04-05.md

1. Lezione del 05/04

Nota sulla lezione precedente: puo' capitare in Rust come spesso accade in altri linugaggi orientati agli oggetti di voler implementare dei "metodi" getter o setter. In questo caso le scelte da farsi son poche: per implementare il metodo get è strettamente necessario esplicitare il primo parametro come reference non mutabile, ovvero `&self`.

Non è possibile usare solo `self` in quanto ciò causerebbe movimento, con perdita della struttura cui si vuole utilizzare il dato. Sarebbe possibile usare anche una referenza mutabile ma la cosa è sconsigliata.

Per i metodi setter invece è necessario che non solo la referenza acquisita sia mutabile, quindi `&mut self`, ma che anche la variabile che contiene la struttura sia dichiarata come tale, altrimenti il compilatore non ci lascerebbe modificare un dato dichirato come immutabile.

Abbiamo parlato l'ultima volta di come ridefinire l'operazione di indicizzazione secondo il tratto `Index`. Questa operazione è definibile solo una volta per tipo, vale a dire che non posso eseguire una doppia indicizzazione di un tipo di dato direttamente. Per fare ciò, nell'esempio più semplice possibile volendo definire una struttura per contenere una matrice di interi, è necessario definire due strutture.

Si puo' scegliere a piacere tra due opzioni nel caso di una matrice, ma banalmente il da farsi è definire due strutture, una `Matrice` e una `Riga` oppure `Colonna`, a seconda della scelta della seconda l'operazione di indicizzazione su `Matrice m[i]` restituirà un `Riga` o una `Colonna` su cui richiamare nuovamente l'indicizzazione per ottenere il dato.

Quando si ha un tratto sono in grado di derivare le funzionalità di dereferenziazione, ovvero di intendere il dato come fosse quel che in C era un puntatore e accedere al dato puntato tramite l'operatore `*`.

```
struct Selettore {
    elementi: Vec<String>,
    corrente: usize,
}

impl Deref for Selettore {
    type Target = String;

    fn deref(&self) -> &String {
        &self.elementi[self.corrente]
    }
}
```

Cosa significa dereferenziare una sturttura, come si vede sopra, dipende dall'implementazione che si vuole avere.

Un altro tratto interessante è quello che ci permette di definire il comportamento di una conversione di tipo. Banalmente potremmo voler implementare la conversione di una nostra struttura al tipo String per poterne definire una eventuale funzione `to_string`. Per fare ciò esistono due tratti, quello `From` e quello `Into`, che permettono rispettivamente i attuare la conversione dal dato al nostro tipo o dal tipo al nostro dato.

```
impl From<[i32; 2]> for Punto {
    fn from([x, y]: [i32, 2]) -> Punto {
        Punto { x, y }
    }
}
```

Notiamo che nel momento in cui tentiamo di stampare una nostra struttura il linguaggio non vada a vedere se esiste una implementazione di conversione verso il tratto `&str` o `String`, ma bensi abbia due tratti appositi per l'operazione, ovvero il tratto `Debug` ed il tratto `Display`. Il primo viene richiamato nel caso di stampa in debug, ovvero interpolazione di una stringa contentente i caratteri `{:?}` mentre il secondo viene usato per interpolazioni verso i caratteri `{}`.

```
#[derive(Debug)]
struct ErrorePersonalizzato {
    info: String
}

impl std::fmt::Display for ErrorePersonalizzato {    // Derivazione manuale
    fn fmt(&self, f: &mut std::fmt::Formatter) -> std::fmt::Result {
        write!(f, "{}", self.info)
    }
}

impl std::error::Error for ErrorePersonalizzato {
    fn description(&self) -> &str {
        &self.info
    }
}
```

Notiamo come la macro `[#derive(Tratto)]` riduca considerevolmente il codice da scrivere e quindi anche le probabilità di errori banali. Questa operazione non è sempre possibile, poiché per quanto moderno ed avanzato possa il compilatore essere non è comunque perfetto, ma per casi banali che vanno a comporre una larga detta del codice che poi realmente si usa questoè quasi sempre possibile.

Una caratteristica tipica dei linguaggi ad alto livello sono i tipi generici. Il contenitore `Vec` ad esempio ha un comportamento invariato indipendentemente dal tipo di dato che contiene. In Rust tuttavia, così come accade anche in C++ ma non in tutti i linguaggi ad oggetti, la gestione del tipo generico non avviene in fase di esecuzione ma in fase di compilazione.

Prendiamo ad esempio Java. Quando il linguaggi si trova a gestire un tipo generico, attinge ad un'unica sorgente di istruzioni macchina, e tratta il tipo generico come solamente una referenza, risultando in qualche modo agnostico al tipo di dato che ospita. Lo stesso comportamento si puo' ottenre in C tramite l'utilizzo di `void *`.

Rust invece segue la filosofia del C++, dove pur di ottimizzare le prestazioni delle operazioni che si eseguono, definita una struttura dati generica `Struttura<T>` e utilizzandola poi con due tipo, `A` e `B`, esso duplicherà il codice di gestione della struttura per i due tipi, inserendo quindi nel nostro linguaggio macchina il caso specifico per gestire `Struttura<A>` e `Struttura<B>`. Non è detto che i due codici risultanti siano identici a meno del tipo, perché a seconda delle loro caratteristiche diverse ottimizzazioni potrebbero venir compiute dal compilatore.