02_1 Tipi composti

STRUCT

Costrutto che permette di memorizzare **informazioni eterogenee**.
Viene allocata tanta memoria quanta ne serve per la somma di tutti i cambi + un eventuale per compattamento (padding ulteriore aggiunto)

```
struct Player {
    name: String,
    health: i32,
    level: u8,
}
```

Nota:

- Nome della struct con lettera maiuscola **CamelCase**
- Nomi dei campi minuscoli snake_case
- Quando inizializzo una struct devo dare un valore a tutti i suoi campi (altrimenti non compila).

Istanziare una struct:

Blocco preceduto dal nome della struttura, contenente un valore per ciascun campo.

Nota: se il nome de valore coincide con quello del campo, si può omettere.

```
let mut s = Player { name: "Mario".to_string(), health: 25, level: 1 };
let p = Player { name, health, level }; // {name: name, health: health, level: level}
```

Nota:si può **istanziare** una struct a partire da un'altra dello stesso tipo

```
let s1 = Player {name: "Paolo".to_string(), .. s}
```

Nota: per accedere ai singoli campi si usa dot.

```
println!("Player {} has health {}", s.name, s.health );
s.level += 1; //l'accesso in scrittura richiede che s sia mutabile
```

Nota: Quando si crea un struct, si arricchisce il sistema dei tipi di rust

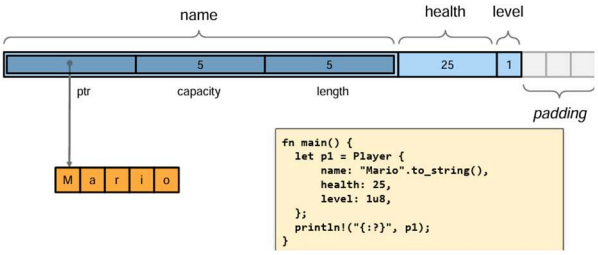
Nella definizione della struct, si può definito solo il tipo dei campi (omettendo il nome) → *in questo caso i cambi hanno stessa visibilità della struct che possiamo definire come tupla.*

```
struct Playground ( String, u32, u32 );
struct Empty;   // non viene allocata memoria per questo tipo di valore

let mut f = Playground( "football".to_string(), 90, 45 );
let e = Empty;
```

Si può definire una **struct vuota** → esempio di 0size type → tipo che occupa 0 byte

Se non si specifica come salvare in memoria la struct, compilatore **Rust li riordina** in mood tale da ottenere un buon **allineamento** → singoli campi devono rispettare vincoli di allineamento *(vett/Stringa 8; int32 bit mult 4; int 16 bit, mult 2)*



#[repr(..)] → forza il compilatore a salvare in un determinato modo → se scrivo #[repr(C)], ottengo rappresentazione coerente con le librerie C.

PUB

Cambia visibilità di struct se davanti a struct e cambia visibilità all'interno se anche davanti ai campi al suo interno.
Di default struct e campi interni sono privati → appartengono al modulo e possono essere visti dai sottomoduli.

Incapsulamento :

- Occorrre poter **associare comportamenti/metodi** alla nostra struct
- La definizione dei metodi associati ad una struct avviene separatamente in un **blocco impl**
    - SERIE DI ISTRUZIONI: hanno tra gli argomenti i valori di tipo self
    - Le funzioni che non hanno come parametro self, vengono dette funzioni associate → costruttori e altri metodi statici.

```
struct Something {
  i: i32,                          Dati
  s: String
}

impl Something {

  fn process(&self) {...}          Metodi
  fn increment(&mut self) {...}

}
```

Metodi

- Definiti in un blocco impl
- Nel blocco sono presenti funzioni
    - Con parametro self/&self/mut &self → metodi
    - Senza → funzioni associate
- Legati ad un'istanza di un dato tipo
    - Metodo invocare a partire da tale istanza (ricevitore)
- Il metodo può accedere al contenuto del ricevitore attraverso self

```
struct Something {
  i: i32,                          Dati
  s: String
}

impl Something {

  fn process(&self) {...}          Metodi
  fn increment(&mut self) {...}

}
```

```
Pritnln!("{}", str::len(str1));  //→3
```

```
impl str {
  pub const fn len(&self) -> usize //…
}
```

```
let str1: &str = "abc";
println!("{}", str1.len());
3
```

