Struct in rust

Si utilizza il costrutto **struct** che permette di rappresentare un blocco di memoria in cui sono disposti una serie di campi il cui nome e tipo sono indicati dal programmatore. - ogni struct introduce un nuovo tipo

```
struct Player {
    name: String,  // nickname, 24 byte
    health: i32,   // stato di salute (in punti vita),4 byte
    level: u8,     // livello corrente,1 byte

    //dim totale è pari almeno alla somma dei campi (il compilatore potrebbe
aggiungere qualcosa in fondo: qua 3 byte per arrivare a 32)
}
```

- Si utilizza la notazione **CamelCase** (struct inizia in maiuscola, seconda parola inizia in maiuscolo). I campi invece seguono la **snake_case** in minuscolo separati da underscore.
- Per istanziare una struct:

```
 let mut s = Player{name : "Mario".to_string(), health:25, level : 1}
 //per instanziare da un'altra struct già esistente :
 //i campi sono tutti presei dalla precedente tranne il name
let s1 = Player {name; "Paolo".to_string(),..s}
//si accede al singolo campo con la notazione puntata:
s.name
```

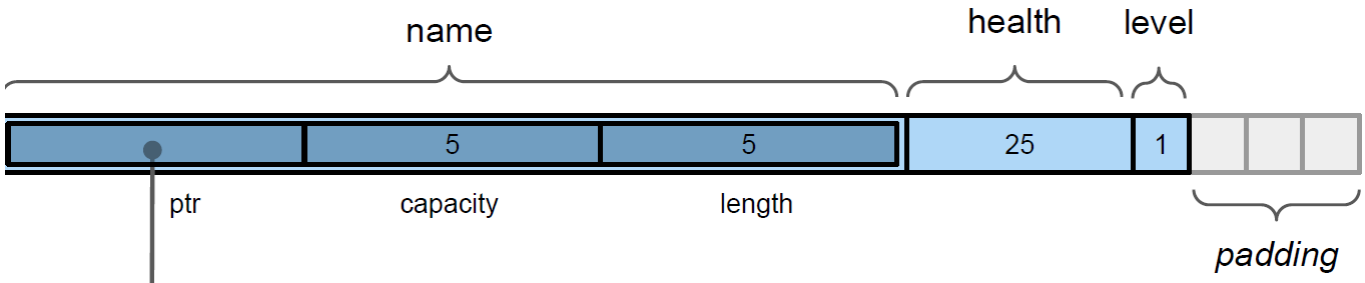
Secondo modo per definire le struct

Potremmo non avere necessità di dare nome esplicito ai campi. Le struct di questo tipo si istanziano come una tupla con l'aggiunta del nome della struct. Si può definire una struct vuota, che non alloca memoria analogamente al tipo ()

```
 struct Playground ( String, u32, u32 );
 struct whatever(); // ha dominio nullo, non ha spazio per contenere nulla: è un
tipo senza dimensione. E' un marcatore.
 Se torno una struct di questo tipo return whatever => torno qualcosa simile al
return void ma che ha un nome e non occupa memoria.
 Il compilatoore mi aiutera a usare in modo coerente quel valore
struct Empty; // non viene allocata memoria per questo tipo di valore. Mi serve  a
marcare delle cose
let mut f = Playground( “football”.to_string(), 90, 45 );
let e = Empty;
```

Rappresentazione in memoria

In rust il compilatore può reorganizzare i campi come meglio crede




```
fn main() {
  let p1 = Player {
      name: "Mario".to_string(),
      health: 25,
      level: 1u8,
  };
  println!("{:?}", p1);
}
```

Davanti alla definizione posso mettere

```
#[ repr(...) ]
//unico caso utile è  #[ repr(C) ] : in tal caso non riordina ma ottengo una
rappresentazione coerente con le interfacce binarie definite dal linguaggio c
```

Esistono una serie di funzioni che ci aiutano a lavorare con la memoria

- **std::mem::align_of(...)** ci da informazioni sull'allineamento richiesto da un tipo
- **std::mem::size_of(...)** indica la dimensione del tipo
- **std::mem::align_of_val(...)** permette di conoscere l'allineamento richiesto da un particolare valore
- la funzione **std::mem::size_of_val(...)** ne indica la dimensione

Moduli

Fa dell'idea di modulo anche quella di incapsulamento.

```
mod Esempio {
    struct  S2 {
       alfa : i32,
       beta : bool
   }
}
```

Tutto ciò che sta in un modulo non è visibile a meno di dichiararlo pubblico

```
mod Esempio {
    pub struct  S2 {
        alfa : i32,
        beta : bool
    }
}
```

Per poterlo utilizzare devo scrivere :

```
 use Esempio::S2; //prima del main
```

Ora s2 è visibile ma non i singoli campi (è visibile all'interno del modulo).

