`Rust::possesso`

Le caratteristiche di `Rust` descritte in questo capitolo, come ad esempio il possesso ed il movimento, sono quelle che fanno del linguaggio un linguaggio di programmazione che sia *memory safe*. L'obiettivo finale è quello di introdurre dei **meccanismi** per evitare i tre problemi legati all'allocazione della memoria: (i) memory leakage, (ii) double free, (iii) dangling pointer.

Sommario

Ownership

Nel linguaggio di programmazione `RUST` ogni variabile del programma è **posseduto** da una ed uba sola variabile. Il compilatore della *toolchain del Rust* ha un modulo logico chiamato **Borrow Checher** che si occupa di verificare che questo **vincolo** sia effettivamente rispettato $\Longrightarrow$ ogni violazione di questa regola porta ad un errore di compilazione (eg. tentativo di associare un valore a 2 variabili diverse).

Il **possesso** di una variabile implica essere responsabili del suo rilascio, cioè se ad esempio alloco della memoria dinamica per un programma, questa deve essere rilasciata *restituendola* al sistema operativo in modo che può essere assegnata ad altri processi che la richiedono.

Ok, ma quando avviene il rilascio della variabile? E' importante sottolineare che la memoria associata ad una variabile viene rilasciata in uno di questi due casi:

1. La variabile esce dallo *scope sintattico*
2. La variabile viene riassegnata con un'altro valore

C'è però una *eccezione*: il rilascio può essere rimandato quando il valore viene **mosso** in un'altra variabile che a questo punto sarà *responsabile* del rilascio della memoria.

Per chiarire questo importante aspetto consideriamo il seguente esempio:

```
fn main{
    let mut v = Vec::withcapacity(4);

    for i in 1..5{
        v.push(i);
    }
    println!("{:?}", v);
}
```

La variabile `v` è un `Vec` di interi, essa viene dichiarata con capacità 4 e poi riempita tramite un ciclo for. Finché la variabile è nello scope, le sue risorse risultano *sempre accessibili*, qunado esce dal proprio scope sintattico, ci si occupa di rilasciare le risorse possedute, in questo caso il **vettore allocato sullo heap** con tutto il suo contenuto.

Movimento

Il **movimento** è uno dei concetti che rende il Rust un linguaggio di programmazione più robusto rispetto agli altri che hanno caratteristiche generali simili.
Quando una variabile viene inizializzata **prende possesso** del relativo valore, MA se questa viene riassegnata la precedente viene *rilasciata* e la variabile prende possesso del nuovo valore.

Lasciamoci guidare dal seguente esempio per capire questo importante meccanismo:

```
let mut my_box=Box::new(4);
my_box=Box::new(6);
println!{"{:?}", my_box};
```

Qui la variabile `my_box` viene inizializzata con `Box::new(4)` quindi viene creato un puntatore sullo stack e sullo heap viene allocata una variabile intera. Quando alla riga di codice successiva riassegniamo la variabile: la memoria occupata precedentemente da `4` viene rilasciata, mentre `my_box` diventa possessore del nuovo valore allocato.

In generale, se una variabile viene **assegnata** ad un'altra variabile, **passata come parametro** ad una funzione, il suo contenuto viene **MOSSO** nella destinazione questo implica che:

- La variabile cessa di possedere il valore (non ha più la responsabilità del rilascio) che invece **è posseduto dalla variabile destinazione**
- La variabile originale resta allocata finché non esce dallo scope sintattico, ma il suo utilizzo è soggetto alle seguenti restrizioni:
  - Il compilatore considera la variabile *non inizializzata* (uninitialized)
  - Prudentemente Rust non lascia utilizzare le variabili in questo stato per cui un accesso in **lettura** a queste variabili genera un errore di compilazione.
  - Eventuali accessi in **scrittura** invece, avranno successo e ne riabilitano la lettura.
- La variabile destinazione conterrà una **copia bit a bit** del valore originale.

Esempio:

```
let mut my_string = "hello".to_string;
let my_string2 = my_string;
println!("{:?}", my_string2);
//da qui in poi my_string non è utilizzabile in lettura
```

Dopo la sua dichiarazione e inizilizzazione il valore di `my_string` viene mosso, `my_string2` ne prende il possesso.
Negli esempi fatti finora abbiamo utilizzato l'inizializzazione e l'assegnazione per introdurre il concetto di *Movimento*, ma Rust applica la semantica del movimento a qualsiasi uso di un valore. Ad esempio:

- Quando un valore viene passato ad una funzione, il possesso viene trasferito al *parametro attuale* della funzione.
- Quando una funziona ritorna un valore, il suo possesso viene trasferito al chiamante.

