

# Tipi composti

Struct, enum

# Tipi composti in C e C++

- In C e C++ il costrutto **struct ...** permette di creare un nuovo tipo che contiene un gruppo di campi la cui accessibilità è aperta a tutti (pubblica)
  - In C++, ad una struct è lecito associare metodi a livello istanza o a livello di tipo (**static**)
- Sempre in C++, è possibile introdurre il costrutto **class ...**
  - Esso è simile a **struct ...**, ma permette di limitare l'accesso ad un sottoinsieme del proprio contenuto (campi e metodi) al solo tipo corrente (**private**) o alle classi che da essa derivano (**protected**)
  - Come nel caso di struct, è anche possibile consentire l'accesso a qualsiasi contesto (**public**)

# Tipi composti in C e C++

```
class Foo {  
public:  
    // Methods and members here are publicly visible  
    double calculateResult();  
protected:  
    // Elements here are only visible  
    // to this class and to its subclasses  
    double doOperation(double lhs, double rhs);  
private:  
    // Elements here are only visible to ourselves  
    bool debug_;  
};
```

C++

```
struct Foo {  
    bool debug_;  
};  
  
double calculateResult(struct Foo s);  
double doOperation(struct Foo s, double lhs, double rhs);
```

C

```
class Foo {  
    // by default everything here is private  
    double calculateResult();  
    double doOperation(double lhs, double rhs);  
    bool debug_;  
};
```

C++

```
struct Foo {  
    // by default everything here is public  
    double calculateResult();  
    double doOperation(double lhs, double rhs);  
    bool debug_;  
};
```

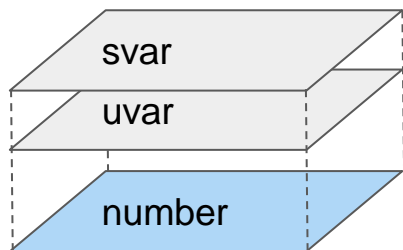
C++

# Tipi composti in C e C++

- In entrambi i linguaggi è possibile definire altre forme di tipi composti
  - `enum ...`
  - `union ...`
- Il tipo `enum` definisce un insieme di costanti che vengono associate, dal compilatore o dal programmatore, a valori interi distinti
  - Una variabile di tipo `enum` è implementata come numero intero il cui valore è vincolato ad essere uno di quelli definiti dal tipo
  - C++11 consente di usare altri tipi scalari (`char`, `short`, ...) per rappresentare il valore
- Il tipo `union` permette di usare lo stesso blocco di memoria per rappresentare dati di tipo diverso, in alternativa l'uno all'altro
  - Una variabile di tipo `union` occupa una dimensione pari al più grande dei dati contenuti al suo interno
  - È responsabilità del programmatore sapere cosa è contenuto in una `union` ed accedervi di conseguenza

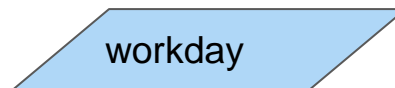
# Tipi composti in C e C++

```
union sign
/* A definition and a declaration */ C/C++
{
    int svar;
    unsigned uvar;
} number;
```



La porzione di memoria della variabile **number** ospita, alternativamente, un intero con segno o uno senza segno

```
enum DAY          /* Defines an enumeration type */
{
    sunday = 0,    /* Names day and declares a */
    monday,        /* variable named workday with */
    tuesday,       /* that type */
    wednesday,     /* wednesday is associated with 3 */
    thursday,
    friday,
    saturday       /* saturday is associated with 6 */
} workday;
```



**workday** è un intero il cui dominio è limitato all'insieme  $\{ x \mid 0 \leq x \leq 6 \}$

# Struct in Rust

- Spesso occorre mantenere unite informazioni tra loro eterogenee
  - Sebbene sia possibile utilizzare una tupla, questa tende a nascondere la semantica complessiva del dato: una tupla contenente due numeri interi potrebbe essere usata per rappresentare una frazione oppure la coordinata di un pixel sullo schermo
  - Quando si intende associare una semantica particolare o legare ad una struttura dati un insieme di comportamenti, è possibile introdurre una **struct**
- Una **struct** è un costrutto che permette di rappresentare un blocco di memoria in cui sono disposti, consecutivamente, una serie di campi il cui nome e tipo sono indicati dal programmatore