Cerco un modo per lavoraci: aggiungo metodi. In rust i metodi sono separati dalla struttura. Sono affianco alla struct

```
//questo è un metodo della struttura
//il metodo è visibile ora
    impl S2{
        #[derive(Debug)] //questo serve a renderlo stampabile
        //con l'istruzione **println!("{:?}",s2);**
          pub  fn new(alfa:i32, beta: bool) -> Self{ //Self rappresenta il tipo
sui cui sto lavorando
               Self{
                   alfa: alfa,
                   beta: beta
               }
           }
    }
//nel main
let s1 = S1{alfa: 5, beta : false};
Nel main cosi posso creare una nuova struct senza accedere ai singoli campi:
let s2=S2::new(5,false); //funzione costruttrice
println!("{:?}",s1);
println!("{:?}",s2);
```

Struct completamente visibili nel proprio modulo, invisibili fuori. La struct diventa visibile se la rendo pubblica, ma non vuol dire che i singoli campi lo diventano

Visibilità

Sia la struct nel suo complesso che i singoli campo che la formano possono essere preceduti da un **modificatore di visibilità**. Di base i campi sono privati : possono essere resi pubblici con la parola chiave pub. A differenza di quanto avviene in altri linguaggi, in cui struttura e comportamento sono definiti contestualmente in un unico blocco (classe), in Rust la definizione dei metodi associati ad una struct avviene separatamente, in un blocco di tipo **impl**

Metodi

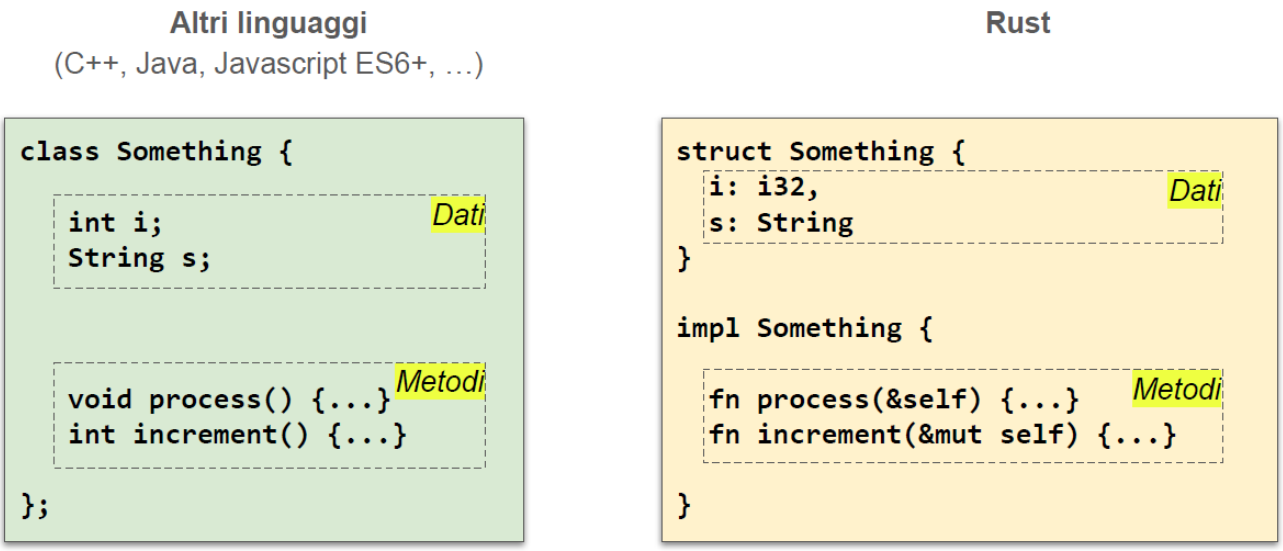
Rust non ha il concetto di classe nativo.

