03-09.md

1. Lezione 09/03 – Introduzione a Rust II

Come già detto Rust offre un ricchissimo sistema di tipizzazione, che è necesario esplorare prima di addentrarsi in dettagli funzionali ben più complessi. Tuttavia, essendo Rust come già detto non un vero linguaggio ad oggetti, esso non ha una struttura gerarchica dei tipi di dato, ma raggiunge le sue funzionalità tramite i tratti, su cui il linguaggio si basa e usa in maniera intensiva, di cui parleremo più avanti.

Parlando del valore di `null` in Rust, abbiamo detto che è noto come `Unit` ed è indicato dal simbolo `()`. In realtà `()` sta ad indicare un tipo ben specifico di dato, la tupla, che è a nostra disposizione in maniera nativa. Segue le regole descritte finora delle variabili e permette etorgeneità dei dati. È possibile accedere al dato in una data posizone come `tupla.n` con `n` numero del dato cui si vuole accedere

```
let t = (1, 2, 3);
let x = t.1; // x = 2
t.0 = 4; // No! La tupla è immutabile
let mut t2 = (1, 2, 3);
t2.0 = 4; // Ok! La tupla è mutabile
let t3 : (i32, mut i32, i32) = (1, 2, 3);
t3.1 = 4; // Ok! La tupla è immutabile ma il suo secondo elemento è mutabile
let t4 : (i32, f32) = (1, 2.0); // Totalmente ok
```

Tuttavia, l'interesse principale ai tipi di Rust è da dedicarsi ai rierimenti perché sono strettamente legati alla funzionalità più unica del linugaggio, il Borrow Checker.

In Rust deve essere in ogni momento definito il proprietario di una variabile, ma questo proprietario puo' prestare una variabile ad un altro, in due modi doversi: prestito mutabile e prestito immutabile. Va da sé che un prestito mutabile permetterà la modifica della variabile ed un prestito immutabile no.

La cosa cui dobbiamo stare attenti è che, esattamente come non è asolutamente problematico che molte persone leggano da una singola variabile costante potrò cederla infinite volte, mentre un prestito immutabile deve essere fatto uno per volta e con nessuno che possegga un prestito immutabile nel mentre. Si indica con `&var` il prestito immutabile e con `&mut var` il prestito mutabile.

```
let x = 10;
let y = &x;
println!("{}", y); // 10
println!("{}", x); // 10
```

Nel caso descritto qui sopra, `x` rimane il proprietario mentre `y` la ha solamente in prestito. Questo non crea problemi con l'esecuzione delle due print in quando tutte le operazioni che stiamo eseguendo sono solo ed unicamente in lettura. La situazione cambia radicalmente quando:

```
let mut x = 10;
let y = &mut x;
println!("{}", y);
println!("{}", x);
```

Questo codice non passerebbe la compilazione in Rust. Anzitutto notiamo l'aggiunta della parola chiave `mut` dinanzi ad `x`: questo è dovuto al fatto che ovviamente non è possibile cedere un prestito mutabile di una variabile che è immutabile. Tuttavia la compilazione fallierebbe per via della `println!` su `x`. Questo perchè internamente per richiamare la funzione di stampa è necessario prendere in prestito immutabile la variabile; tuttavia come descritto sopra, i prestiti mutabili sono incompatibili con i prestiti immutabili, il borrow checker si accorge di questa situazione e fa fallire la compilazione.

Sebbene quindi fin qui i riferimenti in Rust possano apparire molto simili ai puntatori del C++, alcune differenze sono nette. Infatti Rust che si pone come obiettivo quello di eliminare i comporatamenti indefiniti, rende impossible la creazione di un caso in cui si abbia un riferimento nullo. L'utilizzo delle referenze in modo esclusivo, secondo una politica singolo scrittore **o** multipli lettori, garantisce questa particolarità.

