# 05-11.md

## 1. Lezione dell'11/05 – Crate e Testing

**Questa lezione non l'ho riscritta che mi pare inutile**

Il processo di gestione dei crate in Rust segue regole molto specifiche. Quando scriviamo il codice e cerchiamo di distribuirlo in più file, il compilatore sembra agire in modo strano. Questo perché un progetto è sostanzialmente equivalente ad una qualunque libreria esterna, ed è visto quindi come un crate che definisce cosa espone e le dipendenze di cui ha bisogno.

Con un crate io posso avere due possibili obiettivi: scrivere una libreria o scrivere un'eseguibile La differenza tra le due è ovviamente semplice.

Dovendo supportare entrambe le modalità, il compilatore deve supportare sia funzioni statiche che dinamiche, e fare in modo che l'eseguibile le richieda nel modo corretto.

La dicitura funzione al posto di libreria non è casuale. Infatti compilando un linguaggio in C in modo statico, in cui si è inclusa la `stdio`, operazione pressoché fondamnetale, allegherò nell'eseguibile tutta la `stdio`, anche le sue parti che in realtà non vado ad utilizzare. Se invece lo compilassi in modo dinamico potrei evitare di fare ciò, e l'eseguibile richiederà le funzioni al SO, e solo quelle che realmente andrà ad usare. Lo svantaggio di questa tecnica è la probabilità di incappare in qualche errore nel caso in cui esse non siano compatibili al 100%.

Dall'altro lato, l'utilizzo di librerie dinamiche aumenta leggermente i tempi di esecuzione nel caso in cui le parti di libreria richieste non siano già caricate in memoria quando vengono richieste, visto che il loro caricamento segue un principio di lazy loading.

In linea di massima, in sistemi general purpose come sono i nostri computer, vorrò nella quasi totalità dei casi un eseguibile con librerire linkate in modo dinamico.

Nel `.toml` potremo indicare una sezione `[lib]` che definisce come essa verrà eventualmente prodotta. Questo è utile perché ad esempio scrivendo una libreria in Python è possibile che io non sia capace di utilizzarla in un programma C.

Questo è dovuto al fatto che le librerie vengono compilate in modo da essere trasversali al linguaggio scelto usando delle interfacce comuni, ovvero quelle del C. In questa sezione quindi posso andare a definire se sto costruendo una libreria Rust pensata per essere usata puramente in progetti Rust o se ho la necessita di produrre simboli compatibili anche con C, e questo richiede da parte del compilatroe un po' di manipolazione extra.

I progetti di Rust sono divisi in moduli. Rust utilizza la parola modulo sia per definire qualcosa di fisico che qualcosa di virtuale, e questa cosa inizialmente puo' creare dei fastidi. Il loro funzionamento è simile ai `namespace` di C++, che è un modo per disambiguare funzioni che hanno lo stesso nome raggruppandole sotto etichette differenti. Ad esempio io potrei in un mio modulo C++ definire una funzione `cout`, e potrei usare entrambi nello stesso frangente di codice disambiguandole come `std::cout` e `mymod::cout`.

I moduli possono a loro volta avere dei sottomoduli per accedere a funzioni private non esposte all'esterno della libreria. Queste regole si applicano anche ai tratti.

I crate in rust si importano con una sintassi semplice: `use crate::modulo::funzione;` Questa operazione la eseguiamo anche per i tratti standard, a volte implicitamente visto che per i tratti standard più comuni anche se non lo scriviamo esplicitamente è il compilatore che va ad aggiungere l'import per noi.

Come altri linguaggi è possibile importare tutti i simbooli di un modulo tramite `*`, quindi facendo `use crate::modulo::*` importeremmo tutte le funzioni contenute in `modulo`. Il nome del file è

automaticamente il nome del modulo, quindi il file `mymod.rs` definisce il modulo `mymod`.

Un altro modo, vecchio ed oggi sconsigliato per quanto ancora supportato per quesitoni di compatibilità, è quello di creare una cartella `mymod` ed inserire al suo interno un file chiamato stettamente `mod.rs` con le nostre funzioni. È possibile dividere in più file all'interno della stessa cartella, affinchè questi simboli vengano dichiarati in `mod.rs` in quanto Rust esegue in prima istanza solo la lettura di quei file, e va poi eventualmente a cercarne altri.

La sezione `[dependencies]` è pensata raramente per essere gestita a mano, andando quindi a modificare manualmente i suoi campi. Questa parte è per lo più delegata a `cargo` o al nostro IDE.