- Le struct non sono organizzate in una gerarchia di ereditarietà
- il concetto di metodo si applica a tutti i tipi compresi quelli primitivi
- si definiscono i metodi collegati ad un tipo in un blocco racchiuso tra parentesi graffe, preceduto dalla parola chiave ***impl*** seguita dal nome del tipo.
    - le funzioni presenti in tale blocco il cui parametro sia ***self***, **&self o mut self diventano metodi** (self è circa il this)
    - **le funzioni che non hanno come primo parametro self sono dette funzioni associate** e svolgono il ruolo giocato dai costruttori e dai metodi statici nei linguaggi ad oggetti

```
//Metodo getter : ne esistono 3 di base
impl S2{
pub fn getAlfa(self) -> i32  { //passo per movimento
             self.alfa
         }
}
impl S2{
  pub fn getAlfa(&self) -> i32  { //passo per sola lettura
             self.alfa
         }
}
impl S2{
 pub fn getAlfa(&mut self) -> i32  {
             self.alfa
         }
}
let  s2=S2::new(5,false);
println!("{:?}",s2.get_alfa()); /*questa va con la seconda*/
//con la prima no perchè perdo il possesso

 let mut  s2=S2::new(5,false);
println!("{:?}",s2.get_alfa()); /*questa va con la terza*/
```

Senza self non sono d'istanza ma costruttrici (tipo metodi statici di java) Se hanno self come parametro corrispondono ai metodi di istanza



I metodi sono funzioni legate ad un'istanza di un dato tipo Il legame si manifesta sia a livello sintattico, che a livello semantico Sintatticamente, un metodo viene invocato a partire da un'istanza del tipo a cui è legato : si usa la notazione **instance.method(…), dove instance è una variabile del tipo dato (detto anche ricevitore del metodo), e method è il nome della funzione**. Semanticamente, il codice del metodo ha accesso al contenuto (pubblico e privato) del ricevitore *attraverso la parola chiave self*. Di fatto, i metodi legati ad una struct vengono implementati sotto forma di funzioni con un parametro ulteriore (chiamato self, &self o &mut self) il cui tipo è vincolato alla struct per la quale sono definiti

```
impl str {
  pub const fn len(&self) -> usize //…
}
/*-------------------------------------*/
let str1: &str = "abc";
println!("{}", str1.len());        // 3
println!("{}", str::len(str1));    // 3 , qua metto esplicitamente il ricevitore
```

Self

Il primo parametro di un metodo definisce il livello di accesso che il codice del metodo ha sul ricevitore .

- **self** indica che il ricevitore viene passato per movimento, di fatto consumando il contenuto della variabile: è una forma contratta della notazione **self: Self**
- **&self** indica che il ricevitore viene passato per riferimento condiviso: è una forma contratta di **self: &Self**
- **&mut self** indica che il ricevitore viene passato per riferimento esclusivo: è una forma contratta di **self: &mut Self**

Se presente, il parametro self compare come primo elemento nella dichiarazione del metodo. All'atto dell'invocazione del metodo, esso è ricavato implicitamente dal valore che compare a sinistra del punto che precede il nome del metodo

Esempio

```
struct Point {
  x: i32,
  y: i32,
}

impl Point {

  fn mirror(self) -> Self {
    Self{ x: self.y, y: self.x }
  }

  fn length(&self) -> i32 {
    sqrt(self.x*self.x + self.y*self.y)
  }

  fn scale(&mut self, s: i32) {
    self.x *= s;
    self.y *= s;
  }
}
```

Consuma una struct Point e **produce** una nuova struct dello stesso tipo

Opera su una struct Point senza possederla né mutarla

Opera su una struct Point cambiandone il contenuto

```
fn main() {


  let p1 = Point{ x: 3, y: 4 };
  let mut p2 = p1.mirror();

  let l1 = p2.length(); // l1: 5

  p2.scale(2);

  let l2 = p2.length();
  // l2: 10
}
```

`p1` non potrà più essere usato dopo questa linea: il suo valore è stato mosso nel parametro `self` del metodo `mirror()`

Al parametro `self` del metodo `length()` è stato legato un riferimento condiviso a `p2`: tale riferimento cessa di esistere quando il metodo ritorna

Al parametro `self` del metodo `scale(...)` è stato legato un riferimento mutabile a `p2`: tale riferimento cessa di esistere quando il metodo ritorna

Costruttori

In rust **non esiste il concetto di costruttore**. Qualunque frammento di codice, in un qualunque modulo che abbia visibilità di una data struct e dei suoi campi, può crearne un'istanza, indicando un valore per ciascun campo. Per convenzione, un metodo di questo tipo viene chiamato

```
pub fn new() -> Self {...}
```