Bisogna inoltre prestare particolare attenzione alle situazioni in cui nel codice ci siano:

- Strutture condizionali in cui può avvenire il movimento $\iff$ le variabili mosse lasciano comunque il proprietario in una condizione *uninitialized*

```
let r=Vec::with_capacity(5);
if(<Condizione>){
    g(x)        //MOVE x to the parameter of g
}
else{
    h(x)        //MOVE x to the parameter of h
}
```

- Strutture cicliche, se ci sono chiamate a funzioni nel ciclo e c'è un movimento, questa rende inaccessibile la variabile passata alla funzione per le successive chiamate.

```
let r=Vec::with_capacity(4);
while(<Some_condition>){
    g(x)        //NOO!
}
```

Spesso si usa dire che quando avviene un *movimento* questo corrisponde a **consumare** il valore di una variabile.

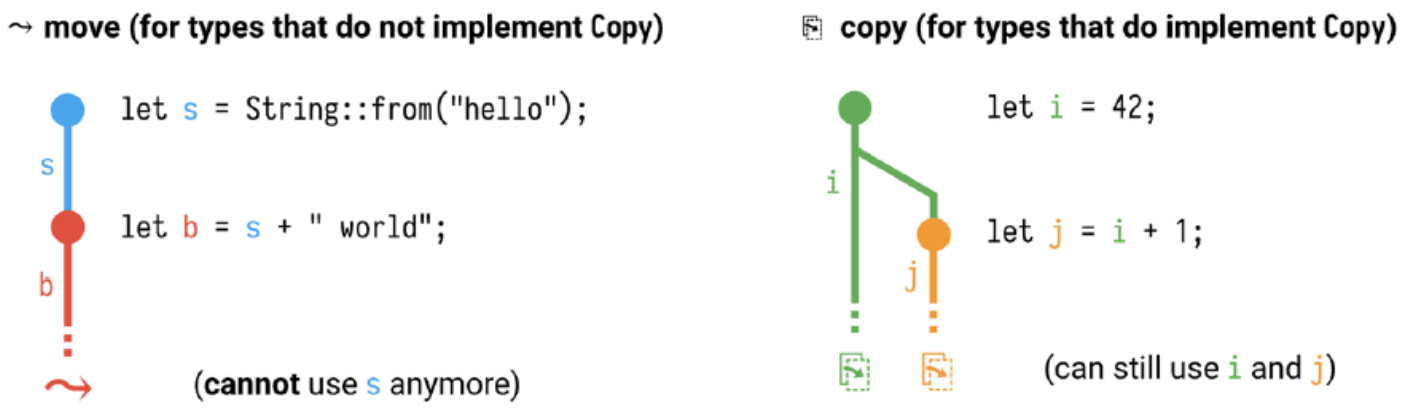
Tipi `Copy` : eccezioni al movimento

Gli esempi fatti finora circa valori che venivano mossi riguardavano variabili di tipo `Box` , `String` , `Vec` e altri tipi che richiedendo tanta memoria, la loro copia richiederebbe uno spreco di risorse.
Alcuni tipi invece, tra cui quelli numerici, sono definiti **copiabili** e hanno alcune caratteristiche particolari:

- Implementano il tratto `Copy`
- L'assegnazione ad un'altra variabile non lascia in stato *uninitialized* l'altra variabile
  Questo è possibile quando il valore contenuto nella variabile non costituisce una risorsa che richiede il *rilascio* esplicito.
  I **tipi semplici** e le loro combinazioni (esempio: tuple e array di numeri) sono copiabili, inoltre sono copiabili i **riferimenti a valori non mutabili**, mentre non lo sono quelli per valori mutabili.

Copia e movimento

La figura seguente mostra la differenza tra **Copia** e **Movimento**



Nel caso in cui c'è moviemento una semplice assegnazione rende la variabile di partenza non inizializzata, nel caso in cui il valore contenuto è copiabile e avvenga un assegnazione, posso continuare ad usare entrambe le variabili senza problemi → non c'è bisogno di gestire il rilascio di risorse allocate dal SO. Per concludere questo paragrafo forniamo qui un ulteriore esempio:

```
let my_string="Ciao sono Carlo".to_string();
let my_string2 = my_string; // --> MOVIMENTO di my_string

let my_num=5;
let my_num2 = my_num;       // --> COPIA di my_num
```

Clonazione

In `Rust` la clonazione è un comportamento descritto atttraverso il tratto `Clone`, in particolare il metodo `clone()` può essere chiamato affinché avvenga una *duplicazione del valore*. Dal momento che la clonazione può portare ad una **deep copy** il suo costo può essere elevato. L'operazione di clonazione è modificabile dal programmatore, mentre copia e movimento non sono modificabili e fanno uso della funzione `memcpy()`.

