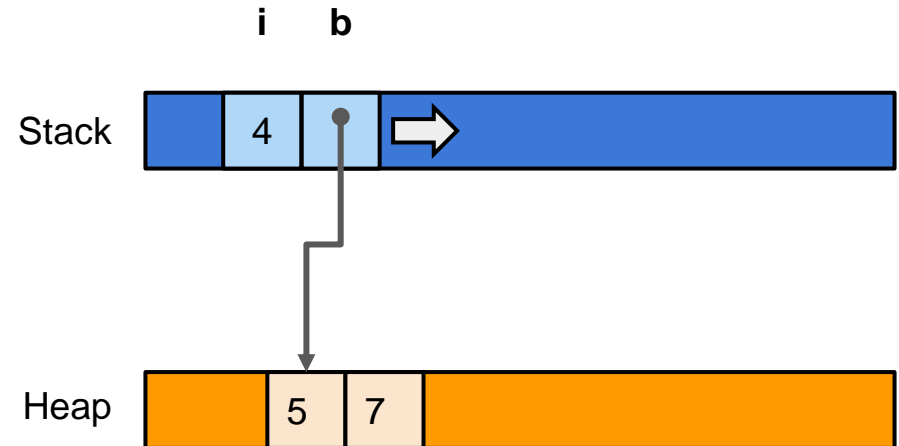
Box<T>

```
fn useBox() {  
  let i = 4;  
  let mut b = Box::new( (5, 2) );  
  
  (*b).1 = 7;  
  
  println!("{:?}", *b); // (5,7)  
  println!("{:?}", b);  // (5,7)  
}
```



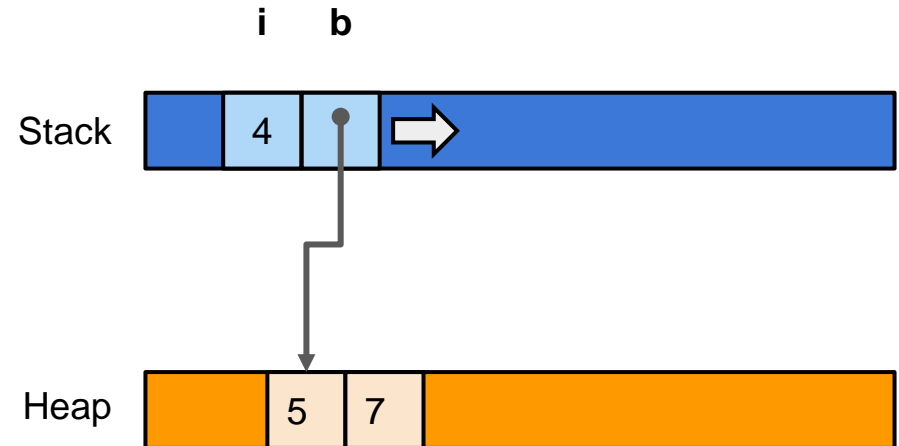
Box<T>

```
fn useBox() {  
  let i = 4;  
  let mut b = Box::new( (5, 2) );  
  
  (*b).1 = 7;  
  
  println!("{:?}", *b); // (5,7)  
  println!("{:?}", b);  // (5,7)  
}
```



Box<T>

```
fn useBox() {  
  let i = 4;  
  let mut b = Box::new( (5, 2) );  
  
  (*b).1 = 7;  
  
  println!("{:?}", *b); // (5,7)  
  println!("{:?}", b);  // (5,7)  
}
```



Box<T>

```
fn useBox() {  
  let i = 4;  
  let mut b = Box::new( (5, 2) );  
  
  (*b).1 = 7;  
  
  println!("{:?}", *b); // (5,7)  
  println!("{:?}", b);  // (5,7)  
}
```

Stack



Heap



Tratti per Stampa

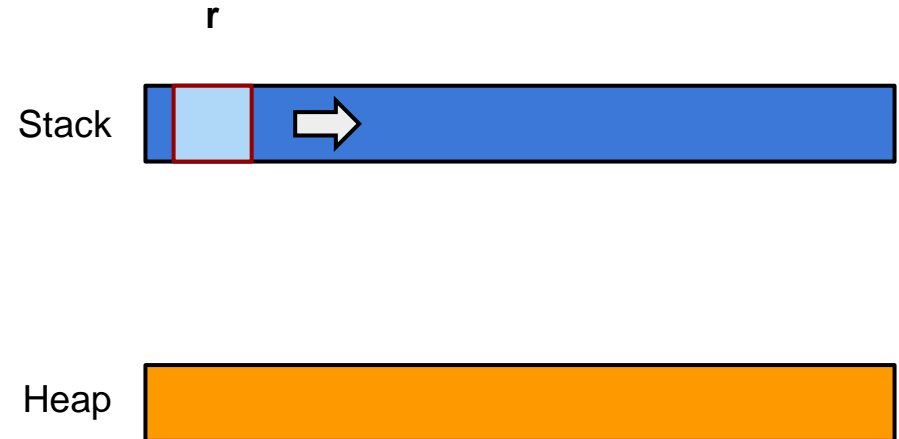
- Il tratto Display stampa una variabile riferita con `{}` e fornisce un formato di stampa orientato all'utente finale
- Il tratto Debug stampa una variabile riferita con `{:?}` e fornisce un formato orientato al programmatore; l'opzione `{:#?}` fornisce un formato più leggibile (*pretty printing*)
- Il tratto Pointer `{:p}` permette di stampare l'indirizzo di una variabile puntatore

```
fn main () {  
    let x = &42;  
  
    println!("{x:p}");  
}
```

- Le tuple implementano il tratto Debug
- I numeri implementano, per default, il tratto Display.

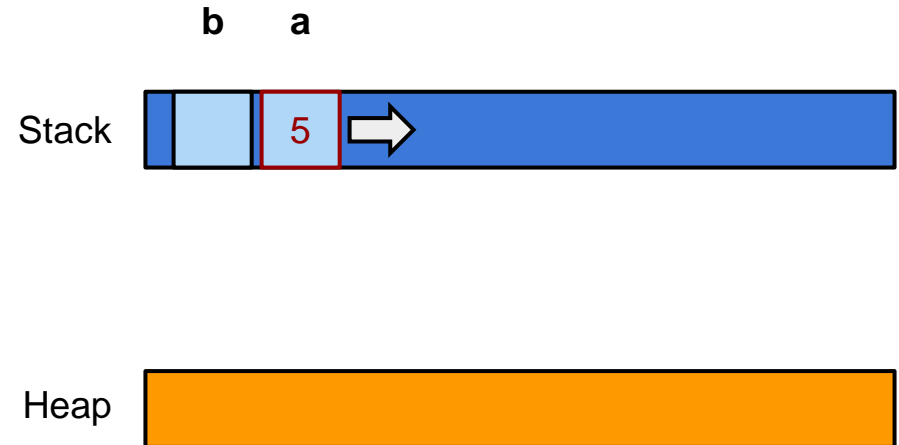
Box<T>

```
fn makeBox(a: i32) -> Box<(i32,i32)> {  
  let r = Box::new( (a, 1) );  
  return r;  
}  
  
fn main() {  
  let b = makeBox(5);  
  let c = b.0 + b.1;  
}
```



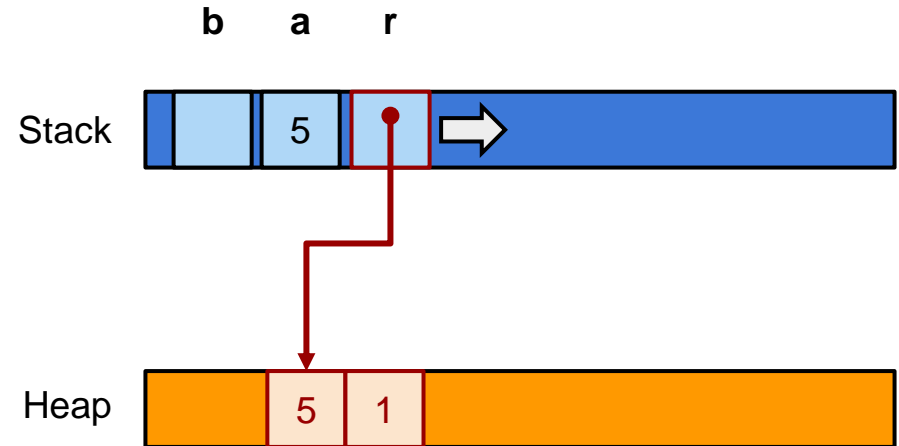
Box<T>

```
fn makeBox(a: i32) -> Box<(i32,i32)> {  
  let r = Box::new( (a, 1) );  
  return r;  
}  
  
fn main() {  
  let b = makeBox(5);  
  let c  = b.0 + b.1;  
}
```



