



Modularità

Organizzare il codice sorgente

Crates

- Il processo di compilazione di un progetto Rust è basato sul concetto di **crate**
 - A livello sorgente, un crate è un'unità di compilazione (file sorgente)
 - La compilazione di un crate sorgente da origine ad un file oggetto che potrà essere collegato (linked) con altri file oggetto per diventare un **eseguibile** completo (dotato di un punto di ingresso esplicito, `main()` , e di tutte le sue dipendenze statiche) oppure diventare una **libreria**
- Un crate binario costituisce un'unità di versionamento e di caricamento in fase di esecuzione
 - Quando un crate fa riferimento a codice (funzioni, strutture, tratti, ...) esportato da un altro crate (di tipo libreria), quest'ultimo deve essere combinato con il primo per mettere a disposizione quanto richiesto
 - Questa operazione è detta collegamento e dipende da come la libreria è stata realizzata
- Rust supporta la creazione di librerie con diverse forme di collegamento
 - Statico, in cui l'eseguibile finale ha una copia completa di tutto il codice utilizzato
 - Dinamico, in cui il codice esterno viene caricato all'atto dell'avvio dell'eseguibile, a partire da un file di tipo `.dll` (in Windows) o `.so` (nei sistemi Unix-like)

Librerie

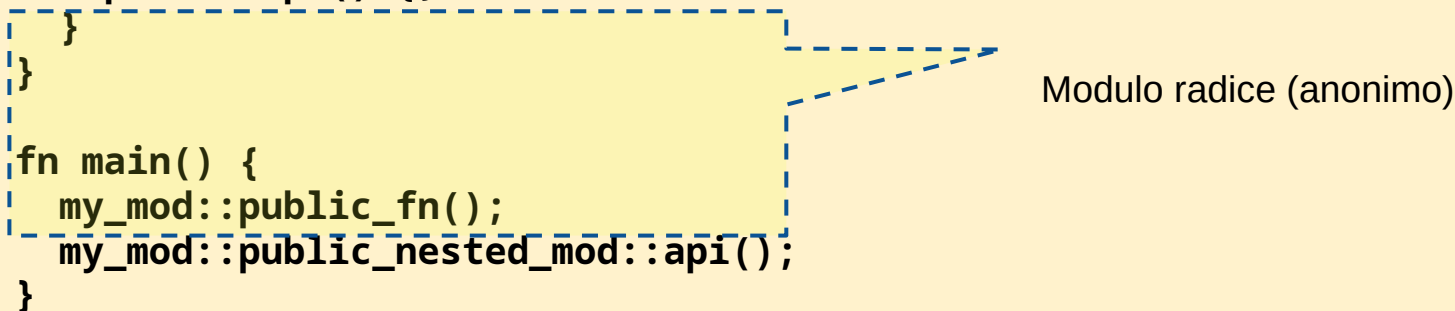
- La sezione **[lib]** del file Cargo.toml che descrive il progetto contiene dettagli relativi al tipo di libreria che si intende creare
 - La chiave **crate-type** può assumere i seguenti valori:
 - **rlib**: libreria statica Rust, valore di default, il file risultante può solo essere usato da altri eseguibili Rust che ne includeranno le parti importate direttamente nel proprio codice binario
 - **dylib**: libreria dinamica Rust; viene trasformata in un file .dll in windows, .so in linux, .dylib in macOS; può essere utilizzata solo da codice Rust, ma sarà caricata in fase di esecuzione
 - **cdylib**: libreria dinamica conforme alle specifiche C; viene trasformata in un file .dll in windows, .so in linux, .dylib in macOS; può essere usata da eseguibili scritti in altri linguaggi; se viene usata da un programma Rust deve essere inclusa sotto forma di funzioni esterne
 - **staticlib**: libreria statica conforme alle specifiche C; viene trasformata in un file .lib in windows e .a in linux e macOS; può essere usata da eseguibili scritti in altri linguaggi; se viene usata da un programma Rust deve essere inclusa sotto forma di funzioni esterne

