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

```
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
```

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

Una tupla ha tipo $(T_1, T_2, \ldots, T_n)$, dove $T_1, T_2, \ldots, T_n$ sono i tipi dei singoli valori membro

modo più semplice per realizzare un dato custom:

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

```
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)
```

Si accede ai vari cambi della tupla in modo posizionale:

- Nome.0
- Nome.1
- Nome.2

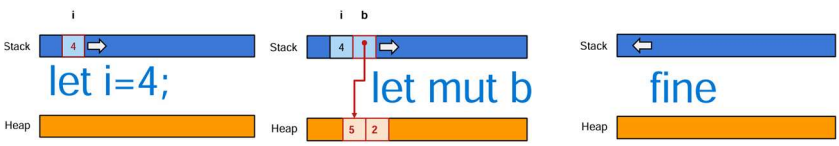
3.Puntatori:

vari modi per rappresentare indirizzi in memoria:

- **REF** → non copia il valore, ma gli da un puntatore `let r1 = &v;`
  - 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 `let r2 = &mut v;`
  - 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** `let b = Box::new(v);`
  - 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*

```
fn useBox() {
  let i = 4;
  let mut b = Box::new( (5, 2) );

  (*b).1 = 7;

  println!("{:?}", *b); // (5,7)
  println!("{:?}", b);  // (5,7)
}
```



- **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

```
fn main() {
  let mut i = 32;

  let r = &i;
  println!("{}", *r);

  i = i+1;    // Problematico!
  println!("{}", *r);
}
```

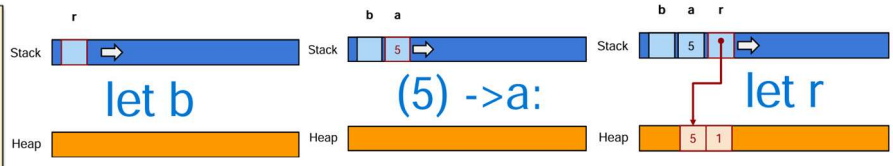
sto ancora usando r e provo a cambiare il dato
→ il compilatore mi blocca

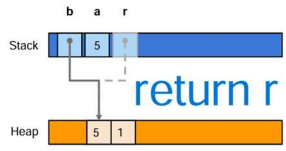
```
fn main() {
  let mut i = 32;

  let r = &mut i;
  println!("{}", i);// Problema!

  *r = *r+1;
  println!("{}", *r);
}
```

r è ancora attivo ma sto accedendo ad i

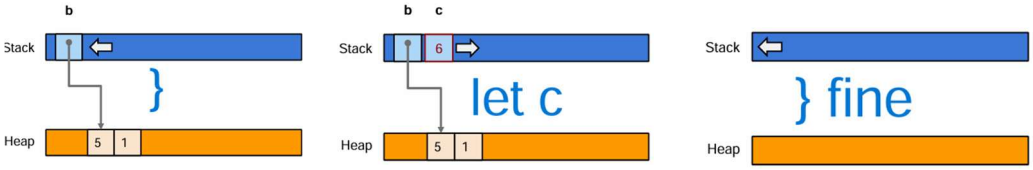
BOX t, esempio di variabile con vita più lunga

```
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;
}
```





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

```
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();                   // l vale 5
let e = a[3];                      // e vale 4
```

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

- 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[..];`

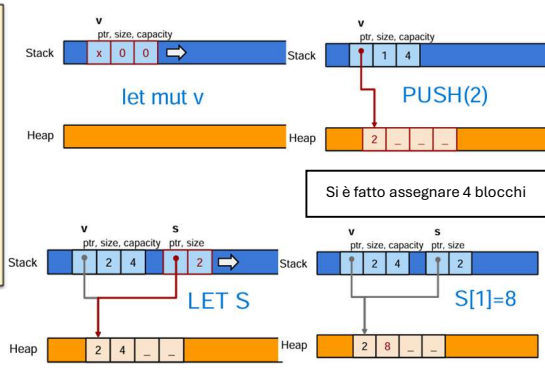
Se uso una mutable slice, l'intero array a cui fa riferimento la slice è inacessibile.
La slice è un **fat pointer** in quanto contiene il riferimento al primo valore e il numero di eleemtni che fanno parte della mia fetta.