> **Nota che...**
> In `Rust` il compilatore a fronte di un'assegnazione o di un passaggio di un valore ad una funzione - a meno che questo non sia copiabile - applica un *movimento*.
> In `C` l'unico paradigma disponibile è quello della copia, con tutti i problemi che questo può causare.
> In `C++` è possibile avere il movimento a patto che questo sia chiamato esplicitamente e che il tipo lo preveda/lo implementi. Non c'è però un equivalente del *Borrow checker* che possa garantire che tutti i vincoli sul possesso siano rispettati.

Riferimenti

Abbiamo visto che in `Rust` operazioni semplici come assegnare una variabile, passare un parametro ad una funzione possono renderla inutilizzabile. Per risolvere questi problemi, vengono introdotti diversi tipi di puntatore. Il *Borrow checker* si occupa che il codice nel suo complesso rispetti le regole circa **responsabilità** e **diritti** di chi li usa.

Riferimenti semplici `&` (Ref)

Un *riferimento* è un *puntatore in sola lettura* ad un blocco di memoria **posseduto da un'altra variabile**. Questo è uno strumento utile che:

- Permette di accedere ad un valore senza doverne diventarne proprietari
- Non deve esistere più a lungo di quanto esista il dato a cui punta
- Nel caso in cui si crei un puntatore ad una tupla ad esempio, il compilatore si occupa di dereferenziare applicando l'operatore '.'

Un riferimento **prende in prestito** l'indirizzo di memoria in cui esiste il valore, fino a che questo riferimento esiste non è possibile **modificare il valore** né tramite il riferimento né ricorrendo alla variabile che ne detiene il possesso.

Creazione di un riferimento

Un riferimento può essere creato dal dato originale o da un riferimento ad esso. In certe situazioni è il compilatore ad applicare l'operatore di *dereferenziazione*.

Proprietà dei riferimenti

I riferimenti sono un tipo **copiabile** (viene duplicato il puntatore), il modulo *Borrow checker* del compilatore si assicura che quel riferimento punti sempre ad una porzione di memoria valida.
Finché esiste anche solo un riferimento la variabile a cui punta NON può essere modificata.
I riferimenti permettono la realizzazione della politica **multiple reader** di un valore a cui si fa riferimento.

```
let tupla=("Carlo", "Alberto", "di", "Savoia");
let ref_tupla = &tupla;
//Stampo i primi due campi della tupla
println!("{} {}", ref_tupla.0, ref_tupla.1);
```

Riferimenti mutabile `&mut` (RefMut)

In `Rust` può essere creato **un solo** riferimento mutabile per volta questo viene creato con la sintassi `let r = &mut v` (si legge "Ref mut"). Finché esiste un riferimento mutabile:

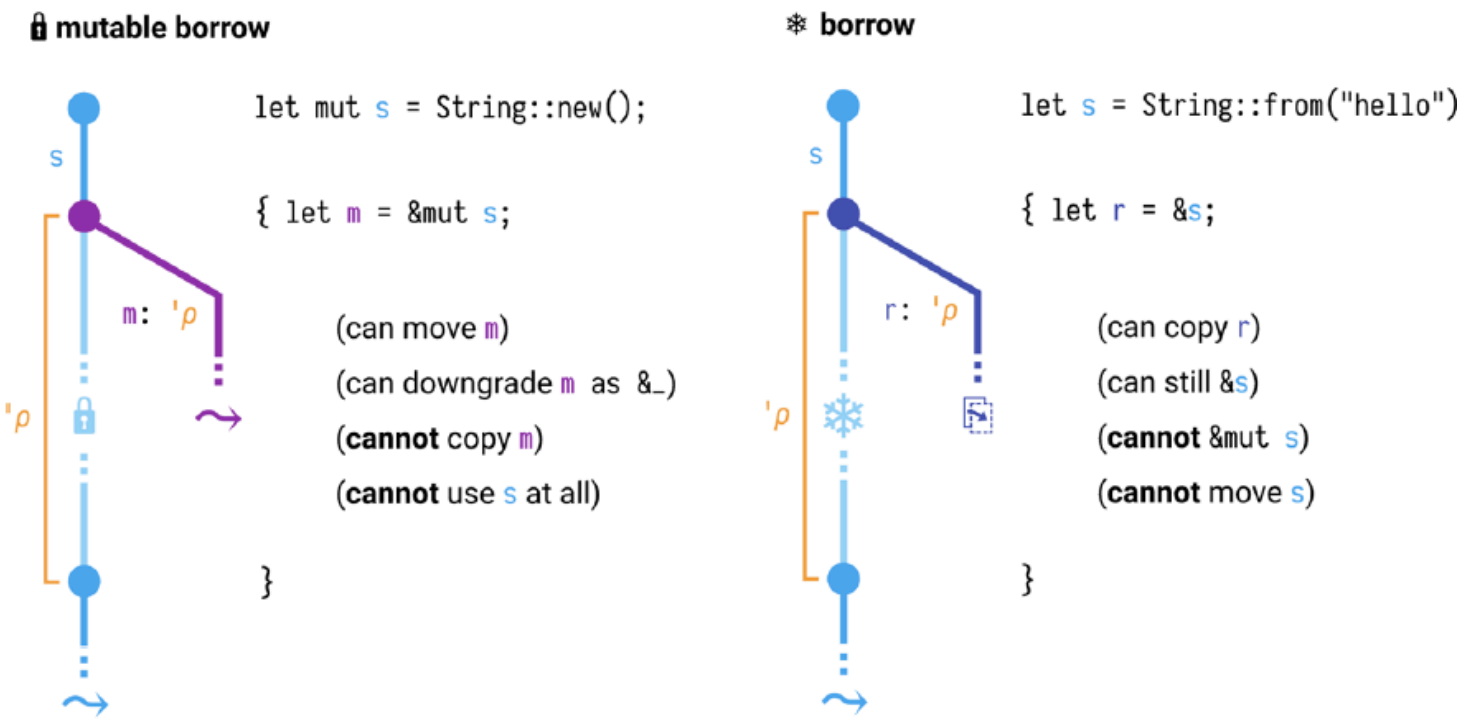
- Non possono esistere riferimenti semplici
- La variabile che possiede il valore NON può essere né modificata, né mossa.
- La variabile da cui si estrae deve essere dichiarata a sua volta mutabile con `mut` .
  I riferimenti mutabili NON sono copiabili, quindi una assegnazione, passaggio a funzione, ritorno a funzione... costituisce un movimento.

I riferimenti mutabili realizzano il paradigma **single writer**.

```
let mut tupla  = (0, 5);
let ref_mut = &mut tupla;
//modifica della tupla tramite riferimento mutabile
ref_mut.0 = 4;
ref_mut.1 = 6;
```

Ref vs RefMut

La figura seguente riassume in modo efficace i concetti che sono stati appena esposti.



Slice e possesso

Una **slice** `&[T]` è una vista di una sequenza contigua di elementi, in quanto tale non possiede i valori a cui fa riferimento che invece **appartengono sempre ad un'altra variabile**, il compilatore si assicura che i valori a cui si fa riferimento esistano quando i riferimenti vengono utilizzati.
Internamente viene rappresentata con una *tupla* costituita da:

1. Un puntatore al primo valore della sequenza
2. Un numero riferito al numero di elementi contigui a cui si fa riferimento.

Note conclusive

Possesso: Riassunto delle regole

1. Ciascun valore ha un unico possessore, esso viene rilasciato quando la variabile esce dallo scope o quando ad essa viene *assegnato un nuovo valore*;
2. E' possibile tuttavia creare dei riferimenti che sono:
   - *Semplici* e *immutabili*, possono esistere anche molteplici copie e puntano allo stesso valore, MA finché ne esiste almeno uno il valore puntato non deve essere modificato
   - *Riferimenti mutabili*, ne può esistere al massimo uno, non è copiabile (assegnazione $\rightarrow$ movimento), il valore non può essere letto finché esso esiste
3. Ogni tipo di riferimento deve esistere per un tempo **minore o uguale** del tempo in cui esiste il valore puntato.

Vantaggi derivanti dal concetto di possesso

- Un programma scritto in `Rust`, seppure non si serva del garbage collector, riesce a garantire un utilizzo 'sicuro' delle risorse in termini di *accesso* e *rilascio*.
- Non esiste l'equivalente di un **Null Pointer*, e quindi non c'è modo di dereferenziarlo.
- Tutte le variabili sono **immutabili**, serve un riferimento esplicito per renderle invece mutabili.
- Assenza di *Garbage collector* $\iff$ assenza di comportamenti non deterministici.