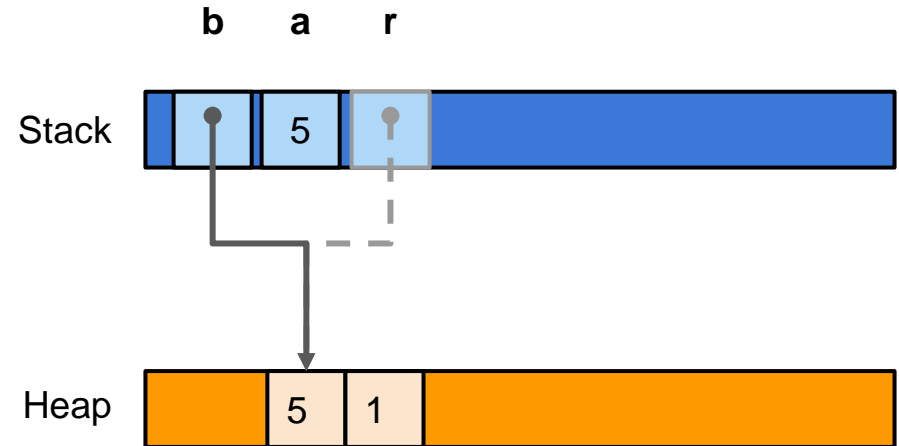
Box<T>

```
fn makeBox(a: i32) -> Box<(i32,i32)> {  
    let r = Box::new( (a, 1) );  
    return r;  
}  
  
fn main() {  
    let b = makeBox(5);  
    let c = b.0 + b.1;  
}
```



Box<T>

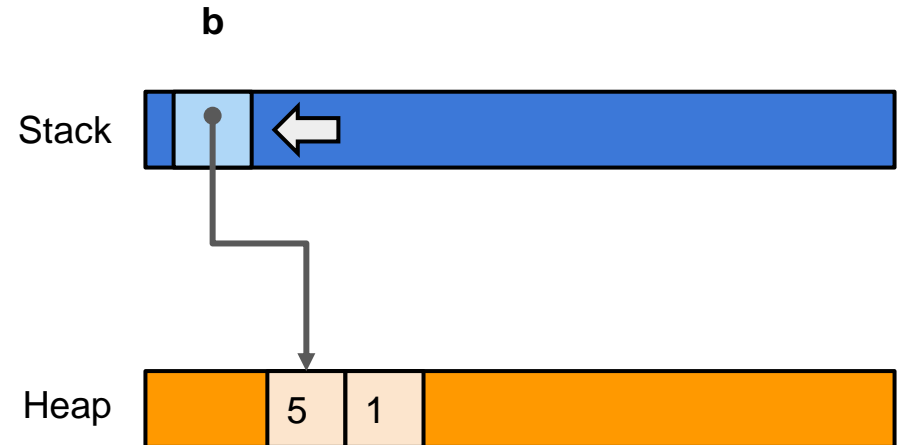
```
fn makeBox(a: i32) -> Box<(i32,i32)> {  
  let r = Box::new( (a, 1) );  
  return r;  
}  
  
fn main() {  
  let b = makeBox(5);  
  let c = b.0 + b.1;  
}
```



L'assegnazione determina un movimento che sposta il possesso del dato

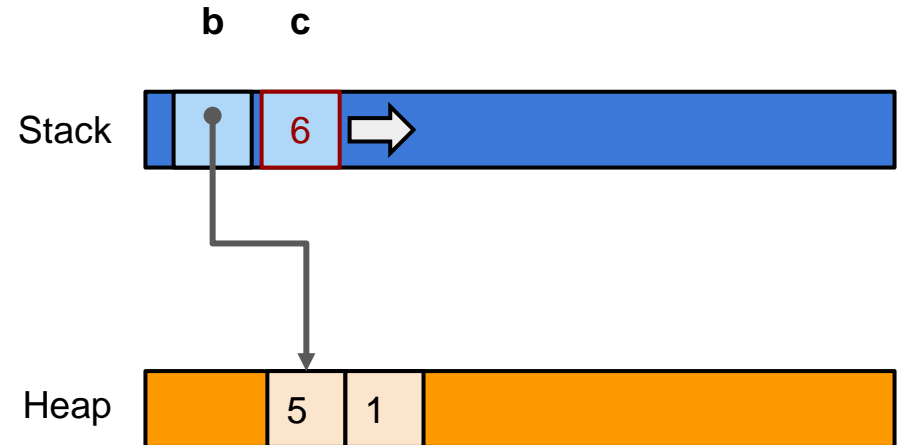
Box<T>

```
fn makeBox(a: i32) -> Box<(i32,i32)> {  
  let r = Box::new( (a, 1) );  
  return r;  
}  
  
fn main() {  
  let b = makeBox(5);  
  let c  = b.0 + b.1;  
}
```



Box<T>

```
fn makeBox(a: i32) -> Box<(i32,i32)> {  
  let r = Box::new( (a, 1) );  
  return r;  
}  
  
fn main() {  
  let b = makeBox(5);  
  let c = b.0 + b.1;  
}
```



Box<T>

```
fn makeBox(a: i32) -> Box<(i32,i32)> {  
  let r = Box::new( (a, 1) );  
  return r;  
}  
  
fn main() {  
  let b = makeBox(5);  
  let c  = b.0 + b.1;  
}
```

Stack



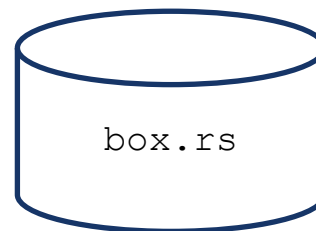
Heap



Cambio di Possesso

- Il nuovo possessore può cambiare la mutabilità (decisione del possessore)

```
fn makeBox(a: i32) -> Box<(i32,i32)> {  
    let r = Box::new( (a, 1) );  
    return r;  
}  
  
fn main() {  
    let mut b = makeBox(5);  
    b.0 = b.0 * 2;  
    let c  = b.0 + b.1;  
  
    println!("{c}");  
}
```



Copy vs. Clone

- Il tratto Copy permette di duplicare valori attraverso memcpy
- Se il tipo implementa solo il tratto Clone, l'operatore = determina un movimento del dato

```
fn main() {  
  
let x: u8 = 123; // u8 implements Copy  
let y = x;      // x can still be used
```

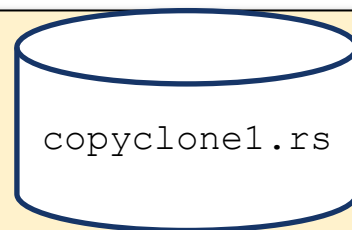
```
println!("x={}", y);
```

```
let v: Vec<u8> = vec![1, 2, 3]; // Vec<u8> implements Clone, but not Copy
```

```
let w = v;           // This would move the value, rendering v unusable
```

```
println!("w={:?}", w);
```

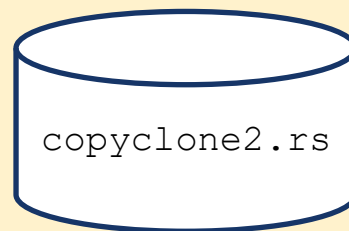
```
}
```



Copy vs. Clone (II)

- Se voglio copiare un dato, senza movimento (deep copy) si deve utilizzare esplicitamente il metodo `.clone()`

```
fn main() {  
    let mut v: Vec<u8> = vec![1, 2, 3]; // Vec<u8> implements Clone, but not Copy  
    let mut w = v.clone();    // This would move the value, rendering v unusable  
  
    v[0] = 10;  
    w[1] = 50;  
  
    println!("v={:?} w={:?}", v, w);  
}
```



Copy vs. Clone (III)

- Il tratto Copy determina il tratto Clone, ma il metodo `.clone()` può essere definito in modo arbitrario dal programmatore
- il tratto Copy è eseguito implicitamente e non può essere re-implementato,
- il tratto Clone deve essere eseguito esplicitamente e può essere re-implementato

```
#[derive(Debug, Clone, Copy)]
pub struct PointCloneAndCopy {
    pub x: f64
}
#[derive(Debug, Clone)]
pub struct PointCloneOnly {
    pub x: f64
}
fn test_copy_and_clone() {
    let p1 = PointCloneAndCopy { x: 0. };
    let p2 = p1; // because type has Copy, it gets copied automatically.
    println!("{:?} {:?}", p1, p2);
}
fn test_clone_only() {
    let p1 = PointCloneOnly { x: 0. };
    let p2 = p1; // because type has no Copy, this is a move instead.
    println!("{:?} {:?}", p1, p2);
}
pub fn main() {
    test_copy_and_clone();
    test_clone_only();
}
```

copyclone3.rs

Drop


```
struct S(i32); // struct che contiene un unico campo di tipo i32
               // alla struct associo dei metodi

impl S {
    fn display (&self) { // metodo chiamato su un oggetto di S
                        // ricevendolo come riferimento non mutabile
        println!("Sono S e contengo {} @ {:p}", self.0, self);
                        // mi dice che cosa contiene S e dove è memorizzato
    }
}

impl Drop for S { // metodo utilizzato per rilasciare la memoria
    fn drop(&mut self) // ha la possibilità di manipolare il dato
    {
        println!("Dropping S{} @ {:p}", self.0, self);
    }
}

fn main() {
    let s1 = S(1);
    let s2 = Box::new(S(2));

    s1.display();
    s2.display();
}
```



drop1.rs

```
Sono S e contengo 1 @ 0x42f65af844
Sono S e contengo 2 @ 0x23b2c13ab10
Dropping S(2) @ 0x23b2c13ab10
Dropping S(1) @ 0x42f65af844
```