Moduli e visibilità

- Il codice contenuto in un crate è costituito da un albero di moduli
 - Un modulo è un costrutto sintattico, introdotto dalla parola chiave `mod` e dal relativo nome che racchiude funzioni, tipi definiti dall'utente (`struct`, `enum`, `union`), tratti, implementazioni ed altri moduli
 - Tutto il codice che non è racchiuso in un blocco di tipo `mod nome_modulo { ... }`, fa parte del modulo radice
- I moduli formano una gerarchia ad albero, la cui radice è l'unità di compilazione corrente (crate)
 - Di base, tutti gli elementi definiti all'interno di un modulo sono considerati privati, e sono accessibili solo al codice presente nel modulo stesso o nei suoi sotto-moduli
 - Se un elemento definito all'interno di un modulo "m1" è preceduto dalla parola chiave `pub`, diventa pubblico e può essere utilizzato dal codice presente in un modulo "m2" che non sia un suo discendente, a condizione che tutti gli antenati del modulo "m1" siano a loro volta accessibili al modulo "m2" (perché pubblici o perché il modulo "m2" è contenuto al loro interno)

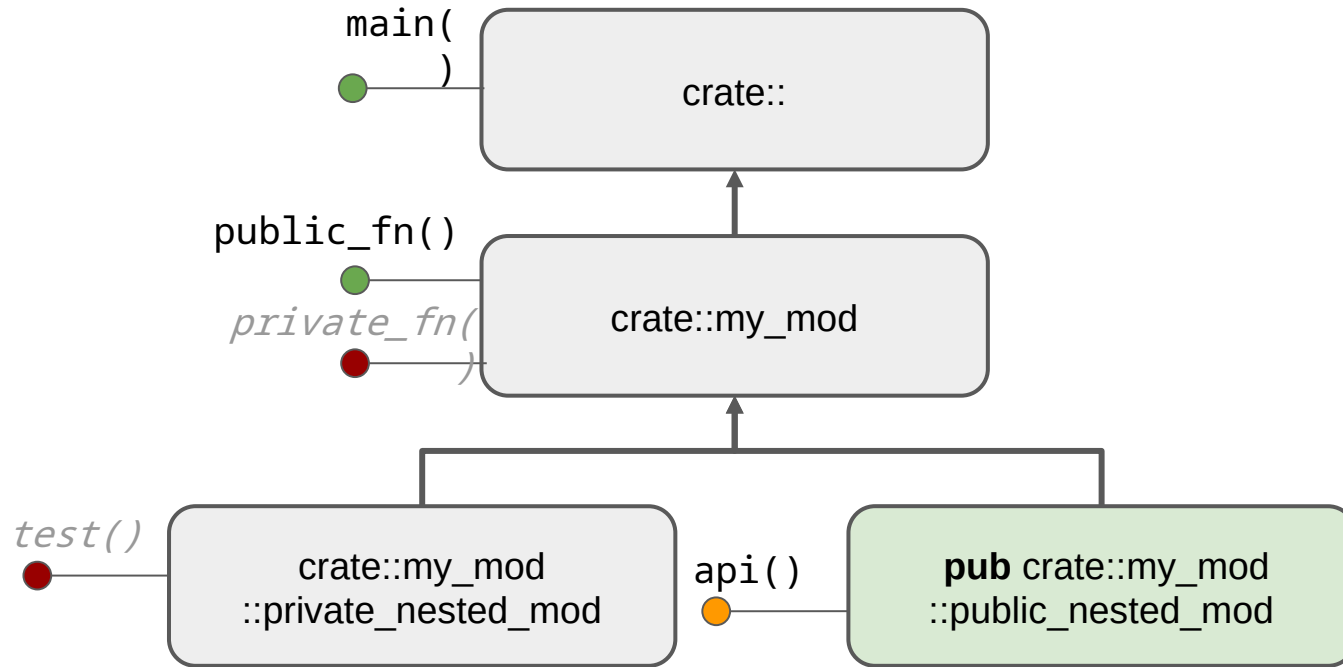
Moduli e visibilità

```
mod my_mod {  
    fn private_fn() {} // visibile solo nel modulo corrente  
    pub fn public_fn() {} // visibile nel contesto dell'unità di  
    compilazione:  
                                // - codice presente in questo stesso file  
                                // - codice presente in un file che dichiara use  
my_mod;  
    mod private_nested_mod { // sotto-modulo privato  
        fn test() {} // può accedere a my_mod::private_fn()  
    }  
    pub mod public_nested_mod { // accessibile a chi ha accesso a my_mod  
        pub fn api() {}  
    }  
}  
  
fn main() {  
    my_mod::public_fn();  
    my_mod::public_nested_mod::api();  
}
```