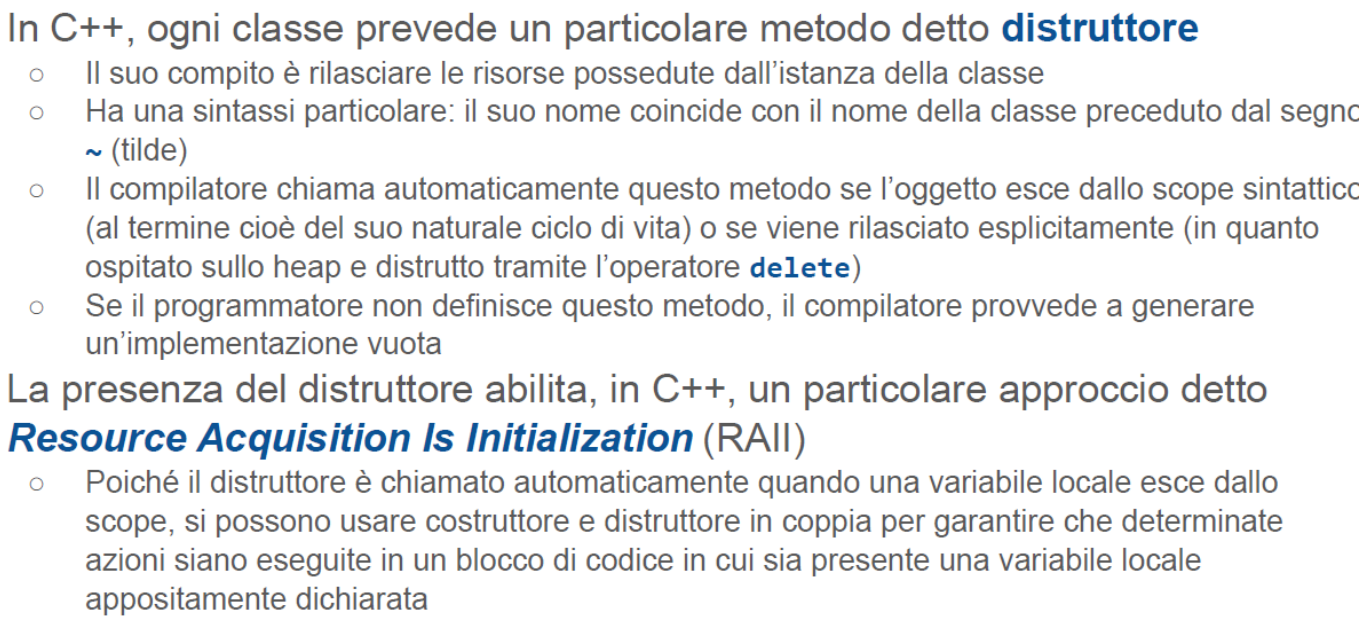
Poiché Rust non supporta l'overloading delle funzioni, se servono più funzioni di inizializzazione, ciascuna di esse avrà un nome differente: in questo caso la convenzione è utilizzare un pattern come

```
pub fn with_details(...) -> Self {...}
```

Distruttori

C++

E' uno solo. Si chiama ~nomeclasse. Il distruttore viene chiamato automaticamente quando la classe esce dal suo scope. Serve a fare particolari azioni prima della distruzione.



Paradigma Raii

Il paradigma RAII in sintesi:

- Le risorse sono incapsulate in una classe (struttura) in cui:
  - il **costruttore** acquisisce le risorse e stabilisce eventuali invarianti, oppure lancia un'eccezione se non può essere fatto
  - il **distruttore** rilascia le risorse e ***NON*** lancia mai eccezioni
- Si usano le risorse attraverso l'istanza di una classe RAII-compatibile che:
  - ha una gestione automatica delle durata di tutte le risorse, **oppure**
  - ha un ciclo di vita connesso al ciclo di vita di un altro oggetto (ad es., è parte di esso)
- In questo contesto, la presenza della semantica del ***Movimento***, garantisce il corretto trasferimento delle risorse, mantenendo la sicurezza del rilascio