Movimento

```
struct S(i32); // struct che contiene un unico campo di tipo i32
               // alla struct associo dei metodi

impl S {
    fn display (&self) { // metodo chiamato su un oggetto di S
                        // ricevendolo come riferimento non mutabile
        println!("Sono S e contengo {} @ {:p}", self.0, self);
                        // mi dice che cosa contiene S e dove è memorizzato
    }
}

impl Drop for S {
    fn drop(&mut self) // ha la possibilità di manipolare il dato
    {
        println!("Dropping S{} @ {:p}", self.0, self);
    }
}

fn main() {
    let s1 = S(1);
    let s2 = s1;

    s1.display();
    s2.display();
}
```

```
21 |     let s1 = S(1);
    |         -- move occurs because `s1` has type `S`,
    |         which does not implement the `Copy` trait
22 |     let s2 = s1;
    |         -- value moved here
23 |
24 |     s1.display();
    |     ^^ value borrowed here after move
```



drop2.rs

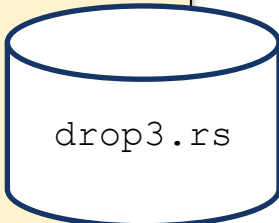
Movimento

```
struct S(i32); // struct che contiene un unico campo di tipo i32
impl S {
    fn display (&self) { // metodo chiamato su un oggetto di S
                        // ricevendolo come riferimento non mutabile
        println!("Sono S e contengo {} @ {:p}", self.0, self);
                        // mi dice che cosa contiene S e dove è memorizzato
    }
}
impl Drop for S {
    fn drop(&mut self) // ha la possibilità di manipolare il dato
    {
        println!("Dropping S({}) @ {:p}", self.0, self);
    }
}

fn main() {
    let s1 = S(1);
    s1.display();
    let mut s2 = s1;

    s2.display();

    s2.0 = 7;
    s2.display();
}
```



drop3.rs

```
Sono S e contengo 1 @ 0x5353b2f8ec
Sono S e contengo 1 @ 0x5353b2f8f0
Sono S e contengo 7 @ 0x5353b2f8f0
Dropping S(7) @ 0x5353b2f8f0
```

Puntatori nativi

- Rust definisce anche i tipi dei puntatori nativi come ***const T** e ***mut T**, per qualsiasi tipo T
 - ***const** è un puntatore ad un dato che non possiamo mutare
 - ***mut** è un puntatore ad un dato che possiamo modificare
 - Questi sono, a tutti gli effetti, equivalenti ai puntatori in C e C++ e ne condividono tutti i problemi
- Tuttavia, è possibile dereferenziarli (accedere al loro contenuto, in lettura e/o scrittura) solo all'interno di blocchi **unsafe { ... }**
 - Se un programma non fa uso di blocchi unsafe, o se quelli che sono usati contengono solo codice corretto, allora si può essere certi che l'esecuzione del programma non darà origine a comportamenti non definiti

Array

- Un array è una sequenza di oggetti **omogenei**, disposti **consecutivamente** nello stack
 - Un array ha una dimensione definita all'atto della sua creazione ed immutabile
- Si crea un array racchiudendo la sequenza dei suoi valori tra parentesi quadre
 - Un array ha tipo **[T; length]**, dove T è il tipo dei singoli elementi, length indica il numero dei valori contenuti
- La lunghezza di un array è immutabile
- Si accede al contenuto dell'array con la notazione **nome[index]**

```
let a: [i32; 5] = [1, 2, 3, 4, 5]; // a è un array di 5 interi

let b = [0; 5];                      // b è un array di 5 interi inizializzati a
0                                     // NOTARE il ; per distinguere le notazioni

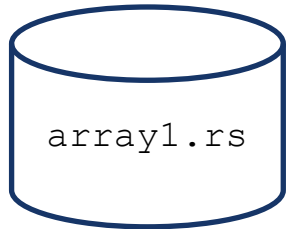
let l = b.len();                     // metodo len() calcola la lunghezza
                                     // l vale 5

let e = a[3];                        // e vale 4
```

Index Out of Bounds (compile time)

```
fn main() {  
    let v = [1,2,3,4];  
    println!("{}", v[1]);  
    println!("{}", v[10]);  
}
```

```
--> src\main.rs:9:20  
9 |         println!("{}", v[10]);  
  |                        ^^^^^ index out of bounds: the  
length is 4 but the index is 10
```

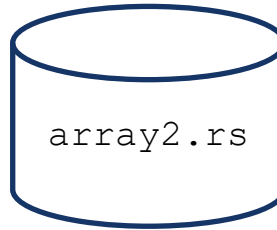


Index Out of Bounds (run time)

```
fn f (i: usize) -> usize
{
    return (i + 10)
}
fn main() {
    let v = [1,2,3,4];

    println!("{}", v[1]);
    println!("{}", v[f(0)]);
}
```

```
thread 'main' panicked at src\main.rs:13:20:
index out of bounds: the len is 4 but the index is 10
```



PANIC!

Slice

- Rust offre la possibilità di fare riferimento ad una sequenza di valori consecutivi la cui lunghezza non è nota in fase di compilazione, ma solo all'atto dell'esecuzione
 - Una slice di elementi di tipo T (scritto `&[T]`) è un tipo di dato formato da due valori consecutivi: il puntatore all'inizio della sequenza e il numero di elementi della sequenza
 - Per questa sua natura, viene detto *fat pointer*
- Si crea una slice come riferimento ad una porzione di un array o di un vec
 - `let a = [1, 2, 3, 4];`
 - `let s1: &[i32] = &a;` //s1 contiene i valori 1, 2, 3, 4
 - `let s2 = &a[0..2];` // s2 contiene i valori 1, 2
 - `let s3 = &a[2..];` // s3 contiene i valori 3, 4
- Di base, una slice è immutabile
 - Si acquisisce la possibilità di modificare il contenuto attraverso la notazione `let ms = &mut a[..];`
- Come nel caso degli array, si accede ai valori contenuti in una slice s con la notazione `s[i]`, dove i è un indice numerico privo di segno
 - Tentativi di accedere ad una posizione illecita comportano l'immediato arresto del programma (panic!)

```
fn main() {  
    let a = [ 1, 2, 3, 4 ];  
  
    let s2 = &a[0..2];  
  
    println!("array {:?}", a);  
    println!("slice {:?}", s2);  
}
```

slice1.rs

array [1, 2, 3, 4]
slice [1, 2]

```
fn main() {  
    let a = [ 1, 2, 3, 4 ];  
  
    let s2 = &mut a[0..2];  
  
    s2[0] = 10;  
  
    println!("slice {:?}", s2);  
}
```

slice2.rs

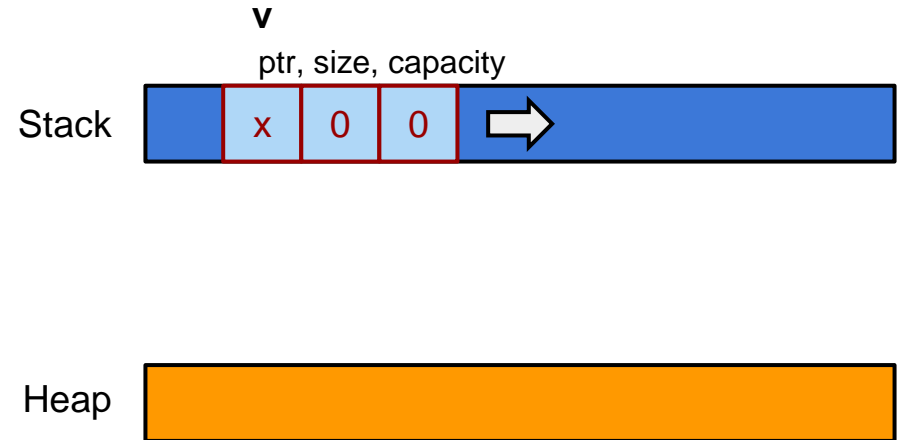
slice [10, 2]

Vec<T>

- Il tipo **Vec<T>** rappresenta una sequenza ridimensionabile di elementi di tipo **T**, allocati sullo heap
 - Offre una serie di metodi per accedere al suo contenuto e per inserire/togliere valori al suo interno
- Una variabile di tipo **Vec<T>** è una tupla formata da tre valori privati:
 - Un puntatore ad un buffer allocato sullo heap nel quale sono memorizzati gli elementi
 - Un intero privo di segno che indica la dimensione complessiva del buffer
 - Un intero privo di segno che indica quanti elementi sono valorizzati nel buffer
- Se si richiede ad un oggetto di tipo **Vec<T>** di inserire un nuovo elemento, questo verrà memorizzato nel buffer nella prima posizione libera
 - E verrà incrementato l'intero che indica il numero di elementi effettivamente presenti
- Nel caso in cui il buffer fosse già completo, verrà allocato un nuovo buffer di dimensioni maggiori
 - E il contenuto del buffer precedente sarà riversato in quello nuovo, dove verrà poi anche inserito il nuovo elemento
 - Dopodiché il buffer precedente sarà de-allocato