Modulo radice (anonimo)

Moduli e visibilità



Moduli e visibilità

- Per poter usare un simbolo (funzione, tipo, tratto, ...) definito in un modulo diverso da quello corrente occorre indicare al compilatore dove andare a reperirlo, facendo precedere il simbolo dal relativo cammino
 - Questo può essere assoluto o relativo
- Un cammino assoluto comincia con il nome del crate in cui si trova il simbolo seguito dalla sequenza gerarchica di moduli che occorre attraversare per giungere al simbolo
 - `let size = std::mem::size_of_val(&f);`
 - Si usa il separatore `::` per connettere i nomi dei (sotto-)moduli
 - Nel caso il simbolo appartenga al crate corrente, si può indicare come elemento iniziale la parola chiave `crate`
- Un cammino relativo comincia con le parole chiave `self`, `super` o direttamente con il nome del sottomodulo in cui il simbolo è definito
 - `let result = self::my_mod::public_nested_mod::api(some, arguments);`
- Per evitare di dover ripetere l'intero cammino per ogni occorrenza del simbolo, è possibile usare il costrutto `use cammino::del::modulo::*;`
 - Questo rende disponibile, nel file corrente, la definizione di tutti i simboli pubblici del modulo indicato
- Per poter accedere ad un simbolo occorre che tutti gli elementi del cammino siano accessibili a chi ne fa richiesta
 - Ovvero essere dichiarati come pubblici o contenere il modulo corrente

Moduli e file sorgente

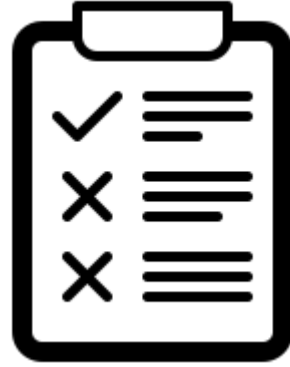
- Il codice relativo ad un sotto-modulo può essere inserito:
 - Nello stesso file sorgente del modulo genitore, con la notazione `mod nome_modulo { ... }`
 - In un file sorgente a sé, presente nella stessa cartella, chiamato `nome_modulo.rs`: in questo caso occorre che nel modulo genitore sia presente la dichiarazione del modulo, nel formato `mod nome_modulo;`
 - In una sotto-cartella chiamata `nome_modulo`, nel file chiamato `mod.rs`: questo può contenere direttamente la definizione degli elementi che lo compongono o fare riferimento ad ulteriori file sorgente, posti nella stessa sotto-cartella, contenenti ciascuno un sotto-modulo
- Eventuali crate esterni, non facenti parte della gerarchia di moduli definiti dal package corrente, vengono elencati nel file `cargo.toml` che descrive la struttura del package stesso
 - Nella sezione `[dependencies]` sono presenti, uno per riga, i nomi dei crate da importare con l'indicazione della relativa versione

Preludio

- Alcuni simboli particolarmente frequenti (come `Vec`, `String`, ...) non necessitano di essere importati in modo esplicito, pur appartenendo a crate distinti da quello corrente (sono infatti parte della libreria standard)
 - Questi sono elencati all'interno di un modulo chiamato `std::prelude` che viene importato automaticamente all'atto della compilazione di un file sorgente
 - In questo modo si evita di rendere verboso il codice sorgente
- Ogni versione di Rust viene con un proprio prelude
 - `std::prelude::v1`, `std::prelude::rust_2015`, `std::prelude::rust_2018`, `std::prelude::rust_2021`
 - Cargo provvede ad includere quello corretto in funzione del valore dell'attributo `package/edition` nel file `Cargo.toml`

Per saperne di più

- Rust Adventures: Rust projects management, understanding packages, Crates and modules
 - <https://levelup.gitconnected.com/rust-adventures-rust-projects-management-understanding-packages-crates-and-modules-b3bcde2eb1c>
- Module prelude
 - <https://doc.rust-lang.org/std/prelude/index.html>
- The edition guide
 - <https://doc.rust-lang.org/stable/edition-guide/>



