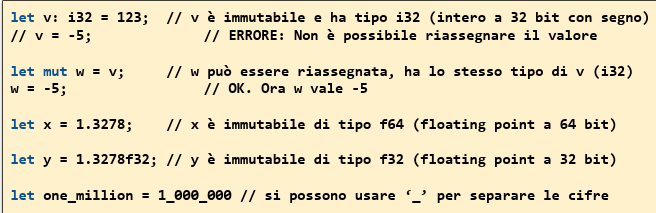
# LINGUAGGIO RUST

Una **variabile** lega un vale ad un nome, introdotta con **let**:

* *let i =25*; → variabile immutabile
* let **mut** i=25 → variabile **mutual**mente esclusiva (in valore e non tipo)
  + mentre qualcuno cambia il valore della variabile, gli altri non possono guardarla o usarla   
     → uno solo alla volta può toccarla

**NOTA**: tipo non si può cambiare.  
**NOTA**: se si ridefinisce una variabile, la nuova sovrascrive la vecchia.

* let v**: i32** = 123; 🡪 dichiaro **specificando il tipo** intero a 32 bit immutabile

### **Valori ed espressioni**

Un’espressione è un costrutto sintattico la cui esecuzione produce un valore di un dato tipo

* 4 + ( 3 \* 2 )
* Tutte le espressioni producono un valore che ha un tipo

### **Tipi e tratti**

#### **TIPI**

#### **1.Tipi elementari**

* + **i** → interi con segno
    - i8, i16, i32, i64, i128, isize
  + **u** → interi senza segno
    - u8, u16, u32, u64, u128, usize
  + **f** → floating point
    - f32 → singola precisione
    - f64 → doppia precisione
  + **bool** → logici
  + **char** → caratteri 32 bit, unicode
  + **()** → Unit
    - **()** rappresenta una tupla da 0 elementi

🡪 insieme delle funzioni che non ritornano un valore esplicito

* + *Per le* ***stringhe*** *si usa unicode UTF8 (8-16-32) → scrivo in base a come serve*
    - *La stringlength a questo punto non funziona più*

#### **2.Tuple:**

collezioni ordinate di valori ± omogenei → **eterogenei**

* Valori racchiusi tra parentesi tonde

Immagine che contiene testo, ricevuta, Carattere, schermata

Descrizione generata automaticamentemodo più semplice per realizzare un dato custom:

* *Latitudine + longitudine*
* *Voto + eventuale lode*

Si accede ai vari cambi della tupla in modo posizionale:

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

* Nome.0
* Nome.1
* Nome.2

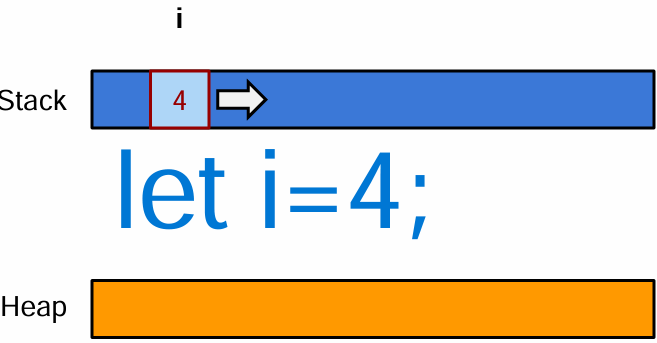
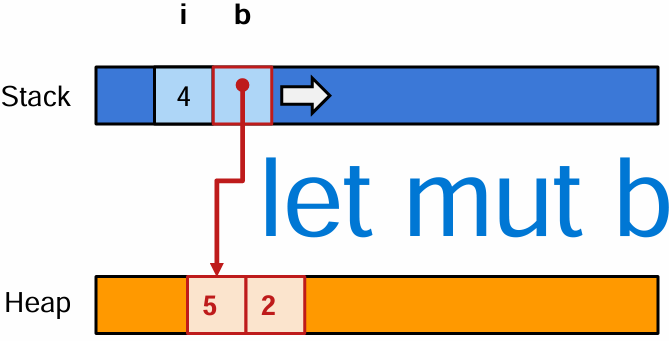
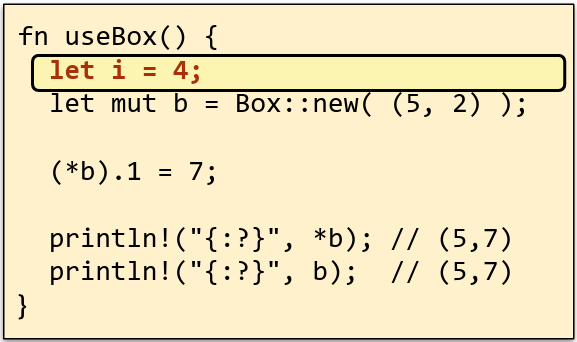
#### **3.Puntatori:**

vari modi per rappresentare indirizzi in memoria:

* **REF** → non copia il valore, ma gli da un puntatore
  + Riferimento **in sola lettura** ( accedo al valore con \*r1)
    - Puntatore senza possesso
  + Nr limitati
  + **Finché esiste un riferimento ed è ancora in uso, nessuno può metterci le mani**
    - Il possessore non la cambia perché c’è qualcuno che sta guardando
    - L’utilizzatore non cambia perché non ha i diritti.
  + Compilatore traccia tempo di vita delle variabili e il possesso
    - Garantisce che l’originale viva più a lungo del riferimento (evita dangling pointer)
* **REF MUT** 🡪 applicabile solo a delle variabili di tipo mut
  + Riferimento mutualmente esclusivo
    - Riferimento in **lettura/scritturra**
    - Max 1
    - Chi riceve può modificare perché ha il possesso temporaneo (\*r2= …)
    - Mentre esiste un ref mut:
      * l’originale è inaccessibile
        + non posso leggerlo
        + non posso crearci altri ref/mut
  + Puntatori privi di possesso
  + Responsabilità di rilascio è nelle mani dell’originale
* **BOX** 🡪 Puntatori che possiedono la memoria, punta ad un dato che sta sullo heap che possiede il valore
  + ****Per dati di cui **non si conosce** a priori la **dimensione**
  + Per dati la cui **vita può durare più a lungo** della funzione in cui il dato nasce
  + Alloco oggetto su Heap, Box possiede quel blocco:
    - Quando il box sparisce (b outOfScope), il dato viene distrutto e viene rilasciata la memoria
    - Il box ha un distruttore per rilasciare il blocco di memoria del quale è padrone

*V viene allocato nell’heap; nella varibaile b ho il puntatore al blocco*

* *Per accedere al valore \*b*
* *Nel caso in cui ci sia un metodo è indifferente fare \*b.metodo o b.metodo*

******

**

* **T\*** → puntatori const int \* 🡪 puntatori read only NATIVI
  + Soggetto ad accessi illeciti
  + Può contenere cose strambe
  + Unico modo per poterlo usare è racchiuderlo in un blocco unsafe
    - Devo essere sicuro di quel che sto facendo
* **Mut int\*** 
  + Unico modo per poterlo usare è racchiuderlo in un blocco unsafe
    - Devo essere sicuro di quel che sto facendo

**Nota per i mut**

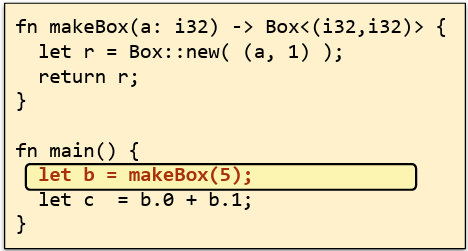
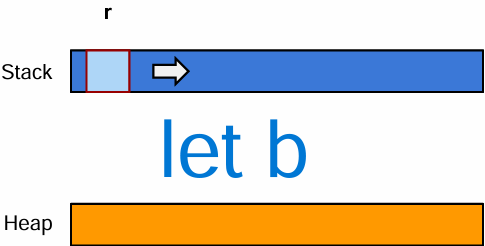
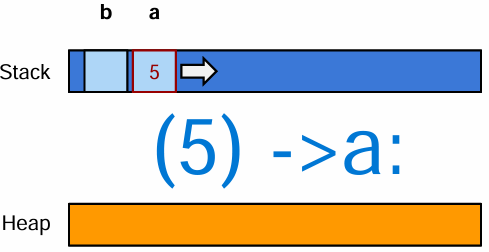
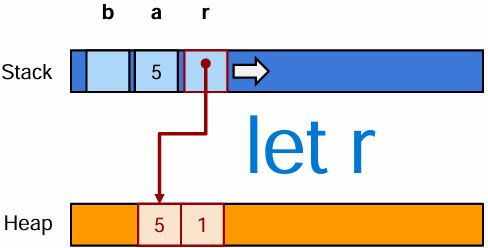
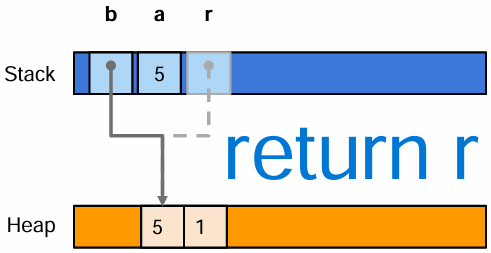
Immagine che contiene testo, Carattere, schermata, numero