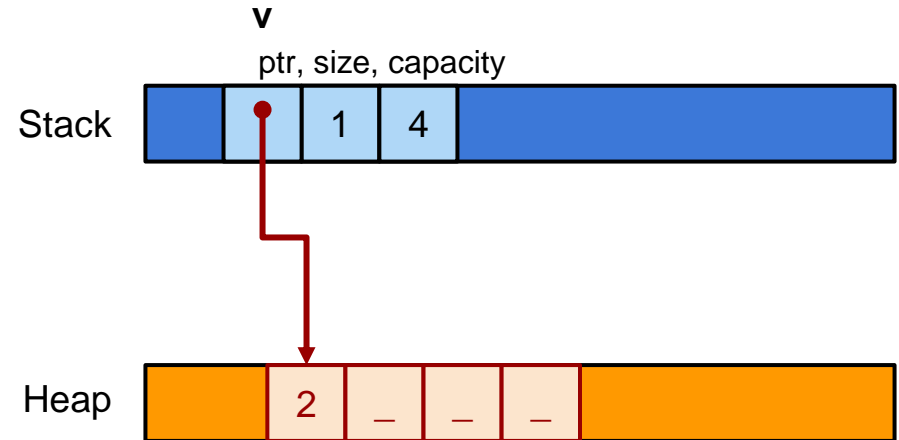
Vec<T>

```
fn useVec() {  
    let mut v: Vec<i32> = Vec::new();  
  
    v.push(2);  
  
    v.push(4);  
  
    let mut s = &mut v;  
  
    s[1] = 8;  
}
```



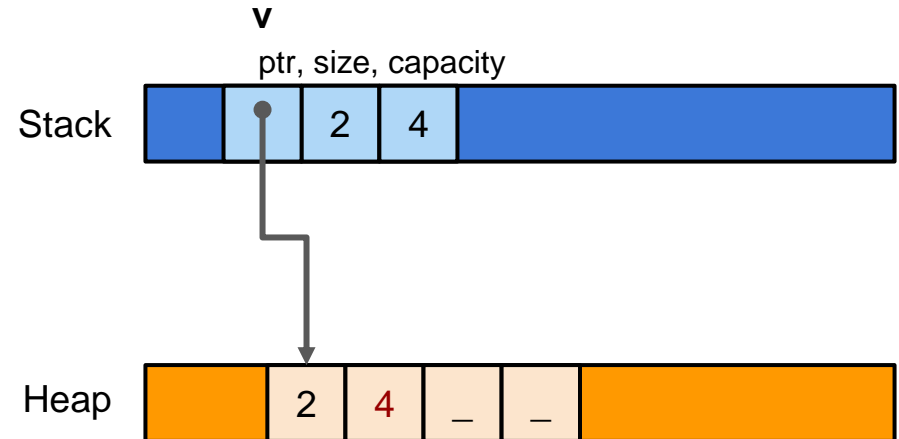
Vec<T>

```
fn useVec() {  
    let mut v: Vec<i32> = Vec::new();  
  
    v.push(2);  
  
    v.push(4);  
  
    let mut s = &mut v;  
  
    s[1] = 8;  
}
```



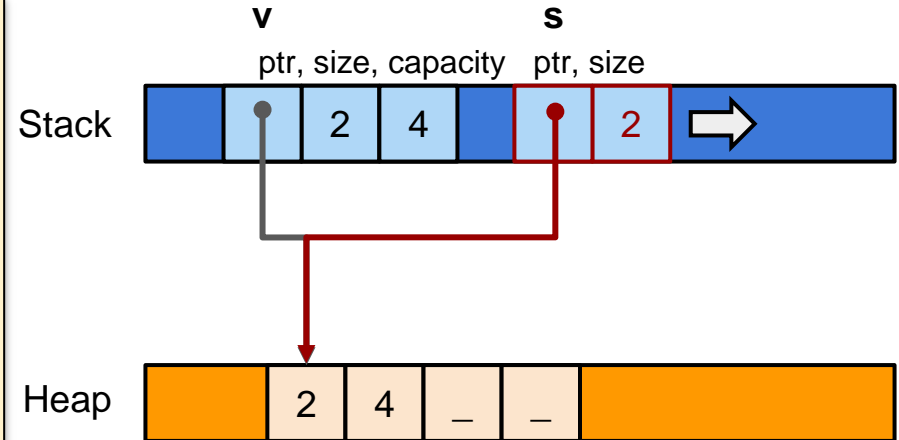
Vec<T>

```
fn useVec() {  
    let mut v: Vec<i32> = Vec::new();  
  
    v.push(2);  
  
    v.push(4);  
  
    let mut s = &mut v;  
  
    s[1] = 8;  
}
```



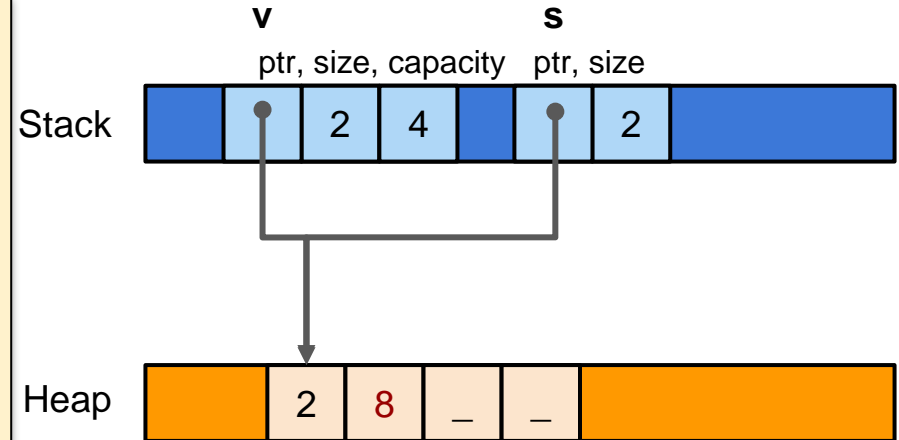
Vec<T>

```
fn useVec() {  
    let mut v: Vec<i32> = Vec::new();  
  
    v.push(2);  
  
    v.push(4);  
  
    let mut s = &mut v;  
  
    s[1] = 8;  
}
```



Vec<T>

```
fn useVec() {  
  let mut v: Vec<i32> = Vec::new();  
  
  v.push(2);  
  
  v.push(4);  
  
  let mut s = &mut v;  
  
  s[1] = 8;  
}
```



Vec<T>

```
fn useVec() {  
    let mut v: Vec<i32> = Vec::new();  
  
    v.push(2);  
  
    v.push(4);  
  
    let mut s = &mut v;  
  
    s[1] = 8;  
}
```

Stack



Heap



```
struct Test (i32);
```

```
impl Drop for Test {  
    fn drop (& mut self) {  
        println!("Destroying Test ({})) at address {:p}", self.0, self);  
    }  
}
```

```
fn main() {  
    let mut v =Vec::<Test>::with_capacity(4);  
    println!("ptr: {:p} {} {}", v.as_ptr(), v.capacity(), v.len());
```

```
    for i in 0..4  
    { v.push(Test(i)); }  
    println!("ptr: {:p} {} {}", v.as_ptr(), v.capacity(), v.len());
```

```
    v.push(Test(10));  
    println!("ptr: {:p} {} {}", v.as_ptr(), v.capacity(), v.len());
```

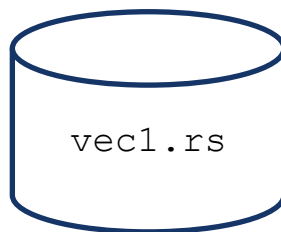
```
    for i in 20..25  
    { v.push(Test(i)); }
```

```
    println!("ptr: {:p} {} {}", v.as_ptr(), v.capacity(), v.len());
```

```
    v.pop();
```

```
    v.shrink_to_fit();
```

```
    println!("ptr: {:p} {} {}", v.as_ptr(), v.capacity(), v.len());  
}
```



ptr: 0x25f2fa90c30 4 0

ptr: 0x25f2fa90c30 4 4

ptr: 0x25f2fa8e1d0 8 5

ptr: 0x25f2fa8ec40 16 10

Destroying Test (24) at address 0xaa4c0ff664

ptr: 0x25f2fa8e1d0 9 9

Destroying Test (0) at address 0x25f2fa8e1d0

Destroying Test (1) at address 0x25f2fa8e1d4

Destroying Test (2) at address 0x25f2fa8e1d8

Destroying Test (3) at address 0x25f2fa8e1dc

Destroying Test (10) at address 0x25f2fa8e1e0

Destroying Test (20) at address 0x25f2fa8e1e4

Destroying Test (21) at address 0x25f2fa8e1e8

Destroying Test (22) at address 0x25f2fa8e1ec

Destroying Test (23) at address 0x25f2fa8e1f0

```

struct Test (i32);

impl Drop for Test {
    fn drop (& mut self) {
        println!("Destroying Test ({})) at address {:p}", self.0, self);
    }
}

fn main() {
    let mut v =Vec::<Test>::with_capacity(4);
    println!("ptr: {:p} {} {}", v.as_ptr(), v.capacity(), v.len());

    for i in 0..4
    { v.push(Test(i)); }
    println!("ptr: {:p} {} {}", v.as_ptr(), v.capacity(), v.len());

    v.push(Test(10));
    println!("ptr: {:p} {} {}", v.as_ptr(), v.capacity(), v.len());

    for i in 20..25
    { v.push(Test(i));}

    println!("ptr: {:p} {} {}", v.as_ptr(), v.capacity(), v.len());

    v.pop();

    v.shrink_to_fit();
    println!("ptr: {:p} {} {}", v.as_ptr(), v.capacity(), v.len());
}

```

```

ptr: 0x25f2fa90c30 4 0
ptr: 0x25f2fa90c30 4 4
ptr: 0x25f2fa8e1d0 8 5
ptr: 0x25f2fa8ec40 16 10
Destroying Test (24) at address 0xaa4c0ff664
ptr: 0x25f2fa8e1d0 9 9