Test

Controllare la correttezza del codice

Test

- Un test è un blocco di codice creato intenzionalmente per verificare se una certa porzione di codice funziona o meno
- Verificare funzionalmente i singoli componenti di un sistema è un modo efficace e pratico di creare e mantenere codice di alta qualità
 - Un test non può dimostrare l'assenza di errori, ma aiuta a generare confidenza nel sistema nel momento in cui questo viene messo in campo e aiuta a conservare la correttezza della base di codice, quando il progetto deve essere mantenuto nel tempo
 - E' molto complesso ri-organizzare su larga scale del codice in assenza di test di unità
- I benefici legati ad un uso intelligente e bilanciato dei test di unità nel software sono profondi
 - Nelle fasi di implementazione, test di unità ben scritti diventano una specifica informale dei componenti di un sistema
 - Nelle fasi di manutenzione, i test esistenti servono da freno alle regressioni nel codice, stimolando una correzione immediata

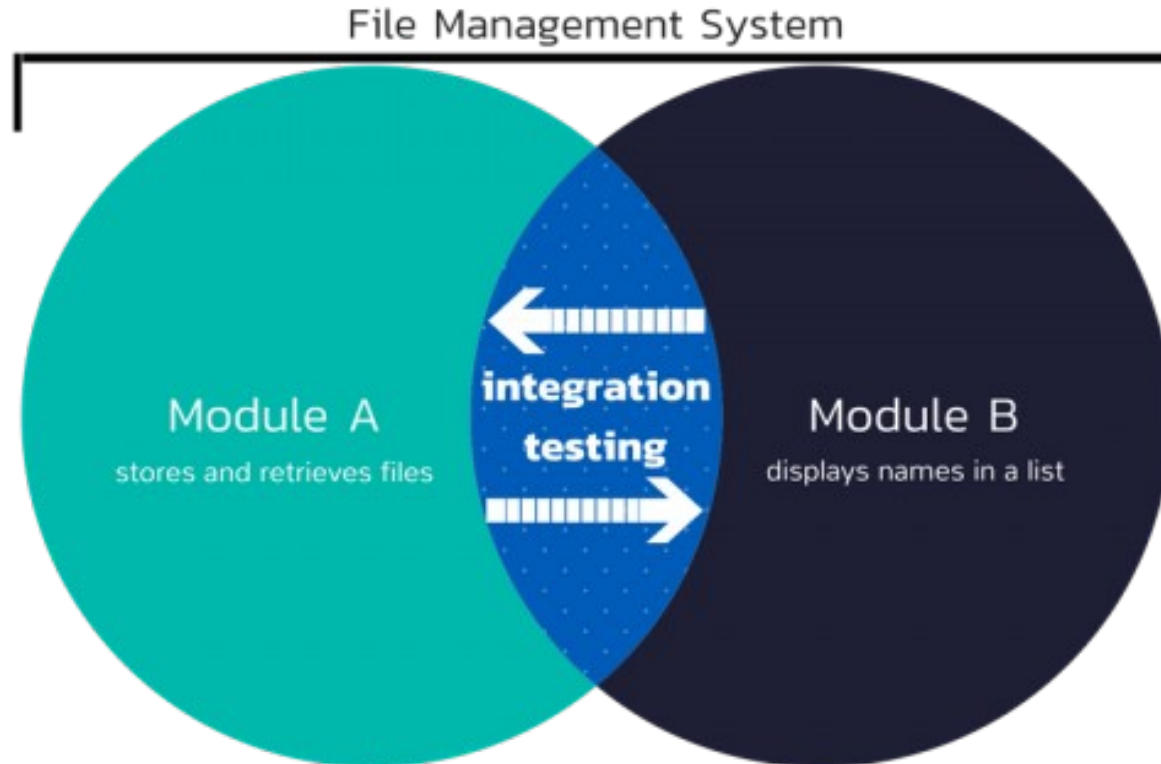
Test di unità

- Un **test di unità** valuta il comportamento di un **singolo componente** software (funzione, struttura dati, ...), indipendentemente dal resto del sistema
 - Sollecitandone sia il comportamento “tipico”, ovvero conforme all’uso per cui è stato progettato, sia quello agli estremi o oltre il suo previsto campo di utilizzo
- Tale sollecitazione viene esercitata tramite frammenti di codice che invocano il componente e verificano che esso risponda in modo atteso
 - Confrontando, ad esempio, i valori ritornati dalla funzione / contenuti nella struttura dati dopo la sollecitazione con valori/condizioni di errore attesi sulla base delle specifiche
- I test di unità sono normalmente scritti ed eseguiti dallo sviluppatore