Descrizione generata automaticamentesto ancora usando r e provo a cambiare il dato   
🡪 il compilatore mi blocca

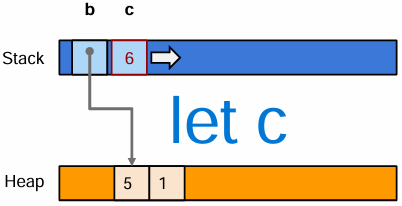
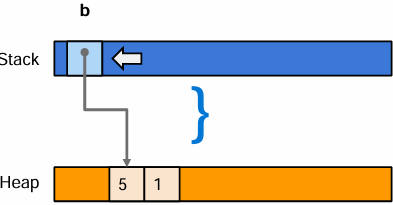
Immagine che contiene testo, Carattere, schermata, calligrafia

Descrizione generata automaticamente

r è ancora attivo ma sto accedendo ad i

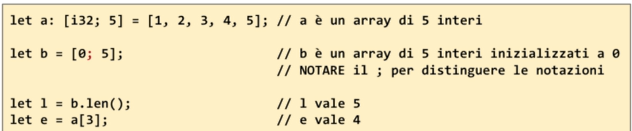
BOX t, esempio di variabile con vita più lunga

perdo il diritto di accesso e il dovere di rilascio perché facendo il return sto cedendo il dato ad un altro



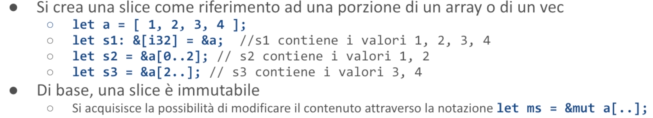
#### **4.Array**

Sequenza continua di oggetti omogenei, disposti consecutivamente nello stack

* **Lunghezza nota a priori e immutabile**
* Array mutabile *( i valori che contiene sono modificabili)*
* .len() mi dice quanto è lungo
* Per accedere al valore nome[ind]
  + Se provo ad accedere ad un valore olre la dimensione 🡪 non me lo permette e mi avvisa

Gli array hanno bisogno di essere agili → considerare alcune parte degli array.  
Rust offre la possibilità di far riferimento ad una sequenza di valori consecutivi la cui lunghezza diventa nota durante l’esecuzione. 🡪**Tipo Slice**

#### **5. Slice**

Segmento all’interno di un array → riferimento ad un blocco di T

* Posso prendere una slice dell’intero array
* Posso prendere una slice di una parte dell’array

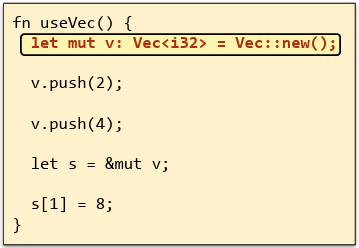
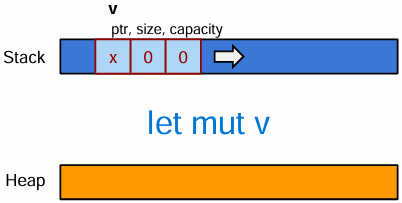
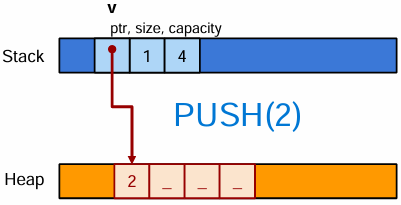
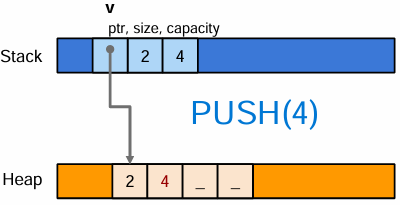
***Se uso una mutable slice, l’intero array a cui fa riferimento la slice è inacessibile.***

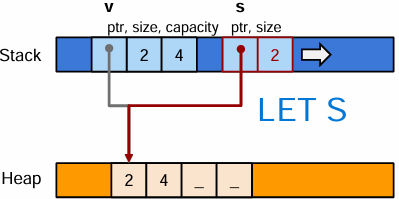
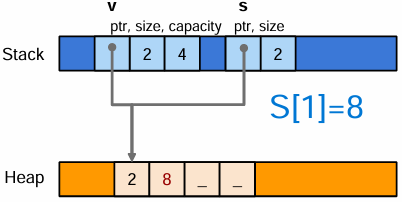
La slice è un **fat pointer** in quanto contiene il riferimento al primo valore e il numero di eleemtni che fanno parte della mia fetta.