```

Indirizzo strano, corrispondente allo stack, dove viene copiato il dato (corrispondente al valore di ritorno della pop) che viene salvato temporaneamente nello stack prima di eliminarlo definitivamente

```

Destroying Test (21) at address 0x25f2fa8e1c0
Destroying Test (22) at address 0x25f2fa8e1ec
Destroying Test (23) at address 0x25f2fa8e1f0

```

```
#[derive(Debug)]
struct Test (i32);
impl Drop for Test {
    fn drop (& mut self) {
        println!("Destroying Test ({} ) at address {:p}", self.0, self);
    }
}

fn main() {
    let mut v =Vec::<Test>::new();
    println!("ptr: {:p} {} {} ", v.as_ptr(), v.capacity(), v.len());

    v.push(Test(0));

    println!("ptr: {:p} {} {} ", v.as_ptr(), v.capacity(), v.len());
    println!("&v[0]: {:p}", &v[0]);

    v.push(Test(1));
    v.push(Test(2));
    v.push(Test(3));
    println!("ptr: {:p} {} {} ", v.as_ptr(), v.capacity(), v.len());

    v.push(Test(4));

    println!("ptr: {:p} {} {} ", v.as_ptr(), v.capacity(), v.len());

    v.shrink_to_fit();
    println!("Shrinking ptr: {:p} {} {} ", v.as_ptr(), v.capacity(), v.len());

    let a = v.remove(1);
    println!("Removing ptr: {:p} {} {} ", v.as_ptr(), v.capacity(), v.len());
    println!("Removed {:?} ", a);

    println!("Terminating");
}
```



```
ptr: 0x4 0 0
ptr: 0x1e4f5cddf60 4 1
&v[0]: 0x1e4f5cddf60
ptr: 0x1e4f5cddf60 4 4
ptr: 0x1e4f5cdec0 8 5
Shrinking ptr: 0x1e4f5cde080 5 5
Removing ptr: 0x1e4f5cde080 5 4
Removed Test(1)
Terminating
Destroying Test (1) at address 0x9b774ff61c
Destroying Test (0) at address 0x1e4f5cde080
Destroying Test (2) at address 0x1e4f5cde084
Destroying Test (3) at address 0x1e4f5cde088
Destroying Test (4) at address 0x1e4f5cde08c
```

Stringhe

- Rust offre due modi principali di rappresentare le stringhe con rappresentazione Unicode UTF-8:
 - Come array di caratteri (**immutabili**), memorizzati in un'area statica, rappresentato dal tipo primitivo **str**
 - Come oggetti allocati dinamicamente (**mutabili**), utilizzando il tipo **String**

Costanti di tipo stringa

- Le **costanti** di tipo stringa presenti nel codice sorgente sono racchiuse tra doppi apici ""
 - Il compilatore provvede ad inserirle in un'apposita **area statica di memoria**, in modo compatto, senza aggiungere alcun terminatore
 - hanno tempo di vita per tutta la durata del programma.

```
let s = "Ciao Mamma";
```

str

- Poiché il tipo primitivo **str** non è direttamente manipolabile, si accede ad esso **solo tramite uno slice**, di tipo **&str**
 - Esso contiene l'indirizzo del primo carattere e la lunghezza della stringa

```
let s = "Ciao Mamma";
```



```
let s: &str = "Ciao Mamma";
```

Stringhe

- Gli oggetti di tipo **String** contengono:
 - un puntatore ad un buffer allocato dinamicamente,
 - l'effettiva lunghezza della stringa
 - la capacità del buffer
- Se la stringa è mutabile e vengono inseriti al suo interno più caratteri di quelli che il buffer può contenere, il buffer viene automaticamente allocato nuovamente con una capacità maggiore, così da ospitare quanto richiesto
- Tutti i metodi che sono leciti su un oggetto di tipo **&str** sono anche disponibili per **&String**
 - Inoltre, se una funzione accetta un parametro di tipo **&str**, è possibile passare come argomento corrispondente il riferimento ad un oggetto **String**

Stringhe

```
fn main() {
```

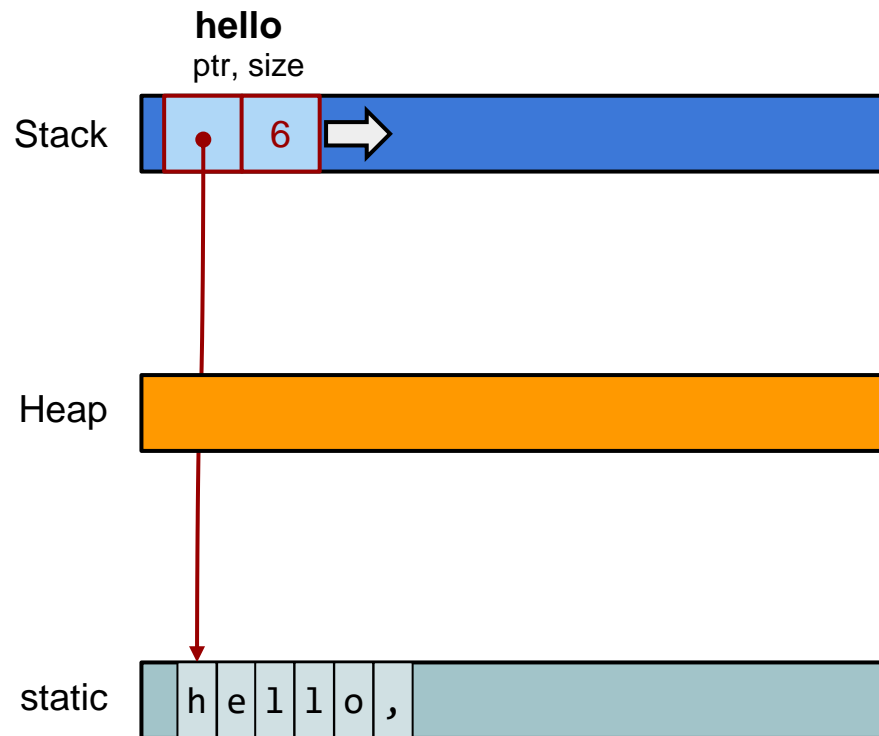
```
    let hello: &str = "hello,";
```

```
    let mut s = String::new();
```

```
    s.push_str(hello);
```

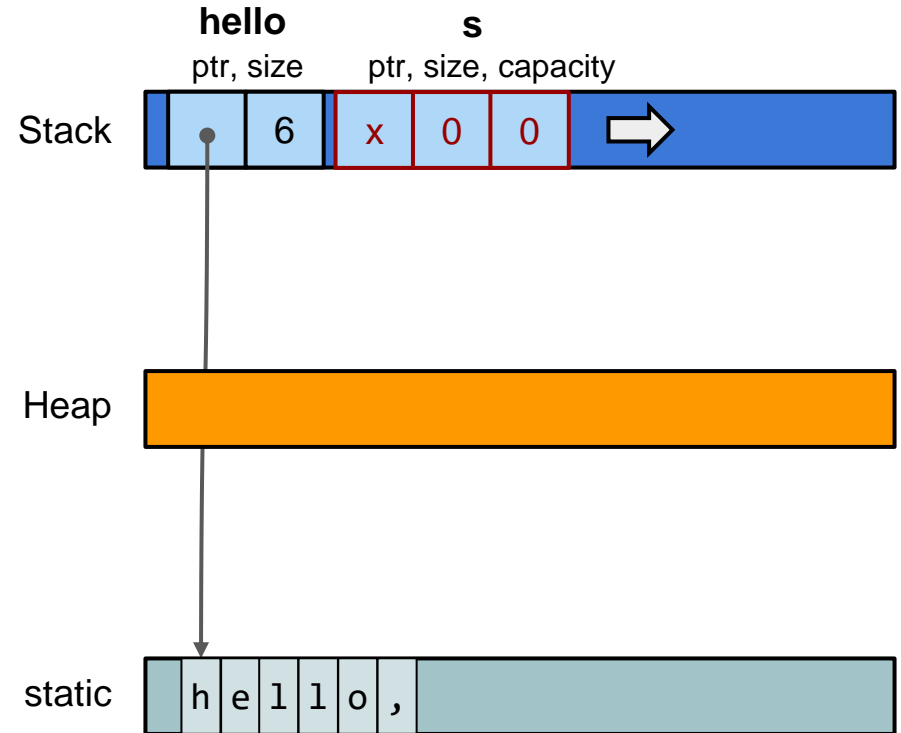
```
    s.push_str(" world!");
```

```
}
```



Stringhe

```
fn main() {  
  
    let hello: &str = "hello,";  
  
    let mut s = String::new();  
  
    s.push_str(hello);  
  
    s.push_str(" world!");  
  
}
```

