03-08.md

1. Lezione del 08/03 - Introduzione a Rust

Il C è stato e per alcune cose è tutt'ora il linguaggio per eccellenza della programmazione basso livello. Quando fu inventato, grazie ai compilatori, si poté finalmente aggiungere un livello di astrazione al proprio codice rispetto all'assembler, che richiedeva una conoscenza puntuale dell'architettura, finanche dei registri, comunque garantendo le prestazioni ed il controllo sull'hardware a basso livello.

Tuttavia, nel tempo le esigenze dei programmatori sono cambiate, dovuto al fatto che la complessità dei programmi da creare è cresciuta notevolmente nei decenni. Durante il corso vedremo anche esempi di C++, linugaggio che nasce come un *super-set* teorico del C per introdurre il paradigma di programmazione a oggetti, ma non si limita ad essere solo questo dato che è anche uno dei linugaggi più moderni nella sfera dei linguaggi compilati.

Tuttavia, il lingaggio che studieremo attivamente nel corso sarà Rust, un linugaggio che esattamente come fece C++ nei riguardi del C nasce molto dopo il C++ (C nasce nei primi anni 70, C++ nasce a fine anni 80, Rust nasce a inizio anni 2000) e che come tale punta a risolvere quelle criticità di design che sono emerse nel corso degli anni in C++.

Rust è un linugaggio molto moderno anche nella suite di tool offerti da esso. Infatti offre supporto nativo per il testing e la gestione di librerie di terze parti, due ambiti dove in C++ la tua miglior speranza è scaricare un `.h` ed un `.c` da internet nella speranza che siano corretti e che siano compatibili con il tuo sistema.

Seppur definito linugaggio ad oggetti, Rust non cade precisamente in questa definizione. Una più corretta nomenclatura per Rust sarebbe infatti un linguaggio a tipi, dato che implementa anche di base un ricchssimo sistema di tipizzazione. Tuttavia, per lo stile di programmazione che generalmente si usa, Rust è impropriamente definibile un linguaggio ad oggetti.

La principale evoluzione di Rust rispettto al C++ sta nella sua catena di compilazione. Se infatti essa è appoggiata ad LLVM, che è la stessa catena di compilazione di C++ e quindi ci garantisce fortissimi livelli di ottimizzazione, Rust aggiunge a tale catena un pezzo pittosto singolare che lo rende un linguaggio unico nel suo genere, vale a dire il Borrow Checker.

Il Borrow Checker è il meccansimo nativo che Rust ci offre come soluzione ai problemi di cui abbiamo parlato sin'ora, ovvero di validità e possesso della memoria. Esso si assicura che in ogni istante di esecuzione del mio processo le mie variabli e la memoria da loro utilizzata sia valida e abbia uno ed un solo proprietario. Questo meccanismo è imposto in fase di compilazione, quindi se queste due condizioni non sono assolutamente verificabili, la compilazione fallirà. Ad esempio:

```
void foo(void){
    Dummy *ptr = (Dummy *)malloc(sizeof(Dummy));
    ptr->a = 2048;
    free(ptr);
    ptr->a = 1024; // ?
    free(ptr); // ????
}
```

Prendiamo questo piccolo esempio didattico di C++, in cui sono presenti tutti i classici meccansimi di uso della memoria dinamica: allocazione, accesso, liberazione. Abbiamo deliberatamente inserito due errori di programmazione. Il primo è l'accesso a una zona di memoria ormai liberata: qui il programma potrebbe non arrestarsi, in altre parole il comportamento non è definito. Tuttavia abbiamo la certezza di incappare in un errore fatale, o `SEGFAULT`, chiamando la funzione `free` due volte di seguito sullo stesso puntatore.

Questo codice, di cui è facile verificare la correttezza anche per un programmatore non esperto, passerà comunque la compilazione in C++, forse generando al più un avviso. In Rust un codice equivalente non sarebbe stato compilabile. Rust offre anche la possibilità di tracciare altri tipi di errori in fase di compilazione, come l'underflow o l'overflow.

Per via di questo suo stringente controllo di validità, Rust è tra i linguaggi più stabili esistenti ad oggi, se non il più stabile in assoluto. Tuttavia questo stiringente controllo lo rende anche uno dei linguaggi con la curva più ripida in assoluto. Filosoficamente parlando, Rust è diametralmente opposto a C e C++: in questi due linugaggi infatti il programmatore ha la totale libertà di fare qualunque cosa voglia, ma puo' eventualemnte cedere parte di questa libertà per ottenere garanzie di correttezza. In Rust si parte invece dalla situazione con meno libertà in assoluto e volta per volta, come vederemo, se ne richiede quel tanto che basta per poter ottenere una funzionalità, fermo restando il controllo del compilatore.