#### **6. Vec<T>**

Blocco di elementi omogenei allocato su Heap, ridimensionabile

* Gestisce in automatico la memoria che utilizza e provvede al rilascio
* Ci sono **3 campi**:
  + **Puntatore** ad un blocco sull’heap su cui può mettere cose
  + Intero unsigned che indica **la grandezza complessiva** del blocco 🡪 capacity
    - Nota: si parte da 4
  + Intero unsigned che indica quanti **blocchi** ha effettivamente **usato 🡪 size**
* Se riempie tutte le caselle che aveva, va dal sistema operativo e chiede se può avere un blocco di grandezza superiore
  + Quando il sistema operativo gli da il nuovo blocco, lui fa una copia delle cose che aveva nel vecchio in quello nuovo e restituisce il blocco vecchio al S.O.





* Let s → v non può più accedere
* Stack si contrae → s buttato vua
* V è tornato in possesso delle sue facoltà
* Si continua a contrarre stack
* Rilascio del blocco di memoria v
* V va via

Si è fatto assegnare 4 blocchi

#### **7.Stringhe**

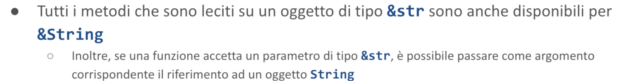
#### Nella rappresentazione multi byte, devo capire da quale sto iniziando a leggere. Perciò si ha il BOM:

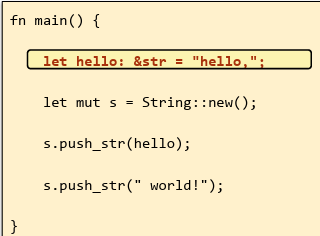
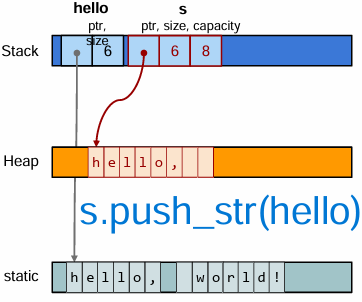
* Marcatore con cui iniziano i file, vale FFFE
  + Se leggo FFFE, il file è little endian
  + Se leggo EFFF, il file è big endian

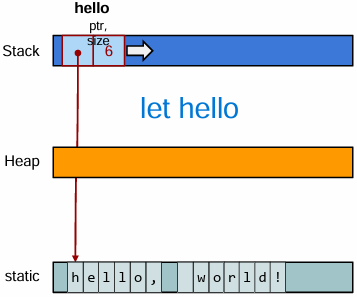
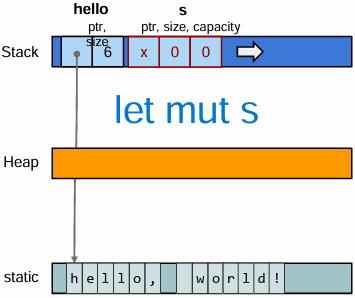
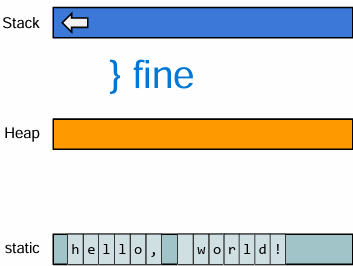
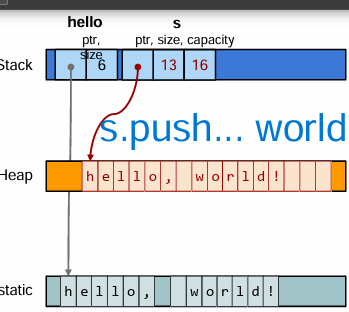
Rust gestisce le stringhe con UTF8 senza BOM.

LA stringa è una sequenza di byte, non tutte le sequenza di byte sono legali.