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.

```
fn useVec() {
  let mut v: Vec<i32> = Vec::new();

  v.push(2);

  v.push(4);

  let s = &mut v;

  s[1] = 8;
}
```



Si è fatto assegnare 4 blocchi

- 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

7.Stringhe

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

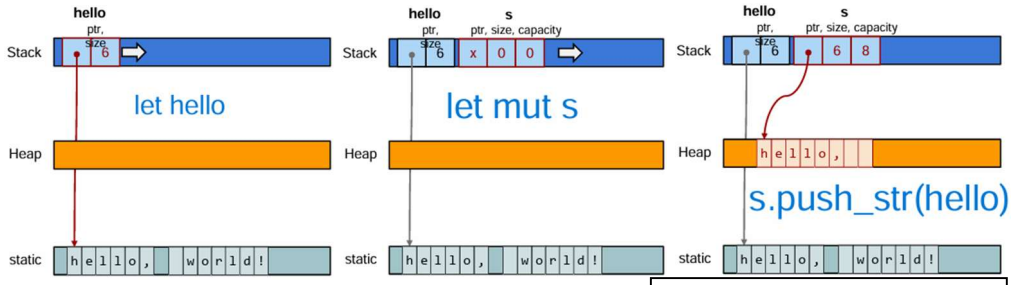
- Marcatore con cui iniziano i file, vale FFFE
  - Se leggo FFFE, il file è little endian
  - Se leggo EFFF, il file è big endian

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

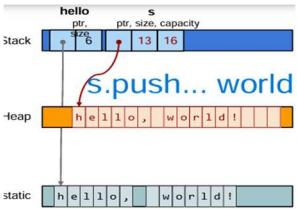
- Immutabili → slice di byte
  - Tipo primitivo **str**
  - **Accedute con &str**
    - Indirizzo primo carattere +nr di byte possibili
  - Non hanno il terminatore /0 in quanto la lunghezza è espressa nello slice
- Mutabili → **String**
  - Contiene 3 pezzettini
    - Puntatore su Heap dove c'è il buffer 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

- 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`

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

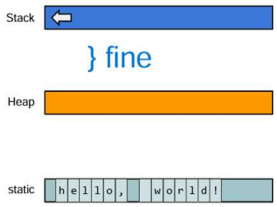


Ho una capacity pari a 0, devo inserire 6. Ouindi chiedo 8 bvte al S.O.



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.



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

OPERAZIONI CON 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();`
- 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`

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;

```
fn main() {
    let i:i32 = {
        println!("fsf");
        43
    };
    println!("{}",i);
}
```

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

```
fn main() {
    let i:i32 = if 3>2 {
        println!("fsf");
        43
    } else { 54 };
    println!("{}",i);
```

FUNZIONI

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

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

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;

```
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
}
```

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

- **break**
- **continue**

```
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);
}
```

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

```
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);
```

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*

- .. 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() {
    for n in 1..10 {                        // Stampa i numeri da 1 a 9
        println!("{}", n);
    }

    let names = ["Bob", "Frank", "Ferris"];
    for name in names.iter() {              // Stampa i tre nomi
        println("{}", name);
    }

    for name in &names[ ..=1 ] {            // Stampa i primi due nomi
        println("{}", name);
    }

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

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

- 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() {
    let mut index = 0;
    while index < 10 {
        println!("This is index: {}", index);
        index += 1;
    }
    for index in 0 .. 10 {
        println!("Same with index: {}", index);
        let s: &str = match index {
            0 ..= 4 => { "I'm in the first half" },
            _ => { "I'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:

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

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

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

```
[dependencies]
clap = { version= "4.1.4", features = ["derive"] }
```

- 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

```
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)
    }
}
```

```
$ 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!
```

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.

```
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

Tipi: UppeCamelCase

Valori (cvariabili, funzioni, case): lower_snake_case

- 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)

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.

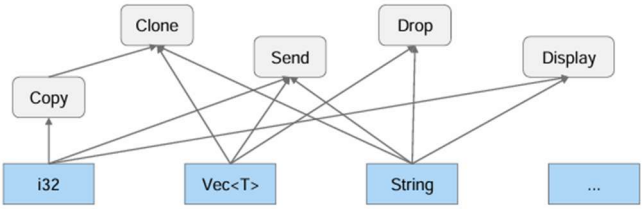
TRATTO:
Descrive un insieme di comportamenti che un dato tipo implementa

- 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

Tratti sono legati tra di loro

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



- 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

- **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) `println!("{:?}", *b); // (5,7)`
  - Se voglio vedere puntatore → :p