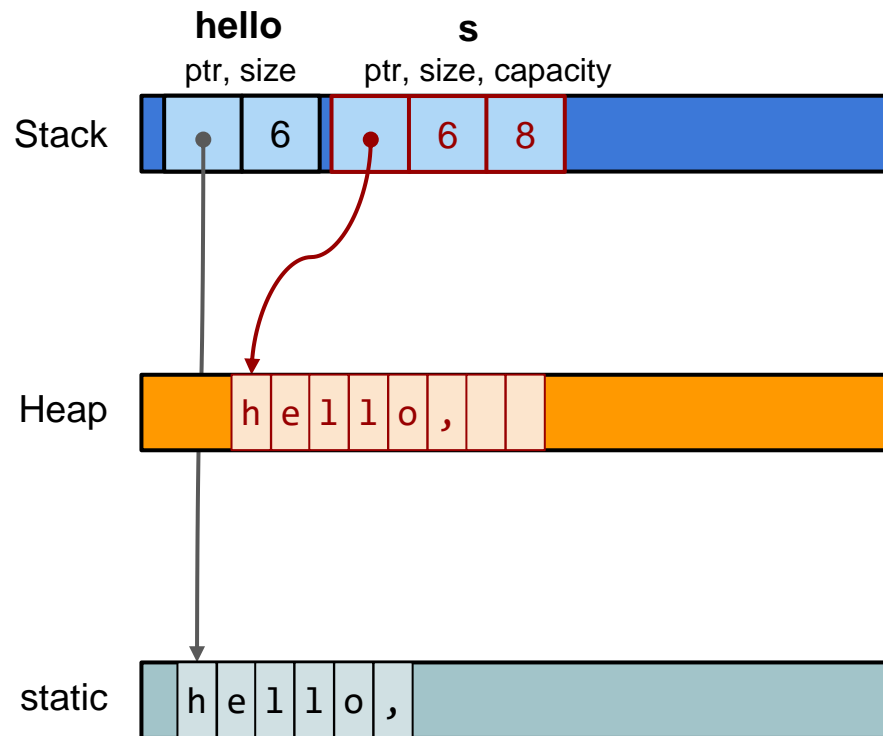


Stringhe

```
fn main() {  
  
    let hello: &str = "hello,";  
  
    let mut s = String::new();  
  
    s.push_str(hello);  
  
    s.push_str(" world!");  
  
}
```

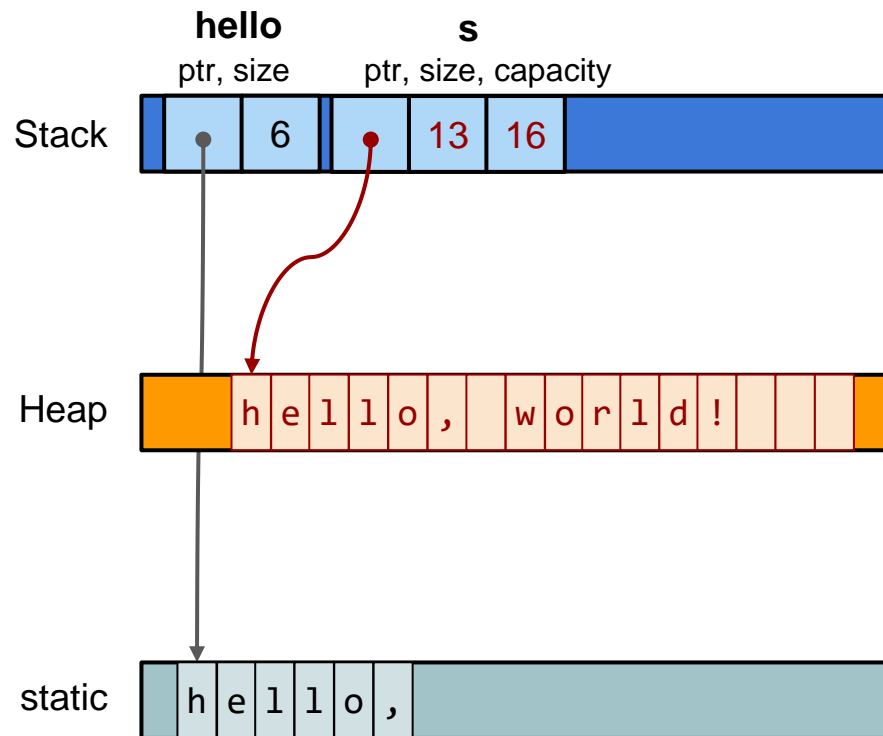
Tipo String è mutabile e posso invocare i metodi che cambiano il contenuto di una variabile di tipo String

.push_str() aggiunge al fondo della stringa s lo slice passato come parametro.



Stringhe

```
fn main() {  
  
    let hello: &str = "hello,";  
  
    let mut s = String::new();  
  
    s.push_str(hello);  
  
    s.push_str(" world!");  
  
}
```



Stringhe

```
fn main() {  
  
    let hello: &str = "hello,";  
  
    let mut s = String::new();  
  
    s.push_str(hello);  
  
    s.push_str(" world!");  
  
}
```

Stack 

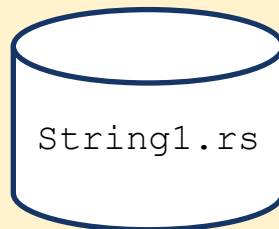
Heap 

static 

Stringhe

- Si crea un oggetto **String** con le istruzioni
 - `let s0 = String::new();` //crea una stringa vuota
 - `let s1 = String::from("some text");` //crea una stringa inizializzata
 - `let s2 = "some text".to_string();` //equivalente al precedente
- Si ricava un oggetto di tipo `&str` da un oggetto `String` con il metodo
 - `s2.as_str();`

```
fn main() {  
    let s1 = String::from("hello");  
    let mystr = s1.as_str();  
  
    let len = calculate_length(mystr);  
  
    println!("The length of '{}' is {}.", mystr, len);  
}  
fn calculate_length(s: &str) -> usize {  
    let length = s.len();  
    length  
}
```



Stringhe

- Un oggetto String (se mutabile) può essere modificato
 - `s3.push_str("This goes to the end");` // aggiunge al fondo
 - `s3.insert_str(0, "This goes to the front");` // inserisce alla posizione data
 - `s3.remove(4);` // elimina il carattere alla posizione indicata
 - `s3.clear();` // svuota la stringa
- In altri casi si può costruire un altro oggetto String
 - `let s4 = s1.to_uppercase();` // forza il maiuscolo (ATTENZIONE alla lingua!)
 - `let s5 = s1.replace("some", " more ");` // sostituisce un blocco
 - `let s6 = s1.trim();` // elimina spaziature iniziali e finali

```
fn main() {  
    let mut s = "ciao mamma".to_string();  
    s.push_str(" come stai");  
    let mut s2 = s.replace("ciao", "Ciao");  
    s2.insert(10, ',');  
  
    println!("{}", s2);  
  
    let s = "tschüß".to_string();  
    let s2 = s.to_uppercase();  
  
    println!("{}", s2);  
}
```

String2.rs

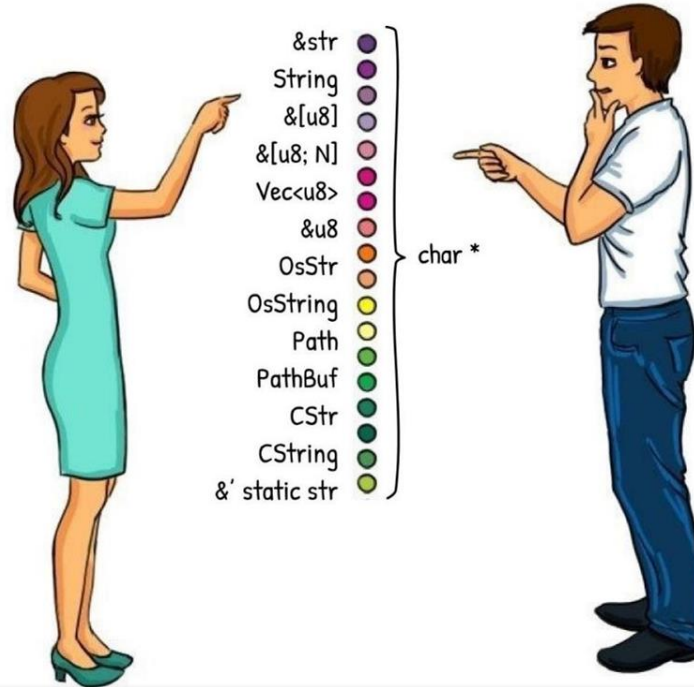
Ciao mamma, come stai
TSCHÜSS

Stringhe

How We See Strings

Rust

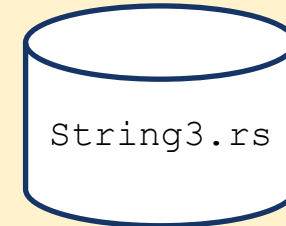
C



<https://www.programmersought.com/article/41316303245/>

Str e String

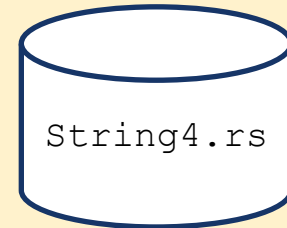
```
fn main() {  
    let name1 = "Matteo";  
    let name2 = "Giovanni".to_string();  
  
    greet(name1);  
    greet(&name2);  
}  
  
fn greet(name: &str) {  
    println!("Hello, {}!", name);  
}
```



Slice di byte: `&[u8]`

- Posso memorizzare un array di caratteri in una slice di byte
- `&[u8]` accetta qualunque combinazione dei valori dei byte
- `&str` accetta solo valori UTF-8.

```
fn main() {  
    let name: &[u8] = b"Torino";  
  
    println!("{:?}", name);  
}
```



La lettera `b` che precede un stringa informa il compilatore che la stringa deve essere trattata come sequenza di byte.