Esistono due tipi di stringhe

* Immutabili → slice di byte
  + Tipo primitivo **str**
  + **Accedute con &str**
    - Indirizzo primo carattere +nr di byte possibili
  + Non hanno il terminatore /0 in quanto la lunghezza è espressa nello slice
* Mutabili → **String**
  + Contiene 3 pezzettini
    - Puntatore su Heap dove c’è il buffer su cui opera
    - Nr che mi dice quanto di quegli elementi sono occupati
    - Capacità → quanto è grande il buffer che sto usando
  + Se ad un oggetto String metto una & davanti, diventa un str e quindi beneficia di ciò che una str può fare





Finito, stack si contrae:

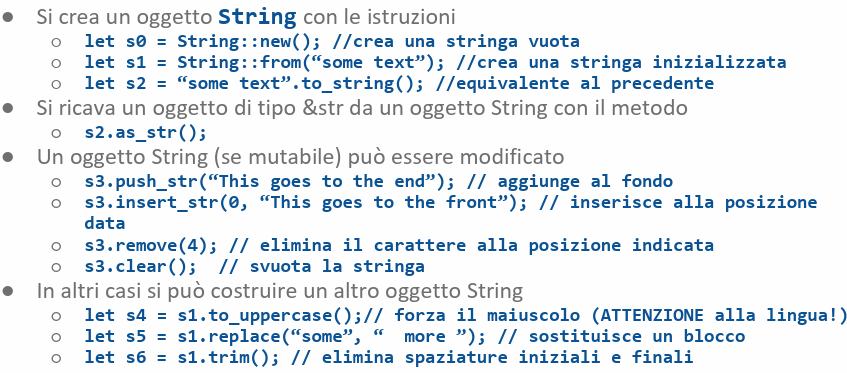
* S possiede un buffer
  + Fa metodo drop che lo rilascia
* Hello è uno slice senza possesso quindi esce senza far nulla
* Lo spazio statico resta così

Non ho 7 byte liberi.. me ne servono in totale 13, chiedo 16 al S.O. .

Il S.O. me li da, faccio la copia lì e restituisco la zona di memoria vecchia da 8.

Ho una capacity pari a 0, devo inserire 6. Quindi chiedo 8 byte al S.O.

OPERAZIONI CON STRINGHE

nota: esisto vari tipi di stringhe oltre ad &str e String :

* **OsStr, OsString** → stringhe come piacciono al sistema operativo. Se sto creando un file, al sistema operativo devo passarlo così.
* **Path, Pathbuf** → stringa del sistema operativo segmentabile con gli slash
* **Cstr, Cstring** → C vuole lo 0 quindi se devo collaborare con C devo usare queste
* **&’ static str** → tempo di vita coincidente con l’intero processo

## ISTRUZIONI E ESPRESSIONI

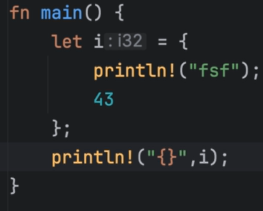
Il corpo di una funzione è costituito da istruzione e/o espressioni separare da **;**

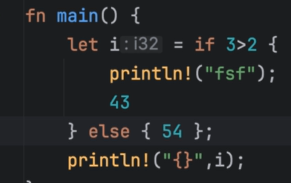
* Una istruzione ha come tipo di ritorno **()**, un’espressione può restituire un tipo arbitrario

I costrutti **let …** e **let mut …** sono istruzioni

* Creano un legame tra la variabile indicata e il valore assegnato

Tutto ciò che è scritto tra graffe è **un’espressione**.  
Valore di ritorno di un’espressione è l’ultimo elemento del blocco a condizione che non termini con ;

es: if else ; loop;

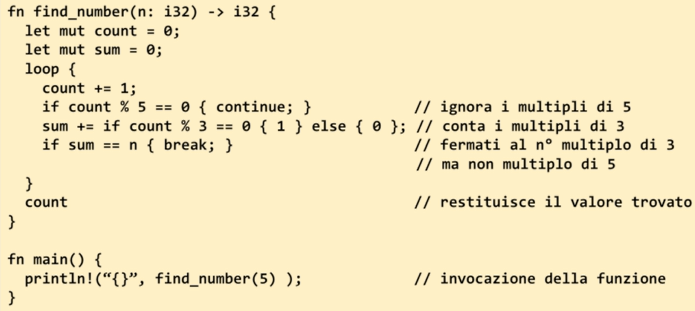
dato che non c’è il ; dopo 43, questo valore viene assegnato ad i

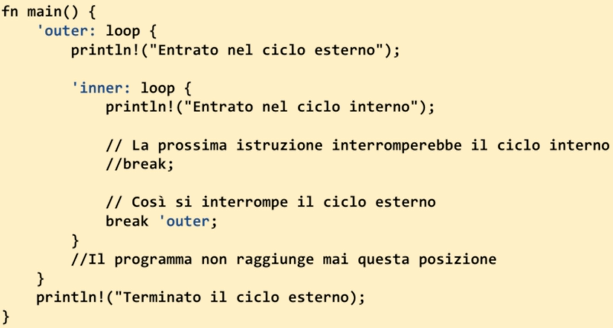
#### 

## FUNZIONI

**fn nome\_funzione(argomenti) → valore\_di\_ritorno{}**

* Se ritorna un valore diverso da (), allora è obbligatorio inserire **→ TipoRitornato**
* Corpo della funzione racchiuso tra parentesi **{ }**
  + Valore di ritorno senza ; oppure return valore;