Quanto detto finora vale per le variabili locali, tuttavia è comune una casistica in cui si voglai avere un oggetto allocato sulla memoria di heap, magari per estenderne il tempo di vita. In quel caso il linguaggio offre il tipo `Box<T>`, che è anche quindi il primo esempio di programmazione generica su tipo `T` che andiamo a vedere. Il tipo dato `Box` è molto più simile a quello che definiremmo un puntatore in C++, e infatti come anche essi permette la dereferenziazione per l'accesso al dato mediante l'operatore `*`.

```
let v = vec![1, 2, 3]
{
    let box = Box::new(v);
    println!("{}", *box); // "1, 2, 3"
}
// `box` distrutta, `v` rilaciato
```

Nel caso delle Box, il dato da loro contenuto terminerà la vita quando la referenza della Box verrà distrutta. Tuttavia in alcuni casi estremi, puo' essere necessario l'uso di puntatori nativi stile C. Per questi casi Rust definisce una parola chiave da usare con cautela, denominata `unsafe`, che non viene sottoposta al controllo del Borrow Checker. In questo tipo di blocchi è consentito l'uso di puntatori come `*const T` e simili, ma bisogna evitare di utilizzarli quando non strettamente necessario (quasi mai).

Passiamo ora agli array, che in Rust sono resi idiomaticamente ricchi. I modi per dichiarare un array possono inizialmente confondere, questo perché ci sono più modi di dichiarare ed inizializzare un array in una sola riga a seconda delle esigenze. È bene notare che ciò scaturisce dal fatto che in Rust, a differenza di quanto accade in C++, non è possibile instanziare un array senza definirne il contenuto.

```
let a : [i32; 5] = [3, 3, 3, 3, 3];
let b = [3; 5];
```

Questi sono due modi del tutto equivalenti di produrre lo stesso array. Nel primo caso annotiamo il tipo e la sua lunghezza, per poi successivamente definirne il contenuto. Nel caso volessimo tutti elementi uguali, come qui, la seconda scrittua permette di creare un array secondo la notazione `[valore_iniziale; lunghezza_array]`.

Se invece avessimo bisogno di un array di dimensione variabile, quindi la cui dimensione non è necessariamente nota in fase di compilazione e con la possibilità di essere eventualmente allargato, si puo' ricorrere al tipo `Vec<T>`.

Anche in questo caso, gli array vivono nello stack mentre i Vec vivono nello stack. Per instanziare un `Vec`, è sconsigliato l'uso del costruttore `Vec::new()` che porterebbe a un vettore inizialmente nullo che dovrà essere riallocato più volte inizialmente, ma è consigliabile utilizzare la comoda macro `!vec[]` oppure la funzione `with_capacity` per dichiarare da subito una dimensone minima da poter sostenere.

```
let v1 = vec![1, 2, 3];
let v2 : Vec<i32> = Vec::with_capacity(3);
```

Anche qui come nei casi precedenti il tipo viene automaticamente dedotto quando possible. Nella seconda riga il compilatore non riuscendo a farlo potrebbe avere bisogno dell'annotazione.

Notiamo una cosa:

```
let v : Vec<i32> = Vec::with_capacity(3);
let x = v[0];
println!("{}", x);
```

Notare che il codice qui sopra è un codice che passerebbe tranquillamente la fase di compilazione di Rust, nonostante non si sappia che valore `x` abbia e abbiamo finora parlato dei grandi sforzi per non ottenere comportamenti indefiniti. Tecnicamente parlando, il comportamento in questo caso è ben definito: facendo accesso fuori posizione al vettore, il programma chiamerà la macro `panic!()` che arresterà brutalmente in programma.

Un ultimo modo per fare riferimento agli array sono le Slice. Esse sono analoghe a come presenti in altri linguaggi che le hanno popolarizzate, come Python, e permettono appunto di ottenere un riferimento a una porzione di un array la cui dimensione potrebbe non essere nota in fase di compilazione.

Le slice vengono inoltre gestite come dei `fat pointers`, che indicano all'interno della variabile cui sono memorizzati l'area di memoria dove effettivamente risiedono e la loro dimensione.

```
let v = [1, 2, 3, 4, 5];
let s = &v[1..3]; // Operatore di range, vedere avanti

println!("{}", v); // 1, 2, 3, 4, 5
println!("{}", s); // 2, 3
```

Parliamo ora delle stringhe, una delle parti dove il design di Rust è più vivido che mai. Di base abbiamo due tipi ben speficific: `String` e `str`.