Ref di array statico: `&[u8; N]`

- Posso memorizzare un array di caratteri in un array statico di N byte
- Non tutte le combinazioni di byte corrispondono a caratteri.

```
fn main() {  
    let name: &[u8; 6] = b"Torino";  
  
    println!("{:?}", name);  
}
```



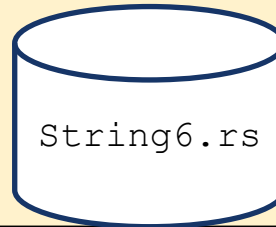
Conversione da array di byte a &str

```
use std::str;

fn main() {
    let bytes_array: [u8; 5] = [104, 101, 108, 108, 111];
    // Array di byte ASCII per "hello"

    let stringa = str::from_utf8(&bytes_array) ;

    println!("{:?}", stringa);
}
```



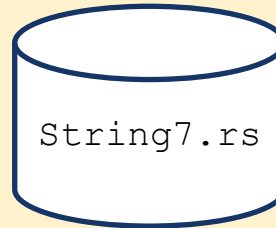
Ok("hello")

Vec<u8>

- Esiste una forte analogia tra una String ed un Vec<u8>

```
fn main() {  
    let my_vec: Vec<u8> = vec![72, 101, 108, 108, 111, 32, 87, 111, 114, 108, 100];  
    // "Hello World" in ASCII  
  
    let vec_to_string = String::from_utf8(my_vec).unwrap();  
    // Converting to string  
  
    println!("{}", vec_to_string);  
}
```

Hello World



OsStr e OsString

- Quando è necessario interagire con il Sistema Operativo, i tipi `&str` e `String` possono essere convertiti nei tipi specifici della piattaforma che si chiamano `OsStr` (vista) e `OsString` (oggetto mutabile), codificato secondo le specifiche dell'API del Sistema Operativo:
 - Linux utilizza UTF-8
 - Windows codifica i caratteri su 16 bit, con la codifica Unicode Basic Multilingual Plane 0.

```
use std::ffi::OsString;

fn main() {
    let stringa = String::from("Hello");

    // Converti la Stringa in un OsString utilizzando il metodo into()
    let os_string: OsString = stringa.into();

    println!("{:?}", os_string);
}
```

"Hello"

String8.rs

Path e PathBuf

- Il path identifica la collocazione di un file nel File System, con la rappresentazione del cammino dalla radice al file con un separatore che in Linux è / (slash) e in Windows è \ (back slash).
- Path (vista) e PathBuf (mutabile) rappresentano una stringa adatta a rappresentare un path

```
use std::path::Path;

fn main() {
    let percorso_str = "/cartella/file.txt";
    let percorso = Path::new(percorso_str);

    // Ora puoi utilizzare il percorso
    println!("Il percorso è: {:?}", percorso);
}
```

Il percorso è: "/cartella/file.txt"

String9.rs

CStr e CString

- CStr e CString ci garantiscono l'interoperabilità con il linguaggio C.
- Il linguaggio C ha bisogno che ci sia lo 0 finale. Rust non mette lo 0 finale perché tenendosi la lunghezza non ha bisogno dello 0 finale. Se abbiamo bisogno di collegare una libreria C e se quella libreria ha bisogno di stringhe dovremo convertire prima le nostre Stringhe RUST nei corrispondenti CStr e CString.

```
use std::ffi::CString;

fn main() {
    let ss = "Hello".to_string();
    let s = CString::new(ss).unwrap();
    println!("{:?}", s);
}
```

"Hello"

String10.rs

&'static str

- Permette di allocare una stringa per tutta la durata del programma.

```
fn main() {  
    let static_string: &'static str = "Hello, world!";  
    println!("{}", static_string);  
}
```

"Hello, world"

String11.rs

Qualche spunto

- <https://stackoverflow.com/questions/41034635/how-do-i-convert-between-string-str-vecu8-and-u8>
- <https://www.programmersought.com/article/41316303245/>
- <https://profpatsch.de/notes/rust-string-conversions>

Istruzioni ed espressioni

- Il corpo di una funzione è costituito da istruzioni e/o espressioni separate da **;**
 - Una istruzione ha come tipo di ritorno **()**, un'espressione può restituire un tipo arbitrario
- I costrutti **let ...** e **let mut ...** sono istruzioni
 - Creano un legame tra la variabile indicata ed il valore assegnato
- Un blocco racchiuso tra **{...}** è un'espressione
 - Restituisce il valore corrispondente all'ultima espressione, a condizione che non sia terminata da **;**
- Il costrutto **if ... else ...** è un'espressione
 - Il ramo positivo ed il ramo negativo sono costituiti da blocchi che devono restituire lo stesso tipo di dato
- Il costrutto **loop ...** è un'espressione
 - Crea un iterazione infinita che può essere interrotta eseguendo l'istruzione **break** seguita dal valore di ritorno (se presente)
 - Una singola iterazione può essere parzialmente saltata eseguendo l'istruzione **continue** (come in C).

Blocco come espressione

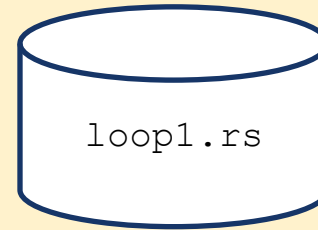
```
fn main() {  
    let y = {  
        let x = 3;  
        x + 1      // espressione  
    };  
  
    println!("The value of y is: {y}");  
}
```

if come espressione

```
fn main() {  
    let condition = true;  
    let number = if condition { 5 } else { 6 };  
  
    println!("The value of number is: {number}");  
}
```

Loop

```
fn main() {  
    let mut counter = 0;  
    let mut sum = 0;  
  
    let result = loop {  
        counter += 1;  
        sum += counter;  
  
        if counter == 10 {  
            break sum * 2;  
        }  
    };  
  
    println!("The result is {result}");  
}
```



Funzioni

- Costituiscono il nucleo principale attorno al quale viene definito il comportamento di un programma
 - Una funzione è introdotta dalla parola chiave `fn` seguita dal nome e dalla lista di argomenti, ciascuno con il relativo tipo, racchiusa tra parentesi tonde
 - Se ritorna un valore diverso da `()`, la lista degli argomenti è seguita dal simbolo `->` e dal tipo ritornato
 - Il corpo della funzione è racchiuso tra `{ }` ed è composto da istruzioni
- L'ultima espressione presente nel corpo, se priva di `';` finale, viene interpretata come valore di ritorno
 - In alternativa, è possibile utilizzare l'istruzione `return` seguita dal valore e da `;`

```
fn print_number(x: i32) /* -> () */ {  
    println!("x is: {}", x);  
}
```

```
fn add_numbers(x: i32, y: i32) -> i32 {  
    x + y // NON c'è il ; finale  
}
```

Istruzioni ed espressioni



```
fn find_number(n: i32) -> i32 {  
    let mut count = 0;  
    let mut sum = 0;  
    loop {  
        count += 1;  
        if count % 5 == 0 { continue; }           // ignora i multipli di 5  
        sum += if count % 3 == 0 { 1 } else { 0 }; // conta i multipli di 3  
        if sum == n { break; }                   // fermati al n° multiplo di 3  
                                                // ma non multiplo di 5  
    }  
    count                                         // restituisce il valore trovato  
}  
  
fn main() {  
    println!("{}", find_number(5) );             // invocazione della funzione  
}
```

Loop annidati

- E' possibile annidare più costrutti di tipo loop ed interrompere o continuare un particolare livello di annidamento, facendo precedere l'istruzione loop da un'etichetta
 - L'etichetta è un identificatore preceduto da '
 - Le istruzioni **break** e **continue** possono indicare l'etichetta cui fanno riferimento

Loop annidati (II)

```
fn main() {  
    'outer: loop {  
        println!("Entrato nel ciclo esterno");  
  
        'inner: loop {  
            println!("Entrato nel ciclo interno");  
  
            // La prossima istruzione interromperebbe il ciclo interno  
            //break;  
  
            // Così si interrompe il ciclo esterno  
            break 'outer;  
        }  
        //Il programma non raggiunge mai questa posizione  
    }  
    println!("Terminato il ciclo esterno");  
}
```

```
fn main() {  
    let mut count = 0;  
    'counting_up: loop {  
        println!("count = {count}");  
        let mut remaining = 10;  
  
        loop {  
            println!("remaining = {remaining}");  
            if remaining == 9 {  
                break;  
            }  
            if count == 2 {  
                break 'counting_up;  
            }  
            remaining -= 1;  
        }  
        count += 1;  
    }  
}
```



while

- L'istruzione **while** ... permette di subordinare l'esecuzione del ciclo al verificarsi di una condizione
 - In modo analogo a quanto avviene in altri linguaggi

```
use std::time::{Duration, Instant};    // Importa dalla libreria standard

fn main() {
    let mut counter = 0;

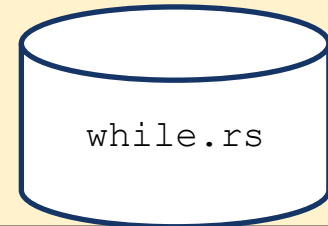
    let time_limit = Duration::new(1,0); // Crea una durata di 1 secondo

    let start = Instant::now();           // Determina l'ora attuale

    while (Instant::now() - start) < time_limit { // Finché non è passato 1 s...
        counter += 1;                       // ...incrementa il contatore
    }

    println!("{}", counter);

}
```