È possibile annidare **più loop** rappresentati da **etichette**  
Per una migliore gestione si hanno delle istruzioni

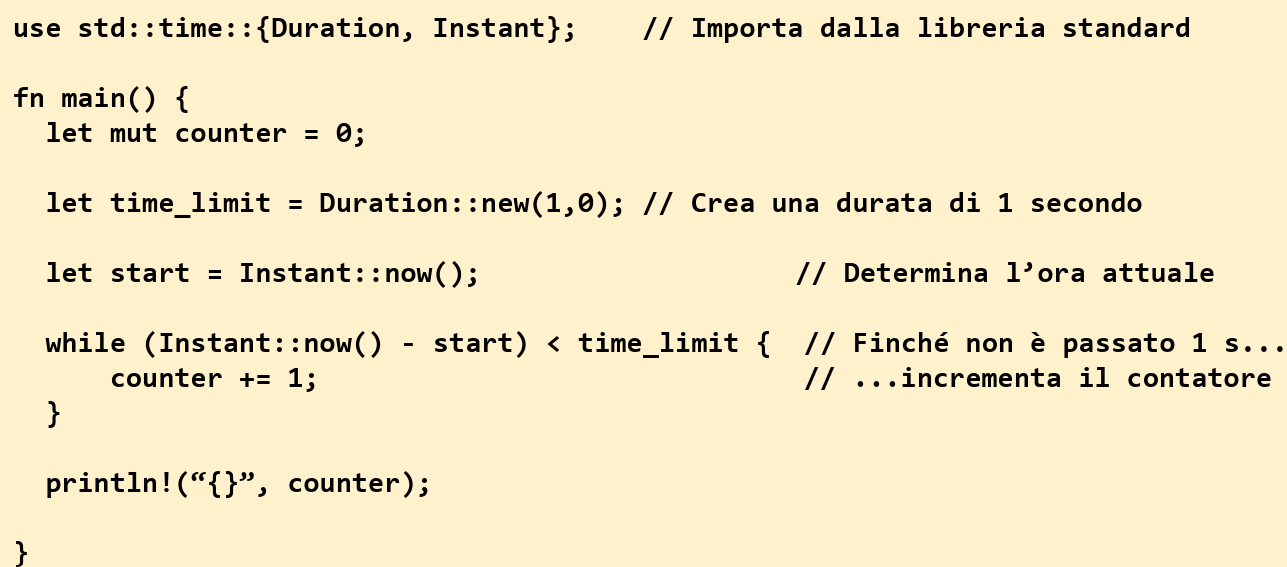
* **break**
* **continue**

**for**: si può fare solo con espressioni iterabili → array, range

se si ha bisogno della data corrente, c’è la libreria

### TEMPO

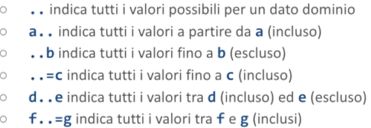
Esistono due particolari:

* **istant** che cerca sul S.O, l’orologio e si fa diredata e ora corrente
* **Duration**: intervallo di tempo

***Nota****: differenza tra due istant → duration*

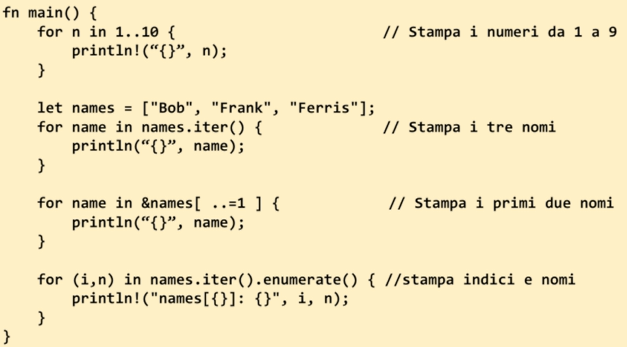
*Nota: somma istant+duration=istant*

#### RANGE:

* ***a..b*** *→ da inizio compreso a fine esclusa*
* ***c..=d*** *→ da inizio a fine, entrambi inclusi*