Come già detto, installando Rust andremo a installare in realtà non solo `rustc`, che è il compilatore di Rust, ma anche `cargo`, che offre varie funzionalità tra cui la gestione dei panchi, lancio di suite di test, esecuzione dei programmi per monitorare le prestazioni.

Rust obbliga, oltre a una correttezza formale del codice, anche una certa forma nella composizione dei suoi progetti. Infatti per creare un progetto in Rust abbiamo due modi: il primo è il comando da CLI `cargo new <project_name>`, che andrà a creare una cartella con nome `<project-name>` e inizializzerà la sua struttura ad albero in un certo modo; La seconda è, a cartella già create e trovandoci all'interno di essa, il comando `cargo init`, e il progetto prenderà il nome della cartella dentro cui ci si trova.

Che si utilizzi `init` o `new`, varie cose verranno create all'interno della cartella di progetto. Oltre alle sottocartelle necessarie allo sviluppo (ad esempio in Rust tutto il sorgente va **necessariamente** all'interno di `<project_name>/src`) ed una repository git, troveremo un file `Cargo.toml` che andrà a descrivere il progetto.

```
[package]
name = "firstproj"
version = "0.1.0"
edition = "2021"

[dependencies]
```

Possiamo notare alcuni campi interessanti: oltre a `name` e `version`, abbastanza auto-esplicativi, troviamo infatti `edition` che ci indica quale versione del linguaggio usare e `[dependencies]`, sotto il quale va riportato l'elenco di pacchetti necessari al nostro programma.

```
fn main(){
    println!("Hello World!");
}
```

In Rust così come in C e tanti altri linguaggi, il nostro programma inizia la sua esucizone dalla funzione main. In Rust per dichiarare una funzione viene impiegata la parola chiave `fn`. A differenza del C, se non si specifica il tipo di ritorno, esso viene assunto essere `void`, in Rust detto `Unit`, indicato col valore `()`. Inoltre offre il ritorno di un valore implicito: infatti se non si inserisce il `;` alla fine di una funzione, il prodotto di quell'istruzione viene automaticamente restituito.

```
fn return_unit() {
    println!("Hello World!");
    // implicit return ()
}

fn return_i32() -> i32 {
    println!("Hello World!");
    return 42;
}

fn implicit_return_i32() -> i32 {
    println!("Hello World!");
```

```
    42
}
```

Rust a differenza del C permette di non dichiarare il tipo delle sue variabili. Questo compito verrà delegato al compilatore, ma è possibile che ci sia il bisogno di dichiararlo esplicitamente perché non riesca a dedurlo da solo (generalmente quando è così abbiamo fatto qualche boiata noi comunque). Per dichiarare una variabile si utilizza la parola chiave `let` e di base tutte le variabili in Rust sono immutabili, ovvero, costanti.

```
fn foo(x : u32) {...}
    ...
    let x = 5;
    let y : i32 = 5;
    foo(x);
    x = 10; // Non concesso
    ...
```

Analizziamo questo piccolo pezzo di codice. Notiamo subito che, nella definzione delle funzioni, è necessario specificare il tipo della variabile che si va ad accettare, in questo caso `u32` corrisponde ad un intero senza segno a 32 bit. Dopodichè, instanziamo due variabili. Nel primo caso non definiamo il tipo mente nel secondo decidiamo di definirla come `i32`, ovvero intero su 32 bit con segno. In questo contesto il compilatore, vedendo che passiamo `x` alla funzione `foo` che accetta un `u32` come parametro, assocerebbe ad `x` il tipo `u32`. Infine come detto tutte le variabili in Rust sono di base immutabili, quindi l'istruzione `x = 10` farebbe fallire la compilazione.

Per permettere che il valore di una variabile venga modificato è necessario usare la parola chiave `mut` in fase di dichiarazione della variabile.

Chiudiamo la lezione inoltre illustrando l'uso della macro (e non funzione, ne parleremo più avanti) `println!` che stampa un messaggio sullo standard output, interpolando i valori passategli come farebbe un'analoga `printf` in C o C++ usando il segna-posto `{}`.

```
let mut x = 0;
println!("x is {}", x); // "x is 0"
x = 10;
println!("x is {}", x); // "x is 10"
```