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

# Struct

- Per convenzione, il nome della struct comincia con la lettera maiuscola e utilizza la convenzione CamelCase
  - Per i campi, come per le variabili, le funzioni ed i metodi, si usa la convenzione snake\_case
- Si istanzia una struct tramite un blocco preceduto dal nome della struttura, contenente un valore per ciascun campo: quando il nome del valore coincide con quello del campo è possibile abbreviare la notazione
  - `let mut s = Player { name: "Mario".to_string(), health: 25, level: 1 };`
  - `let p = Player { name, health, level }; // {name: name, health: health, level: level}`
- Si può istanziare una nuova struct a partire da un'altra dello stesso tipo, i campi omessi ricavano i loro valori dall struct ricevuta
  - `let s1 = Player {name: "Paolo".to_string(), .. s}`
- Si accede ai singoli campi con la notazione puntata (*var.field*)
  - `println!("Player {} has health {}", s.name, s.health );`
  - `s.level += 1; //l'accesso in scrittura richiede che s sia mutabile`
- Ogni struct introduce un nuovo tipo, il cui nome coincide con il nome della struct, basato sui tipi che la compongono
  - Questo permette di disambiguare l'uso che si intende fare delle singole informazioni contenute

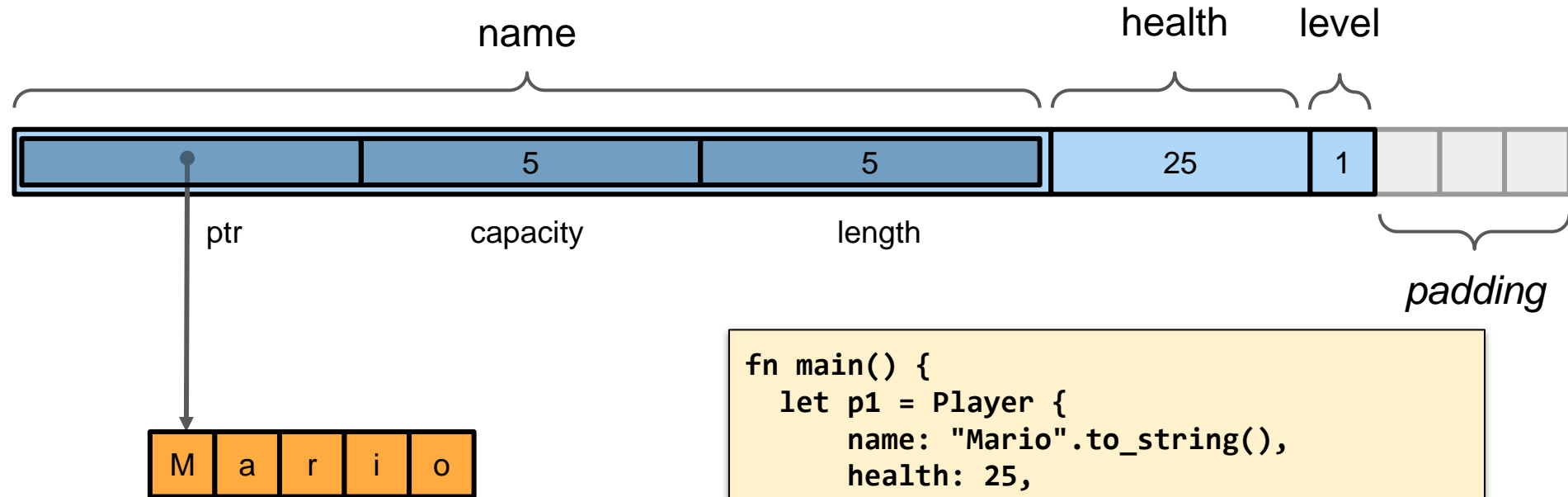
# Struct

- Si possono definire delle struct simili a delle tuple, indicando solo il tipo del campo, senza attribuire un nome
- 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 Empty;    // non viene allocata memoria per questo tipo di valore  
  