*es:*

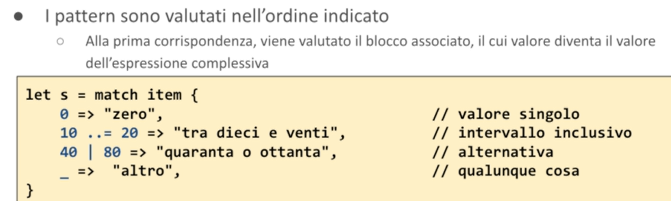
* *for i:u8 in .. 🡪 tutti i numeri da 0 a 255*
  + *tutti i valori del dominio u8*
* *for in:u8 in 25.. → da 25 a 255*

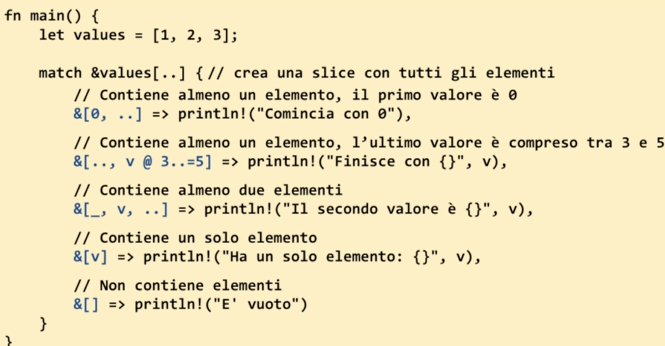
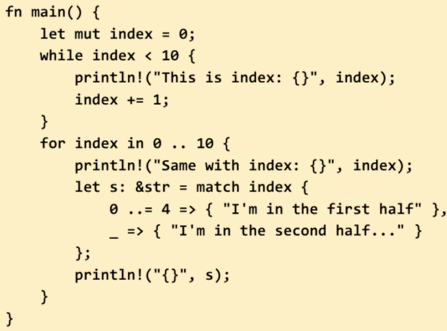


### **MATCH**:

utile quando si deve scegliere una di molte strade

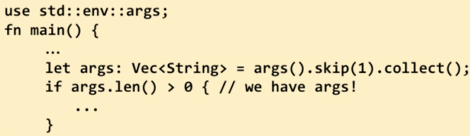
* le strade che elenco devono essere comprensive di tutte le possibile combinazioni
* a differenza dello switch, qui si usa pattern matching





### **RIGA DI COMANDO:**

std∷env∷args; → tipo predefinito

**args()** → fornisce un iteratore ai singoli elementi

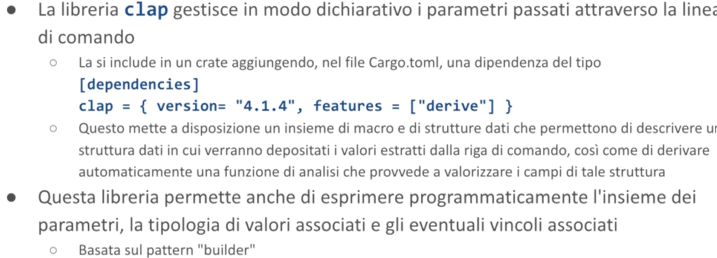
**.skip(1)** → mi permette di saltare il primo elemento in quanto il primo elemento fornito da args è il nome

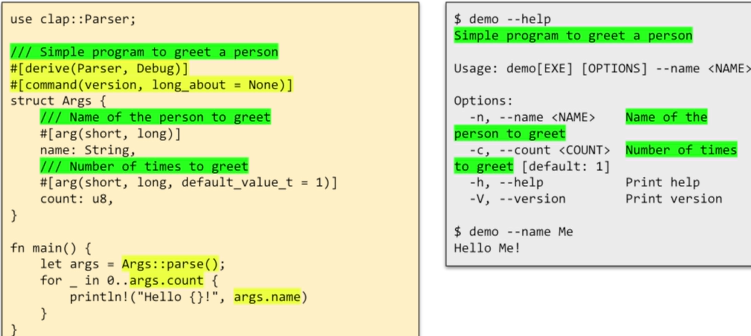
.**collect()** → metti insieme in un vettore

**Args.len()** → mi dice a priori quanti sono

Esiste una libreria per gestire tutto **→ CLAP**

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
* questo mette a disposizione un insieme di macro e strutture dati che permettono di
  + descrivere una struttura dati in cui verranno depositati i valori estratti da linea di comando
  + derivare automaticamente una funzione di analisi che provvede a valorizzare i campi di tale struttura
* permette inoltre di esprimere programmaticamente l’insieme di parametri, la tipologia di valori associati e gli eventuali vincoli associati → pattern builder