Distruttori in rust

- Rust gestisce il rilascio di risorse contenute in un'istanza attraverso il tratto **`Drop`**
  - Tale tratto è costituito dalla sola funzione **`drop(&mut self) -> ()`**
  - Il compilatore riconosce la presenza di questo tratto nei tipi definiti dall'utente e provvederà a chiamare la funzione che lo costituisce quando le variabili di quel tipo escono dal proprio scope sintattico
  - Si può forzare il rilascio delle risorse contenute in un oggetto usando la funzione **`drop(some_object);`** che ne acquisisce il contenuto e determina l'uscita dallo scope

```rust
pub struct Shape {

  pub position: (f64, f64),
  pub size: (f64, f64),
  pub type: String

}
```

```rust
impl Drop for Shape {

    fn drop(&mut self) {
        println!("Dropping shape!");
    }
}
```

- Il paradigma RAII viene mutuato dal C++ e costituisce un importantissimo modo per gestire automaticamente acquisizione e rilascio di risorse
  - Oltreché permettere l'esecuzione automatica di coppie di funzioni
  - Può essere usato ogni qual volta sia necessario garantire il corretto rilascio di risorse di sistema, come la memoria allocata sullo heap, handle di file, socket, ...
- Il tratto **`Drop`** è **mutuamente esclusivo** con il tratto **`Copy`**
  - Se un tipo implementa il primo non può implementare l'altro, e viceversa

Metodi statici

In Rust, è possibile implementare metodi analoghi semplicemente non indicando, come primo parametro, né self né un suo derivato Questo permette la creazione di funzioni per la costruzione di un istanza, metodi per la conversione di istanze di altri tipi nel tipo corrente o, semplicemente, l'accesso a funzionalità statiche (come nel caso di librerie matematiche o l'accesso in lettura di parametri di configurazione)

- L'esempio tipico è il metodo new. la chiamata in questo caso sarà usando

```
<Tipo>::metodo(...)
```

ENUM

- In Rust, è possibile introdurre tipi enumerativi composti da un semplice valore scalare come in C e C++ o anche incapsulare, in ciascuna alternativa, una tupla o una struct volta a fornire ulteriori informazioni relative allo specifico valore.
- Inoltre , è possibile legare metodi ad un'enumerazione aggiungendo un blocco impl.. come nel caso delle struct

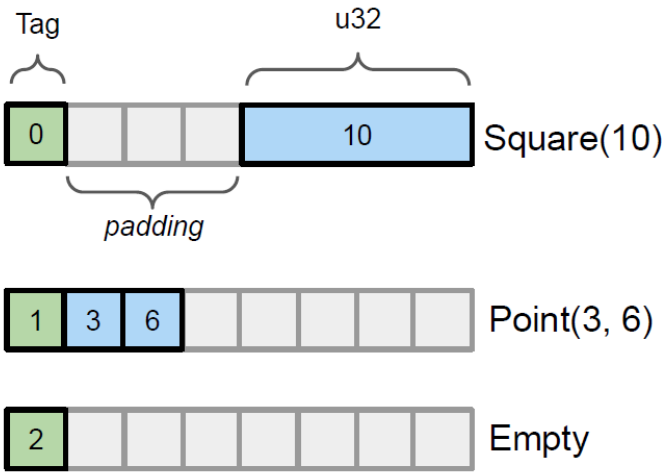
```
enum HttpResponse {
  Ok = 200,
  NotFound = 404,
  InternalError = 500
}
```

```
enum HttpResponse {
  Ok,
  NotFound(String),
  InternalError {
    desc: String,
    data: Vec<u8> },
}
```

- enum grande quanto il più grande dei suoi campi (qua + almeno 1 byte di selettore)
- enum lavora insieme a match
- enum è un tipo somma
- alle varie alternative dell'enum posso associare dei dati e questo è alla base di molti pattern

Rappresentazione in memoria

- In memoria gli enum occupano lo spazio di un intero da 1 byte più lo spazio necessario a contenere la variante più grande




```
enum Shape {
     Square(u32),
     Point(u8, u8)
     Empty,
}
```

- Si genera un byte di selezione **tag** che sta all'inizio (1 byte se ho 255 al max selezioni, 2 byte per di più ecc)
- ho del padding perchè gli interi devono cominciare a multipli di 4
- qua la struttura è grande 4 + 1 tag + totale per arrivare ad 8

Match

- Il costrutto **`match...`** si presta particolarmente per gestire in modo differenziato valori enumerativi
  - Il comportamento offerto dal pattern matching e la possibilità di legare variabili temporanee in base alla struttura del valore che viene analizzato offre una sintassi compatta ed efficiente per esprimere comportamenti alternativi (**destructuring assignment**)
  - Il fatto che i pattern confrontati debbano essere esaustivi, garantisce che il codice resti coerente anche se il numero di possibili alternative presenti nell'enumerazione cambia nel tempo

```
enum Shape {
    Square { s: f64 },
    Circle { r: f64 },
    Rectangle { w: f64, h: f64 }
}
```

```
fn compute_area(shape: Shape) -> f64 {
    match shape {
        Square { s } => s*s,
        Circle { r } => r*r*3.1415926,
        Rectangle {w, h} => w*h,
    }
}
```