`str` rappresenta dei tipi di caratteri con codifica Unicode, memorizzati staticamente. Non è un tipo direttamente manipolabile quindi per utilizzarli come stringa bisogna far riferimento alle loro slice, `&str`.

`String` invece è più un tipo da intendere come un buffer. A differenza di `&str` la sua dimensione non è infatti fissa e puo' espandersi quando necessario. Fare attenzione perchè, essendo appunto un buffer, ha a disposizione due chiamate a funzioni per ottenere la sua lunghezza: una restituisce la dimensione del buffer, l'altra il numero di caratteri in esso presenti. Non solo, ma internamente sia `&str` che `String` sono implementate come un buffer `&[u8]`, ovvero una serie di byte, quindi anche quando il buffer è interamente pieno, è garantito che le due dimensioni non coincidano.

La presenza di questi due tipi puo' inizialmente sembrare strana, soprattutto dovendo scegliere quando usarli. La realtà dei fatti tuttavia, è che spesso è necessario usarli entrambi: per le chiamate a funzioni, o comunque per operazioni che non richiedano la modifica della stringa, è preferibile l'uso di &str, mentre per quel che riguarda la manipolazione delle stringhe, essendo le slice di dimensione fissa, bisogna utilizzare String.

Passiamo ora a parlare di istruzioni per l'iterazione. Il costrutto `while` permette di iterare il blocco fintanto che la condizione è verificata:

```
let mut n = 0;
while n < 10 {
    n += 1;
}
println!("{}", n) // 10
```

mentre i costrutto `for` è ispirato a linguaggi moderni che hanno abrogato l'accesso tramite indice, usando implicitamente gli iteratori, di cui parlerermo approfonditamente più avanti.

```
let vec = vec![1, 2, 3];
for v in vec {
    println!("{}", v);
}
// 1... 2... 3...
```

Aggiunge tra le altre cose un operatore per creare cicli infiniti, ovvero da rompere esplicitamente usando una istruzione `break`.

```
let mut n = 0;
loop {
    n += 1;
    if n == 10 {
        break;
    }
}
println!("{}", n) // 10
```

Questi tipi di cicli possono anche essere annidati ed etichettati, in modo da poter rompere un ciclo esterno da un ciclo interno:

```
let mut n = 0;
'outer : loop {
    'inner : loop {
        break 'outer;
    }
    n = 100;
}
println!("{}", n) // 0
```

Volendo tuttavia utilizzare qualcosa di più simile ai metodi tradizionali del C, anche sol puramente per comodità, il linguaggio rende più conciso anche quello grazie al tipo `Range`. Esso infatti ci permette di definire delle sequenze crescenti di valori tramite gli operatori `..` e `..=`. Nel primo caso, l'ultimo numero è escluso dall'insieme, mentre nel secondo è incluso.

```
let inclusive_range = 0..10 // Ultimo elemento: 9
let exclusive_range = 0..=10 // Ultimo elemento: 10
let even_nums = 0..=10.step_by(2) // 0, 2, 4, 6, 8
```

Non è possible creare dei range decrescenti, ma è possibile iterare su di essi al contrario.

```
let inclusive_range = 0..=10;
let reversed = inclusive_range.rev(); // 10, 9, 8, ...
```

Un'istruzione singolare in Rust, per via della sua potenza, è il costrutto `match`. Esso è analogo a quello che era la parola chiave `switch` in C/C++, ma è molto più forte, tanto che intere funzioni molto complesse sono riscrivibili in termini di solo `match`.

```
let x = random_number();
match x {
    0 => println!("zero"),
    10..=20 => println!("ten to twenty"),
    value if value < 5 => println!("less than five"),
    30 | 60 => println!("thirty or sixty"),
    _ => println!("other"),
}
```

Infine, il lingaggio ha anche una sua convenzione sull'estetica del codice. Infatti se fin'ora il codice ha seguito uno stile di scrittua `snake_case` è perché questa è quella ufficialmente adottata dal compilatore che, nel caso in cui venga infranta, rilascia dei warning. Infine, tramite il comando `cargo fmt`, è possibile eseguire la formattazione, indentazione, etc. di tutto il codice sorgente del progetto.