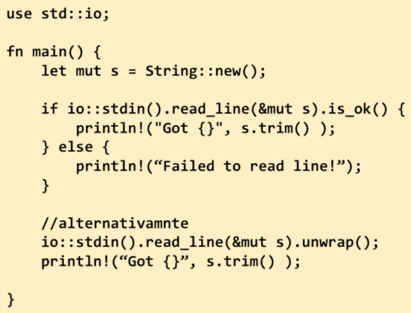
**Name** deve arrivare con un **prefisso** short -, long –

**Count** deve arrivare con un prefisso

**Versione**

* compilando, per prima cosa mi mette una stringa → il commento con 3 slash

#### **i/O da console**

**std∷io** → contiene la definizione delle strutture standard per i/o

Se operazione I/O a buon fine → **Result=ok**, *all’interno di ok c’è il vero risultato*;

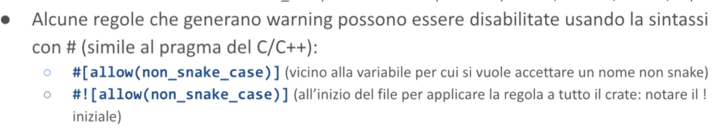
altrimenti **error** e dentro error c’è il valore.   
**Result è una monade.**

**NOTA:** per garantire la correttezza del programma, occorre gestire esplicitamente l’eventuale errore, utilizzando:

* **is\_ok()**  che verifica il contenuto del valore
* **unwrap()** che causa l’interruzione forzata del programma in caso di errore; Se non c’è stato errore, ritorna il valore incapsulato.

#### **Convenzioni sui nomi**

Tipi: UppeCamelCase

Valori (cvariabili, funzioni, case): lower\_snake\_case

**Fat pointer** 🡪 puntatore che oltre all’inizio ha anche la lunghezza

In Rust, il compilatore verifica il possesso e il tempo di vita delle variabile e garantisce accessi safe.  
Nel caso in cui il programmatore vuole creare delle situazioni particolari aggirando queste verifiche, deve racchiudere il codice in un blocco unsafe{ }.

**Nota**: Nella maggior parte dei casi le assegnazioni sono in realtà dei movimenti:

* Ti passo il valore(byte), i diritti e i doveri

Nel caso di oggetti che implementano **il tratto copy** (come i numeri):

* Assegnazioni sono semplici assegnazioni
  + Si può trasferire il valore senza trasferire anche diritti e valori
  + Mutualmente esclusivo con il tratto drop

Nel caso del **tratto clone**: duplicare in profondità .clone

* Copio superficie e oltre
  + Se ad esempio ho un box(S)
    - Alloco nello heap un’altra zona della dimensione di S (a cui il box originale puntava)
    - Muovo tutti i dati da sorgente a destinazione
    - Creo puntatore a tale zona
    - 🡪 ottengo due puntatori distinti

**BOX**:

* Rilascio dato
* Rilascio box

Tipi aggiunti da programmatore

* + Struct
  + Union

Ogni tipo gode di determinate **proprietà definite mediante** un meccanismo dichiarativo basato sui **tratti**.  
*Non esiste il concetto di gerarchia di ereditarietà.*

#### TRATTO:

Descrive un insieme di comportamenti che un dato tipo implementa

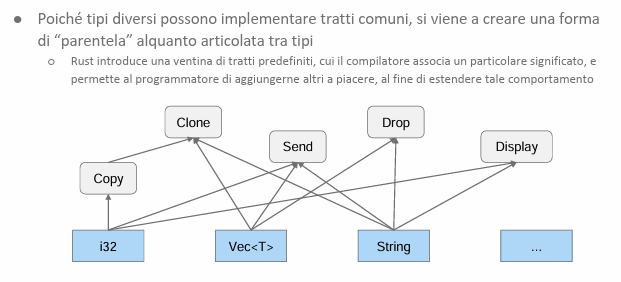
* Compilatore sa che un determinato tipo implementa dei determinati tratti

Ad ogni tipo è possibile associare 0+ **tratti** 🡪 dichiarazione di impegno a fornire una serie di comportamenti

Immagine che contiene linea, diagramma, Parallelo, Carattere

Descrizione generata automaticamenteTratti sono legati tra di loro

* Chi gode di copy, deve avere il tratto clone
* Chi gode di Drop non può avere il tratto copy



* **Display**: serve a dare un’informazione utile all’utente finale di cosa c’è scritto dentro
* **Debug**: solitamente in automatico, deriva una rappresentazione che permetta al programmatore di capirci qualcosa (es: tuple implementano debug)
  + Se voglio vedere puntatore → :p