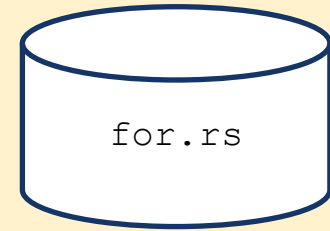
for

- L'istruzione **for** ... ha una sintassi particolare:
 - **for** *var* **in** *expression* { *code* }
 - *expression* deve restituire un valore che sia (o possa essere convertito in) un iteratore: sono leciti, ad esempio, *array*, *slice* e *range* (nella forma *Low*..*high*)

Intervalli

- Le notazioni **a..b** e **c..=d** indicano, rispettivamente, un intervallo semi-aperto e un intervallo chiuso
 - Possono essere usati in senso generale, riferendosi al dominio del tipo della variabile
 - Oppure possono essere applicati ad una slice, riferendosi all'insieme dei valori leciti
- Sono possibili diverse combinazioni
 - **..** indica tutti i valori possibili per un dato dominio
 - **a..** indica tutti i valori a partire da **a** (incluso)
 - **..b** indica tutti i valori fino a **b** (escluso)
 - **..=c** indica tutti i valori fino a **c** (incluso)
 - **d..e** indica tutti i valori tra **d** (incluso) ed **e** (escluso)
 - **f..=g** indica tutti i valori tra **f** e **g** (inclusi)

```
fn main() {
    let a = [10, 20, 30, 40, 50];
    for element in a {
        println!("the value is: {element}");
    }
```



```
for n in 1..10 {
    println!("{}", n);
}
```

// Stampa i numeri da 1 a 9

```
let names = ["Bob", "Frank", "Ferris"];
for name in names.iter() {
    println!("{}", name);
}
```

// Stampa i tre nomi

```
for name in &names[ ..=1 ] {
    println!("{}", name);
}
```

// Stampa i primi due nomi

```
for number in (1..=10).rev() {
    println!("{number}!");
}
```

// Stampa una sequenza al contrario

```
for (i,n) in names.iter().enumerate() {
    println!("names[{}]: {}", i, n);
}
```

Istruzioni ed espressioni

- L'espressione **match** ... permette di eseguire in modo condizionale blocchi di codice confrontando un valore con una serie di pattern alternativi
 - Essa confronta la **struttura** del valore con i singoli pattern indicati
 - Tali pattern possono contenere **variabili**, che - in caso di corrispondenza delle parti costanti - vengono legate al corrispondente frammento del valore confrontato
 - L'elenco dei pattern deve essere **esaustivo** del dominio dell'espressione
- Ciascun pattern è separato dal blocco di codice da eseguire dal simbolo =>
 - il pattern può essere annotato con una clausola **if** ... per limitarne l'applicabilità
 - I diversi rami sono separati da ,
 - Le espressioni di confronto contenute nel pattern possono essere annotate con un identificatore seguito da @, per legare il valore confrontato al nome dato, così da poter fare riferimento ad esso nel blocco corrispondente

Istruzioni ed espressioni

- L'espressione **match** offre una sintassi concisa e sofisticata per confrontare valori multipli così come per estrarre valori da tipi complessi
 - Per indicare un singolo valore, non occorre nessun operatore
 - la sintassi **val₁ ..= val₂** indica un intervallo chiuso
 - Una barra verticale singola **|** può essere usata per indicare una disgiunzione (or)
 - Il segno di sottolineatura **_** corrisponde a qualsiasi valore
- I pattern sono valutati nell'ordine indicato
 - Alla prima corrispondenza, viene valutato il blocco associato, il cui valore diventa il valore dell'espressione complessiva

```
let s = match item {  
  0 => "zero",                // valore singolo  
  10 ..= 20 => "tra dieci e venti", // intervallo inclusivo  
  40 | 80 => "quaranta o ottanta", // alternativa  
  _ => "altro",               // qualunque cosa  
}
```

```
fn main() {  
  
    for index in 0 .. 10 {  
        println!("Index:{}", index);  
        let s: &str = match index {  
            0 ..= 4 => { "\tI'm in the first half" },  
            5 => { "\tI'm in the middle" },  
            _ => { "\tI'm in the second half..." }  
        };  
        println!("{}", s);  
    }  
}
```



```
fn main() {  
    let values = [1, 2, 3];  
  
    match &values[..] {      // crea una slice con tutti gli elementi  
        // Contiene almeno un elemento, il primo valore è 0  
        &[0, ..] => println!("Comincia con 0"),  
        // Contiene almeno un elemento, l'ultimo valore è compreso tra 3 e 5  
        &[.., v @ 3..=5] => println!("Finisce con {}", v),  
        // Contiene almeno due elementi  
        &[_ , v, ..] => println!("Il secondo valore è {}", v),  
        // Contiene un solo elemento  
        &[v] => println!("Ha un solo elemento: {}", v),  
        // Non contiene elementi  
        &[] => println!("E' vuoto")  
    }  
}
```

Riga di comando

- Analogamente al C/C++, anche Rust consente di progettare eseguibili che adattano il comportamento in base ai parametri passati sulla riga di comando
- Tali parametri si trovano dentro il contenitore `std::env::args`
 - Composto da valori di tipo `String`
 - Nessun bisogno di `argc`: `args.len()` ritorna il numero di parametri
 - Si può accedere come un classico `argv` del C/C++ con gli indici
 - Sfruttabile anche con altri costrutti più evoluti

```
use std::env::args;
fn main() {
    let args: Vec<String> = args().skip(1).collect();
        if args.len() > 0 { // we have args!
            println!("{}", args[0]);
        }
}
```



Riga di comando

- La libreria **clap** gestisce in modo dichiarativo i parametri passati attraverso la linea di comando
 - La si include in un crate aggiungendo, nel file Cargo.toml, una dipendenza del tipo **[dependencies]**
clap = { version= "4.5.3", features = ["derive"] }
 - Questo mette a disposizione un insieme di macro e di strutture dati che permettono di descrivere una struttura dati in cui verranno depositati i valori estratti dalla riga di comando, così come di derivare automaticamente una funzione di analisi che provvede a valorizzare i campi di tale struttura
- Questa libreria permette anche di esprimere programmaticamente l'insieme dei parametri, la tipologia di valori associati e gli eventuali vincoli associati
 - Basata sul pattern "builder"

Riga di comando

```
use clap::Parser;

/// Simple program to greet a person
#[derive(Parser, Debug)]
#[command(version, long_about = None)]
struct Args {
    /// Name of the person to greet
    #[arg(short, long)]
    name: String,
    /// Number of times to greet
    #[arg(short, long, default_value_t = 1)]
    count: u8,
}

fn main() {
    let args = Args::parse();
    for _ in 0..args.count {
        println!("Hello {}!", args.name)
    }
}
```

parse.rs

```
$ demo --help
```

Simple program to greet a person

Usage: demo[EXE] [OPTIONS] --name <NAME>

Options:

-n, --name <NAME>	Name of the person to greet
-c, --count <COUNT>	Number of times to greet [default: 1]
-h, --help	Print help
-V, --version	Print version

```
$ demo --name Me
```

Hello Me!

I/O da console

- Il crate (package) **std::io** contiene la definizione delle strutture dati per accedere i flussi standard di ingresso/uscita
 - Questo tipo di operazioni, per definizione, possono fallire: di conseguenza tutti i metodi offerti restituiscono un oggetto di tipo **Result<T,Error>** che incapsula, alternativamente, il valore atteso, se l'operazione ha avuto successo, o un oggetto di tipo **Error**, in caso di fallimento
- Per garantire la correttezza del programma, occorre gestire esplicitamente l'eventuale errore, verificando il contenuto del valore ritornato tramite il metodo **is_ok()**
 - Oppure causare l'interruzione forzata del programma in caso di errore, utilizzando il metodo **unwrap()** che restituisce, se non c'è stato errore, il valore incapsulato
- Per semplificare le operazioni di scrittura, sono disponibili due macro
 - **print!(...)** e **println!(...)**
 - Entrambe accettano una stringa di formato e una serie di parametri da stampare

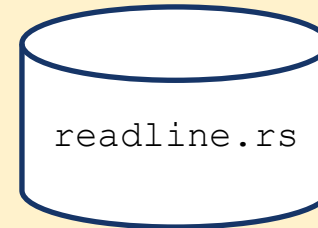
I/O da console

```
use std::io;

fn main() {
    let mut s = String::new();

    if io::stdin().read_line(&mut s).is_ok() {
        println!("Got {}", s.trim() );
    } else {
        println!("Failed to read line!");
    }

    //alternativamnte
    io::stdin().read_line(&mut s).unwrap();
    println!("Got {}", s.trim() );
}
```



Convenzioni sui nomi

- La comunità degli sviluppatori Rust ha elaborato una serie di regole sul formato dei nomi delle diverse entità del linguaggio
 - Si usano nomi nel formato UpperCamelCase per tutti i costrutti legati al sistema dei tipi (struct, enum, tratti, ...)
 - Si usano nomi nel formato snake_case per i costrutti di tipo valore (variabili, funzioni, metodi, ...)
- Alcune regole che generano warning possono essere disabilitate usando la sintassi con # (simile al pragma del C/C++):
 - `#[allow(non_snake_case)]` (vicino alla variabile per cui si vuole accettare un nome non snake)
 - `#![allow(non_snake_case)]` (all'inizio del file per applicare la regola a tutto il crate: notare il ! iniziale)

Esercizio

- Scrivere un programma che riceva da tastiera un testo ed estraiga la prima parola.