In questa sezione andremo a inserire le dipendenze di cui la nostra libreria o eseguibile ha bisogno per funzionare, e per ogni dipendenza si puo' esplicitare una versione, in tutte le salse cui siamo abituati come una versione ben precisa, una versione che deve essere maggiore o minore di quella descritta, e così via.

In Rust le librerire hanno la possibilità di essere anche divise in `feature`, cioè una libreria puo' esporre tutte le sue funzioni subito oppure esporne alcune solo quando quella specifica funzionalità è richiesta.

Infine troviamo il campo `edition` che ci dice quale versione di Rust si sta usando e un campo `version` che indica la versione della libreria. La prima è descrittta dall'anno (ad esempio "2022") e la sconda a nostra descrizione seguendo le usuali convenzioni.

Passiamo ora invece a parlare di testing. Rust ne supporta due modalità: `test` e `benchmark`. Qui parleremo solo dei test, ma come il nome lascia suggerire i benchmark è l'esecuzione di un programma più e più volte per capire quanto la sua esecuzione è veloce, ed è utile al prorammatore per capire se le sue modifiche hanno impattato eccessivamente sulle prestazioni o meno, sia in modo positivo che in modo negativo.

Eseguire i test in C è complesso, questo perché il C nasce in un contesto in cui l'idea attuale di test non esiste ed esistono anche troppi standard del C per produrre un unico tool per eseguirli. Ad esempio, MISRA C non permette di passare ad una funzione più di 5 parametri, quindi se scrivessimo funzioni con 6 o più parametri il programma non potrebbe proprio essere compilato.

I test sono pensati per difidere il programma in funziono o funzionalità in modo da poterli provare individualmente in maniera isolata per poterne valutare la correttezza. L'utilità è ovviamente duplice in quanto questo torna utile sia in fase di sviluppo che in fase di mantenimento.

Altro aspetto importante è l'integrazione, perché il nostro software non è detto sia composto solo dal nostro eseguibile e dalle librerie che esso utilizza ma anche da altri eseguibili, e con i test posso anche valutare la corretta comunicazione di questi componenti.

I test di unità sono implementabili in più modi. Un primo è quello di andare a mischiare le funzioni che eseguono i test con il mio sorgente, e posso fare ciò come:

```
#[cfg(test)]
fn my_test_function(...) {
    ...
}
```

Questa funzione verrebbe eseguita solo in fase di test, richiamabile col comando `cargo test`.

Ovviamente mischiare test e sorgenti è un modo di operare piuttosto scomodo. Quindi quel che soltiamente si va a fare è porre i nostri test nell'apposita sottocartella che Rust crea in automatico per il nostro progetto, appunto `tests`, in modo da rendere più scalabile il tutto.

I test di unità si basano sul richiamare funzioni o funzionalità del nostro programma, solitamente questo richiederebbe che tali funzioni siano testati in modo più granulare e fino possibile, e verificarne la corretta computazione tramite degli `assert`.

Le funzioni di test, seppur riportate nell'apposita cartella, vanno indicate con la macro `#[test]`, in quanto potremmo voler definire anche in questi file funzioni che non sono dei test ma sono utili alla loro esecuzione, come una funzione di setup.

```
// src/mylib.rs
fn add_two(x: i32, y: i32) -> i32 {
    x + y
}
```

```
// tests/my_test.rs
#[test]
fn add_zero_zero() {
    assert_eq!(add_two(0, 0), 0);
}

fn add_num_zero() {
    assert_eq!(add_two(1, 0), 1);
}

...
```

Di base Rust supporta tre tipi di `assert`: - `assert`: verifica che il predicato sia `true` - `assert_eq`: veririca che i due valori passati siano uguali - `assert_neq`: verifica che i due valori passati non siano uguali

Altri tipi di assert che potremmo volere, come un `assert_contains` per verificare che una collezione contenga un dato elemento, sono disponibili come crate esterni alla standard library.

Quando un `assert` fallisce, il test fallisce. Se nessun `assert` all'interno di un test fallisce, il test è passato.

Queste assert impongono il vero funzinamento del test, quindi cerchiamo di non usare solo la `assert!` base ma anche le varianti `assert_eq!` o `assert_ne!` anzichè fare `assert!(val1 == val2)`.

Come già detto, per lanciare tutti i test, basta eseguire `cargo test`. **Commento Personale**: su Exercism di base tutti i test sono marchiati per essere saltati, tramite la macro `#[ignore]`. Ricordate di levarli prima di lanciare i test.