let mut f = Playground( "football".to_string(), 90, 45 );  
let e = Empty;
```



# Probabile rappresentazione in memoria



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

# Rappresentazione in memoria

- La disposizione in memoria dei singoli campi è conseguenza di vincoli ed ottimizzazioni e può essere controllata attraverso opportuni meccanismi
  - Ogni singolo campo, in base al proprio tipo, richiede che l'indirizzo a cui viene collocato sia multiplo di una data potenza di 2 (allineamento)
  - La funzione `std::mem::align_of_val(...)` permette di conoscere l'allineamento richiesto da un particolare valore, mentre la funzione `std::mem::size_of_val(...)` ne indica la dimensione
  - I vincoli di allineamento dipendono dalla piattaforma di esecuzione (modello di CPU)
- L'allineamento e la disposizione di una struct è controllata attraverso l'attributo `#[repr(...)]` anteposto alla dichiarazione della struct
  - In assenza di tale attributo, viene assunta la rappresentazione di default, che lascia libero il compilatore di riordinare la sequenza dei campi, per ottimizzare l'accesso
  - Indicando `#[repr(C)]`, si ottiene una rappresentazione coerente con le regole di interfaccia binaria definite dal linguaggio C, fondamentali per l'interoperabilità con librerie scritte in altri linguaggi

# Visibilità

- Sia la struct nel suo complesso che i singoli campi che la formano possono essere preceduti da un **modificatore di visibilità**
  - Tale modificatore ha impatto sull'accesso al contenuto della struct da parte di codice presente in moduli diversi da quello in cui la struct è stata definita
  - Di base, **i campi sono** considerati **privati** (accessibili solo al codice del modulo corrente e ai suoi sotto-moduli): possono però essere resi pubblici facendo precedere il nome dalla parola chiave **pub**
- Questo permette di implementare il meccanismo di **incapsulamento** (*information hiding*)
  - Per essere efficace, occorre però poter associare un insieme di comportamenti (metodi) alla struct
  - 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 è un linguaggio ad oggetti nel senso tradizionale del termine e, conseguentemente, **non ha** il concetto di classe
  - Sebbene le struct possano apparire simili alle classi di altri linguaggi, il parallelismo è limitato
  - In particolare, le struct **NON** sono organizzate in una gerarchia di **ereditarietà**
  - Il concetto di metodo si applica invece 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 primo parametro sia **self** (una parola chiave che rappresenta l'istanza del tipo di cui si sta facendo l'implementazione), **&self** o **&mut self** diventano **metodi** (*self* corrisponde grosso modo a quello che in altri linguaggi ad oggetti viene chiamato *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

# Metodi

## Altri linguaggi

(C++, Java, Javascript ES6+, ...)

```
class Something {  
    int i; Dati  
    String s;  
  
    void process() {...} Metodi  
    int increment() {...}  
};
```

## Rust

```
struct Something {  
    i: i32, Dati  
    s: String  
}  
  
impl Something {  
    fn process(&self) {...} Metodi  
    fn increment(&mut self) {...}  
}
```

# Metodi

- 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!("{}", str1.len());           //  
3
```

# Metodi

- 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

# Metodi

```
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



# Metodi

```
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 C++, tutte le classi contengono metodi particolari detti **costruttori**
  - Hanno il compito di inizializzare le istanze della classe
  - Se non vengono scritti esplicitamente dal programmatore, il compilatore provvede a generarne alcuni (costruttore di default, privo di parametri, costruttore di copia, con un solo parametro di tipo riferimento costante ad un'istanza della classe corrente)
- 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
  - Questo garantisce che il programmatore sia consapevole delle informazioni contenute al suo interno
- Per evitare significative duplicazioni di codice e favorire l'incapsulamento, le implementazioni spesso includono metodi statici per l'inizializzazione delle 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 {...}**

# Distruttori

- In C++, ogni classe prevede un particolare metodo detto **distruttore**
  - Il suo compito è rilasciare le risorse possedute dall'istanza della classe
  - Ha una sintassi particolare: il suo nome coincide con il nome della classe preceduto dal segno ~ (tilde)
  - Il compilatore chiama automaticamente questo metodo se l'oggetto esce dallo scope sintattico (al termine cioè del suo naturale ciclo di vita) o se viene rilasciato esplicitamente (in quanto ospitato sullo heap e distrutto tramite l'operatore **delete**)
  - Se il programmatore non definisce questo metodo, il compilatore provvede a generare un'implementazione vuota
- La presenza del distruttore abilita, in C++, un particolare approccio detto **Resource Acquisition Is Initialization** (RAII)
  - Poiché il distruttore è chiamato automaticamente quando una variabile locale esce dallo scope, si possono usare costruttore e distruttore in coppia per garantire che determinate azioni siano eseguite in un blocco di codice in cui sia presente una variabile locale appositamente dichiarata

# Resource Acquisition Is Initialization (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

```
Something acquire_resource() { ... }  
void release_resource(Something s) { ... }  
  
class RaiiClass {  
    Something s;  
public:  
  
    RaiiClass() {                // COSTRUTTORE  
        this->s = acquire_resource();  
    }  
  
    ~RaiiClass() {                // DISTRUTTORE  
        release_resource(this->s);  
    }  
};
```

C++

```
void some_function() {  
    RaiiClass c1; // la costruzione di c1  
                // provoca l'invocazione di  
                // acquire_resource()  
  
    ...  
    if (some_condition) return;  
    else {  
        //fai altro poi ritorna  
    }  
    // qualunque sia il modo in cui si esce,  
    // c1 viene distrutta e invocata la  
    funzione  
    // release_resource(...)  
}
```

