03-11.md

1. Lezione 11/03 - Possesso

Torniamo ora sul tema del possesso in Rust. Come già detto, una variabile usata nel linguaggio viene create in modo tale da avere in ogni momento dell'esecuzione un valore valido ed un proprietario. Controllare, inteso come possedere, una variabile, significa avere il potere leggerla e modificarla.

Il possesso di una variabile, inteso come la variabile che passa all'avere nessun proprietario, avviene nel momento in cui la sua memoria viene rilasciata, idiomaticamente detto `drop`. Quando si giunge al termine di un blocco di codice, detto `scope`, viene automaticamente eseguito il `drop` di tale variabile implicitamente. Il `drop`, come tutto in Rust, è definito tramite un tratto, il tratto `Drop`, che non permette di modificare il comportamento di liberazione della memoria ma permette di eseguire delle operazioni preliminari alla liberazione.

```
let x = 5;
{
    let b = 0;
    ...
    // chiamata implicita a drop(b)
}
drop(x);
// Ora x non è più una variabile valida
```

Notiamo una particolarità del linugaggio. In C, non esistendo il concetto di movimento di una variabile, ogni assegnazione viene eseguita per copia. Vale a dire, un'istruzione come `a = b`, copierà il contenuto di `b` in `a` lasciando inalterato `b`.

In C++ è pratica comune aggiungere un livello di astrazione, andando a definire un operatore di movimento, ovvero un operatore tale per cui, eseguendo `a = b`, il contenuto di `b` viene copiato in `a` e `b` viene svuotato. Questo viene più spesso definito per gli oggetti in C++ visto che per i tipi di base ha meno senso.

In Rust il comportamento è diametralmente opposto in quanto, a differenza del C e del C++, di base tutto avviene tramite movimento, e solo intenzionalmente possiamo andare a copiare il contenuto di un tipo complesso (vedremo avanti due tratti per raggiungere questo obiettivo) oppure solo esplicitamente passare la referenza all'aria di memoria, ovvero un riferimento.

```
let mut s1 = "hello".to_string();
let s2 = s1;
println!("s2: {}", s2);

// s1 non è più accessibile, vita in termine dato terminata
```

Tuttavia, dal punto di vista del linguaggio tra copia e movimento non esiste differenza per via della presenza del *borrow checker*, che semplicemente impedisce comportamenti che con il movimento non sarebbero possibili.

```
let point = (1.0, 1.0)
let reference = &point;
println!("({}, {})", reference.0, reference.1);
```

In questo caso `reference` puo' fare accesso alla variabile in sola lettura solo fintanto che la variabile `point` esiste. `reference` non possiede il dato quindi non puo' modificarlo e tantomeno non puo' distruggerlo. Per questo motivo i riferimenti sono copiabili, poichè è essenzialmente la copia di un puntatore. In più, a differenza di C/C++, il compilatore fa in modo che il riferimento non possa essere compilato

in una porzione di codice che mette in pericolo la vita del dato. Per questo esiste la differenza logica tra puntatore e riferimento, poichè il puntatore è un contenitore ad un indirizzo di memoria generico, non solo non tipizzato ma potenzialmente non valido, mentre un riferimento non solo è tipizzato ma grazie alla compilazione ci assicura sia sempre valido.

Per garantire questi meccanismi il linguaggio abilita il i riferimenti mutabili, ma solo una volta. Questo vuol dire ad esempio:

```
let v = "String"
let r = &v; // Ok
let r2 = &v; // Errore, solo UN riferimento mutabile è concesso
```

Questo perchè chiaramente l'operazione di mutabilità tira un po per le braccia il concetto di possesso, e quindi il linguaggio impone alcuni limiti. Per via di ciò, nel prestito è possibile eseguire la copia del riferimento mentre nel prestito mutabile è concesso solo il movimento in quanto solo un riferimento è concesso. Il prestito mutabile impone anche che la variabile passata in prestito non è accessibile in alcun modo perchè quella variabile, fintanto non riprende controllo del dato, è interamente rimossa dallo scope.

```
let mut s = String::from("Hello");

{
    let m = &mut s;
    println!("{}", s); // Errore, s non esiste come variabile
}

println!("{}", s); // Ok
```

Problema comune del linguaggi C è anche che i puntatori non hanno idea di che tipo di dato stiano puntando. Per fare un esempio, un `int*` potrebbe essere:

- Un riferimento a un singolo intero in memoria
- Il primo elemento di un array di interi
- I primi byte di un dato più lungo, tipo `long*`

In Rust tale comportamento "agnostico" non è concesso, infatti tutti i puntatori, in Rust dette referenze, sono talvolta chiamati *fat pointers*, poichè oltre ad avere appresso l'indirizzo dove il dato è memorizzato, portano con se dei dati accessori per sapere le dimensioni dell'area di memoria e del tipo di dato.

È interssante prendere un attimo per discutere ad esempio dei duppi puntatori, in C banalmente utilizzati per ottenre ad esempio delle matrici.

Il tipo `dyn` in Rust permette di definire che non si conosce esplicitamente il tipo di dato puntato ma si sa che implementa alcuni tratti e che quindi espone alcune funzioni. La refernza a un tipo `&dyn` quindi, che potrebbe essere di qualunque tipo, esplica quindi solo le funzionalità che esso puo' offrire.

Ma se non conosce il tipo, e quindi non ne conosce nemmeno la dimensione, quello che si viene a creare in memoria è qualcosa di simile a questo.

(Image: *../imgs/2022-03-11-15-26-59.png*)

Ovvero il puntatore doppio, o comunque annidato, è diviso in due parti: la prima che punta realmente al tipo di dato, la seconda che punta a una tabella virtuale dove sono presenti le chiamate funzioni cui il tipo puo' accedere. Questo accade perché, a differenza di quanto avviene in C o C++, il compilatore non per forza va a sotituire le chiamate a funzioni con delle istruzioni di salto all'atto della compilazione, andando invece a recuperare il puntatore a dove quella funzione è memorizzata in memoria quando il programma è in esecuzione.