 - Presuppongono la conoscenza dei meccanismi interni al componente, allo scopo di verificare come questi reagiscono in presenza di casi limite
 - Il loro numero dipende dalla complessità ciclomatica del modulo testato
 - La loro qualità è alla base del processo di refactoring cui un artefatto software tipicamente è soggetto nel corso del suo naturale ciclo di vita
- Il costo delle modifiche che vengono evidenziate da questo tipo di test è normalmente basso
 - I benefici che la scrittura di test ben fatti portano alla qualità del codice sono alti e compensano abbondantemente nel medio termine i maggiori costi dovuti alla scrittura dei test

Test di integrazione

- Un test di integrazione valuta il comportamento all'interfaccia di due moduli software
 - Si focalizza sul determinare la correttezza di tale interfaccia, evidenziando le incongruenze tra le parti che stanno interagendo
- Come nel caso dei test di unità, è costituito da frammenti di codice che sollecitano l'interazione tra due parti
 - A differenza dei test di unità, tuttavia, ignora volutamente la struttura interna dei moduli che stanno interagendo, assumendo il loro corretto comportamento - sul piano individuale - alle specifiche definite
 - Per questo motivo viene eseguito solo su oggetti che hanno correttamente superato il test di unità
- I costi delle modifiche che occorre introdurre se un test di integrazione fallisce sono generalmente più elevati
 - In quanto richiedono il ridisegno di una o entrambe le parti che interagiscono
- I test di integrazione sono co-progettati da sviluppatori e tester
 - Vengono eseguiti dai tester

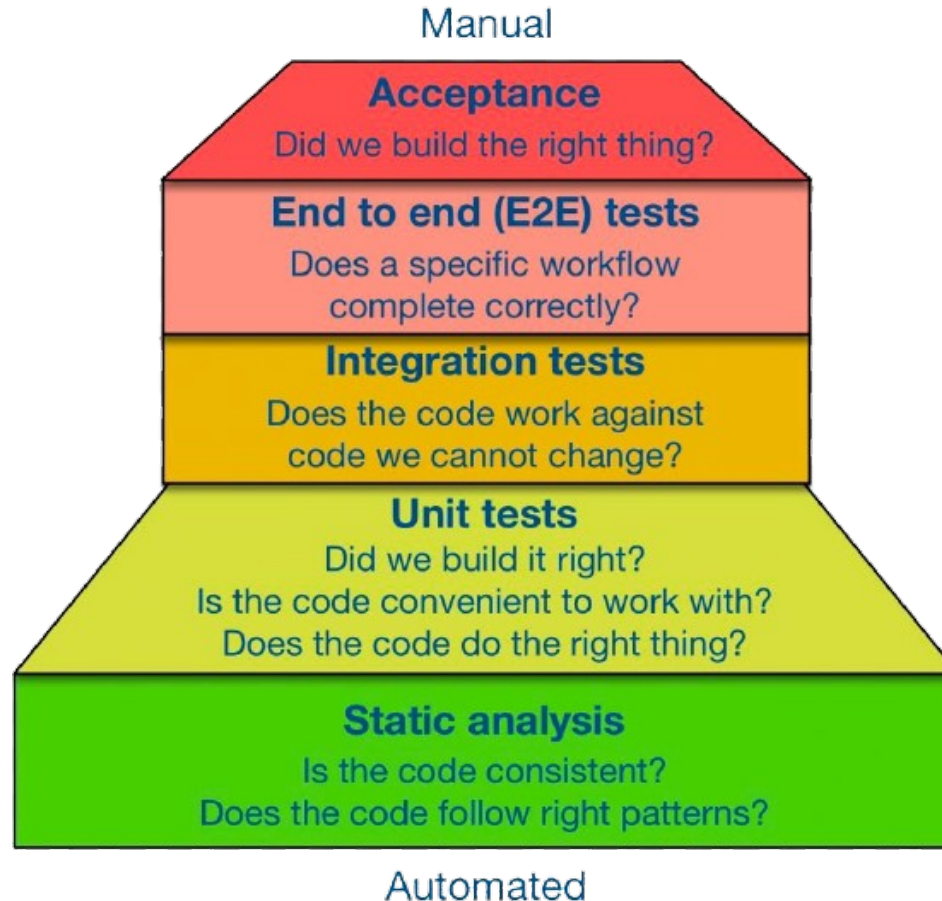
Test di integrazione



Test complessivi (*end-to-end*)