- Viene introdotta s: s{}: se è uno square introduci la variabile s , se è un circle introduci variabile r che corrisponde al corrispettivo nella struct
- Si usa anche con gli if:
  - L'utilizzo del pattern matching e la possibilità di destrutturare un valore complesso non sono limitate al costrutto **`match ...`**
    - E' possibile usare la stessa tecnica anche all'interno di costrutti **`if let <pattern> = <value> ...`** e **`while let <pattern> = <value>`**
    - Tali costrutti verificano se il valore fornito corrisponda o meno al pattern indicato e, nel caso, eseguono le necessarie assegnazioni alle variabili contenute nel pattern

```
enum Shape {
    Square { s: f64 },
    Circle { r: f64 },
    Rectangle { w: f64, h: f64 }
}
```

```
fn process(shape: Shape) {
    // stampa solo se shape è Square...
    if let Square { s } = shape {
        println!("Square side {}", s);
    }
}
```

- Si usa anche con le assegnazioni semplici
    - La destrutturazione è anche utile per ottenere un “parsing” di una struttura, così da gestirne più facilmente i campi, estraendoli in contenitori singoli da trattare separatamente.
        - I campi della struttura sono “estratti” con il loro nome.

```
pub struct Point {
     x: f32,
     y: f32
}
```

```
...

let p = Point { x: 5., y: 10. };
...

// la destrutturazione deve rispettare i nomi dei campi
let Point { x, y } = p;

println!("The original point was: ({},{})", x, y);
```

- utilizza la semantica delle assegnazioni
    - Il processo di destrutturazione utilizza la semantica delle assegnazioni
        - Se il valore implementa il tratto **copy**, le variabili introdotte nel pattern conterranno una copia dell'elemento corrispondente
        - In caso contrario, verrà eseguito un movimento, invalidando il valore originale
    - Se il valore originale non è posseduto (ad esempio è un riferimento) e non è copiabile, occorre far precedere al nome della variabile da assegnare la parola chiave **ref** (eventualmente seguita da **mut**)
        - Indicando così che ciò che viene assegnato è un riferimento (mutabile) alla parte di valore corrispondente

```
enum Shape {
    Square { s: f64 },
    Circle { r: f64 },
    Rectangle { w: f64, h: f64 }
}
```

```
fn shrink_if_circle(shape: &mut Shape) {
   if let Circle { ref mut r } = shape {
       *r *= 0.5;
    }
}
```

Enumerazioni generiche

- Come verrà meglio presentato in seguito, è possibile definire tipi generici
  - Ovvero costrutti che contengono dati il cui tipo è specificato attraverso una meta-variabile, indicata accanto al nome del tipo, racchiusa tra i simboli '<' e '>'
  - I frammenti di codice che utilizzano un tipo generico hanno il compito di indicare quale sia il tipo concreto da sostituire alla meta-variabile
  - Vec<T>, ad esempio, rappresenta un generico vettore di valori omogenei di tipo T
- Rust offre due importanti enumerazioni generiche, che sono alla base della sua libreria standard
  - `Option<T>` - rappresenta un valore di tipo T **opzionale** (ovvero che potrebbe non esserci)
  - `Result<T,E>` - rappresenta **alternativamente** un valore di tipo T o un errore di tipo E
- `Option<T>` contiene due possibili valori
  - `Some(T)` - indica la presenza e contiene il valore
  - `None` - indica che il valore è assente
- `Result<T,E>` si usa per indicare l'esito di un computazione; può valere:
  - `Ok(T)` - Se la computazione ha avuto successo, il valore restituito ha tipo T
  - `Err(E)` - Se la computazione è fallita, il tipo E viene usato per descrivere la ragione del fallimento
- L'istruzione `match ...` risulta particolarmente utile con questo tipo di valori

```
fn plus_one(x: Option<i32>) ->
                    Option<i32> {
   match x {
       None => None,
       Some(i) => Some(i + 1),
   }
}
```

```
fn open_file(n: &str) -> File {

    match File::open(n) {
        Ok(file) => file,
        Err(_) => panic!("error"),
    }
}
```