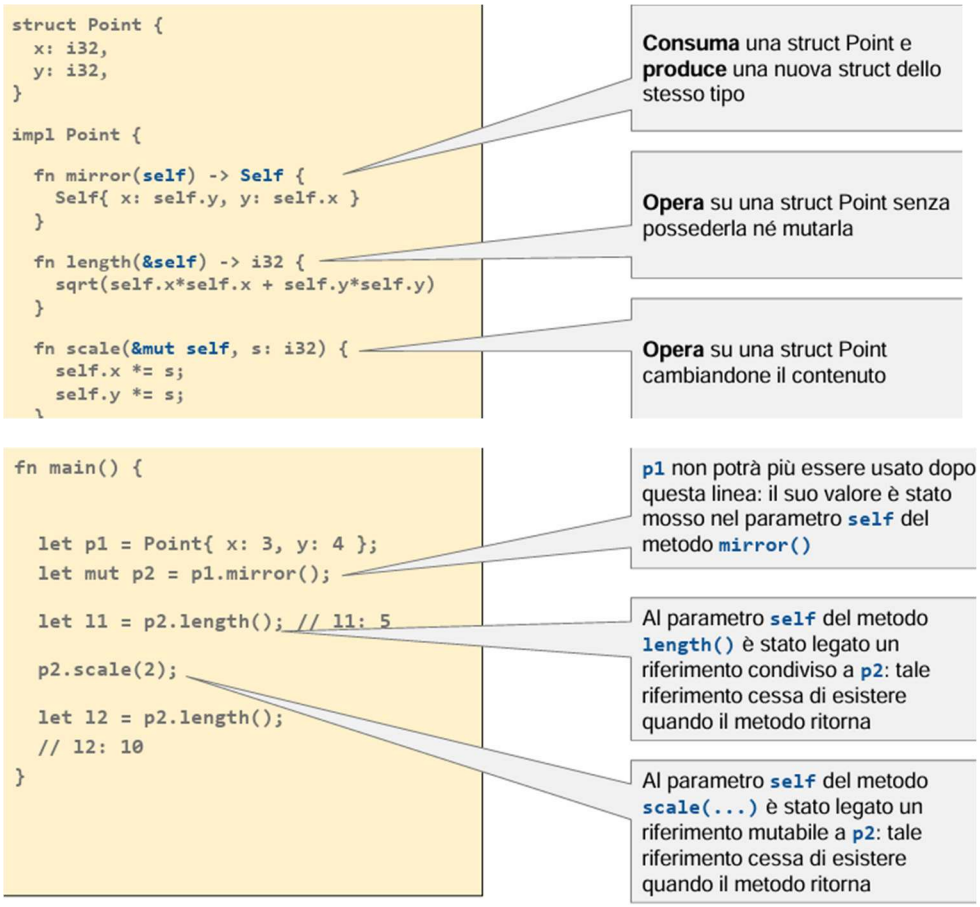
Ricevitore: struttura che viene in gresso (ciò che ho prima del punto)

Nota su self:

- **self**: ricevitore passato per movimento
- **&self :** ricevitore passato per riferimento condiivwo
- **&mut self**: ricevitore passato per riferimento esclusivo → prestito
- *NOTA: self deve essere il primo parametro nel cado in cui si metta*.

Varie note:

- se il metodo è scritto minuscolo, il ricevitore prende il metodo
- Ci sono metodi statici che non possono avere self.
- Se si vuole fare riferimento ad un campo dato della struttura devo specificare self.



In rust non esistono i **costruttori** → qualunque frammento di codice, in un qualunque modulo che abbaia visibilità di una data struct e dei suoi campi, può creare un'istanza.

Per evitare duplicazioni di codice e favorire incapsulamento, le implementazioni inclusono metodi statici per inizializzare le istanze

- 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 {...}`

Distruttore

Rust gestisce il rilascio delle risorse attraverso il **tratto DROP** → se presente, garantisce che se il valore giunge alla fine della sua vita, libera tutto.

- drop(&mut self) -> ()
- nel caso in cui si voglia forzare il rilascio delle risorse di some_object → `drop(some_object);`

Dunque, il Tratto drop → consiste in un metodo drop

```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!");
    }
}
```

Nota : Drop è l'ultima cosa del ciclo di vita di una struct

Nota: per ogni costruzione ci deve essere max una drop → infatti copy e drop sono mutualmente esclusivi

RAII → paradigma molto usato da rust, mutuato dal C++ per la gestione di rilascio e acquisizione risorse

METODI STATICI

Non hanno nulla come parametro e non ritornano nulla.

- Creare funzioni per costruire un'istanza
- Accesso a funzionalità statiche (librerie)
- Metodi per conversione di istanze
- Metodo **new**

Enum

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

Versione semplice → **sequenza di etichette** a cui posso associare dei valori
→ se non dessi valori darebbe 0,1,2 …

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

Versione plus → **legare metodi ad un'enumerazione** (aggiungendo un blocco impl)

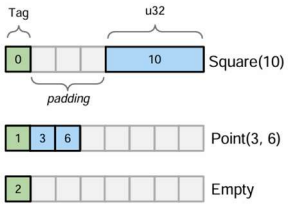
- ok è una struct vuota
- not found è una struct di tipo tupla che contiene un campo di tipo stringa
- internalError è una struct con dati ordinati

Enum fornisce la possibilità di tenere dati di tipo diverso, sapendo cosa mi passerà in ogni caso

➔ Primo byte → in quale alternativa sono
➔ Secondo byte → informazioni relative

?

- 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,
}
```

Enumerazione ***e match***

match: effettua un **controllo tra le varie alternative**, verificando che abbiamo coperto tutte le alternative possibili

```
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,
    }
}
```

se non ho bisogno di usare tutte le strade possibili, posso usare if ***let*** *→ testo solo quella che mi interessa.*

Destrutturazione: `if let <pattern> = <value> …` e `while let <pattern> = <value>`

```
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);
    }
}
```

La destrutturazione è anche usata per ottenere un **parsing** di una struttura per leggere più facilmente i campi separatamente:

```
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);
```

N**OTA**: se il valore su cui si effettua la destrutturazione implementa il tratto copy, si effettua una copia; in caso contrario, si esegue il movimento, invalidando il valore originale.

NOTA: se il valore originale non è posseduto (es: riferimento) e non è copiabile, bisogna usare ref che indica l'assegnazione del riferimento alla parte di valore richiesta.

```
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

- **Option<T>** → valore di tipo T opzionale
  - Some(T) → indica la presenza e contiene il valore
  - None → indica che il valore è assente
- **Result<T,E>** → valore di tipo T o errore di tipo E
  - Ok(T) → computazione ha avuto successo
  - Err(E) → computazione fallita

```
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"),
    }
}
```