C++

# Distruttori

- 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

```
pub struct Shape {  
  
    pub position: (f64, f64),  
    pub size: (f64, f64),  
    pub type: String  
  
}
```

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

# Distruttori

- Il paradigma RAI 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 **C++** è lecito inserire all'interno del costrutto **class ... { }** la dichiarazione di metodi preceduti dalla parola chiave **static**
  - Essi non sono legati ad una specifica istanza, ma possono operare sulle istanze della classe (se ne conoscono l'indirizzo) avendo accesso anche alle componenti private
- 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++
  - Ma 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  
}
```

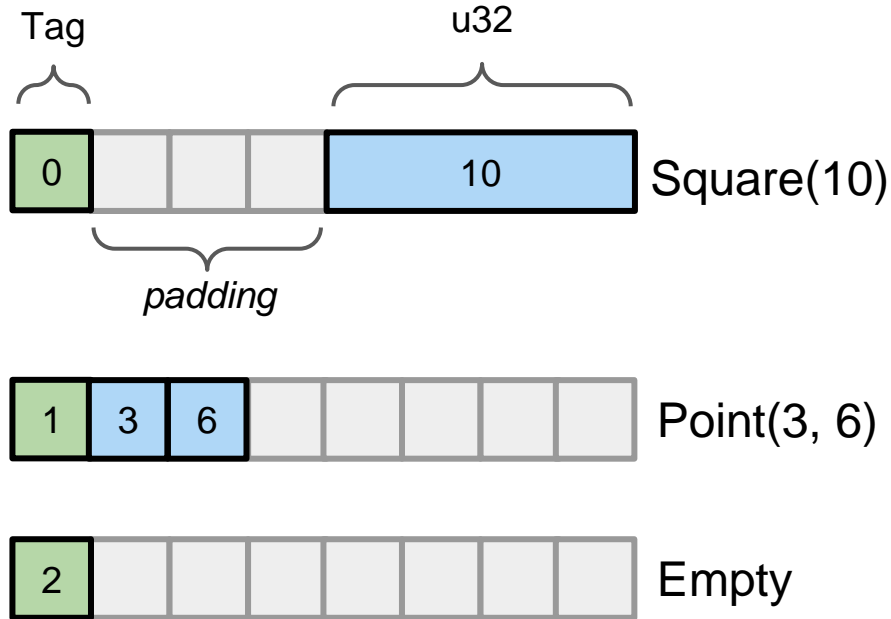
```
enum HttpResponse {  
    Ok,  
    NotFound(String),  
    InternalError {  
        desc: String,  
        data: Vec<u8> },  
}
```

# Enum

- Enum è definito come tipo **somma**
  - L'insieme dei valori che può contenere è l'unione dei valori delle singole alternative
  - Per contro, struct è un tipo **prodotto**: l'insieme dei valori che può contenere è il prodotto cartesiano degli insiemi legati ai singoli campi
- La possibilità di legare, agli specifici valori, uno o più dati è alla base di molti pattern di programmazione tipici di Rust
  - Ad esempio, la gestione dell'opzionalità e la rappresentazione del risultato di una computazione (che può fallire)

# 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,  
}
```

# Enumerazioni e clausole 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,  
    }  
}
```

# Destrutturazione

- 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);  
    }  
}
```

# Destrutturazione

- 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);
```

# Destrutturazione

- 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
  - $\text{Vec}<T>$ , 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
  - **$\text{Option}<T>$**  - rappresenta un valore di tipo  $T$  **opzionale** (ovvero che potrebbe non esserci)
  - **$\text{Result}<T, E>$**  - rappresenta **alternativamente** un valore di tipo  $T$  o un errore di tipo  $E$
- **$\text{Option}<T>$**  contiene due possibili valori
  - **$\text{Some}(T)$**  - indica la presenza e contiene il valore
  - **$\text{None}$**  - indica che il valore è assente



# Enumerazioni generiche

- **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"),  
    }  
}
```



## Per saperne di più

- Rust Basics: Structs, Methods, and Traits
  - <https://medium.com/better-programming/rust-basics-structs-methods-and-traits-bb4839cd57bd>
- Mastering Enums in Rust: Best Practices and Examples
  - <https://sterlingcobb.medium.com/mastering-enums-in-rust-best-practices-and-examples-a0bd76ea8cf>
- Enums and Pattern Matching in Rust
  - <https://medium.com/better-programming/rust-enums-and-pattern-matching-177b03a4152>
- Destructuring
  - <https://aminb.gitbooks.io/rust-for-c/content/destructuring/index.html>
  - [https://aminb.gitbooks.io/rust-for-c/content/destructuring\\_2/index.html](https://aminb.gitbooks.io/rust-for-c/content/destructuring_2/index.html)