- Il **test di sistema** valuta il comportamento del prodotto finito
 - Sia nei suoi aspetti funzionali (cosa deve/non deve fare) che non funzionali (sicurezza, affidabilità, resilienza, prestazioni, scalabilità, ...)
- Basato su modelli di comportamento tratti da casi d'uso reali
 - Progettato ed eseguito da tester, allo scopo di non farsi influenzare dalle assunzioni fatte durante lo sviluppo
- Il test di accettazione verifica che il sistema sia conforme alle aspettative del committente/cliente
 - Eseguito dagli utenti finali

Le domande del test



Test in Rust

- I **test di unità** possono essere scritti nel modulo da testare o, meglio, in un suo sotto-modulo chiamato convenzionalmente 'tests'
 - Questo facilita l'ispezione del codice quando il numero di test cresce significativamente
- Il (sotto-)modulo contenente i test è preceduto dall'annotazione **`#[cfg(test)]`**
 - Questo informa il compilatore che il codice contenuto al suo interno deve essere incluso solo quando la compilazione avviene con il comando **`cargo test`**
- I **test di integrazione** sono invece contenuti in una cartella separata denominata **tests**, a lato della cartella **src** del progetto
 - Sono scritti come se i singoli frammenti di codice fossero i consumatori del crate che deve essere testato
 - I file sorgente all'interno della cartella di test importano i simboli pubblici oggetto di valutazione tramite istruzioni di tipo **`use...`**
 - Solo le librerie possono avere test di integrazione: per questo si creano programmi formati da un eseguibile contenente un main banale e da una libreria che contiene la logica applicativa

Sintassi dei test

```
// funzione da testare
fn sum(a: i32, b: i32) -> i32 {
    a + b
}

#[cfg(test)]
mod tests {
    fn sum_inputs_outputs() -> Vec<((i32, i32), i32)> {
        vec![((1, 1), 2), ((0, 0), 0), ((2, -2), 0)]
    }

    #[test]
    fn test_sums() {
        for (input, output) in sum_inputs_outputs() {
            assert_eq!(crate::sum(input.0, input.1), output);
        }
    }
}
```

Sintassi dei test

- La struttura generale di una funzione etichettata con `#[test]` è la seguente:
 - Preparazione dei valori su cui operare
 - Esecuzione del codice da testare
 - Verifica, mediante asserzioni, che i risultati ottenuti siano quelli attesi
- Per supportare l'ultimo passo, la libreria standard di Rust mette a disposizione varie macro procedurali
 - `assert!(boolean_condition)` - impone che la condizione sia vera
 - `assert_eq!(value1, value2)` - impone l'eguaglianza tra i due valori
 - `assert_ne!(value1, value2)` - impone la disuguaglianza tra i due valori
- Per verificare che un'espressione generi la condizione di panico, è possibile annotare il test anche con l'annotazione `#[should_panic(expected = "msg")]`
 - In questo caso, viene verificato che il messaggio generato da `panic!` contenga `"msg"`
- Se una funzione di test restituisce un valore di tipo `Result<T, Error>`, è possibile utilizzare la condizione di errore per indicare che il test è fallito

Eseguire i test

- Il comando **cargo test** ricompila il programma con la direttiva “test” ed esegue tutti i test contenuti nel package
 - Compresi quelli eventualmente presenti all’interno della documentazione
 - Viene prodotto un report dettagliato che indica ogni singolo fallimento, dove viene evidenziato il risultato atteso e quello realmente ottenuto seguito da un riassunto con il nome di tutte le funzioni di test che sono fallite
- E’ possibile lanciare solo alcuni test, specificando il nome della funzione (o una sua sottoparte) come ulteriore parametro del comando
 - **cargo test add_** esegue tutti i test i cui nomi contengono “add_”
- Se uno o più test è decorato con l’attributo **#[ignore]**, viene normalmente tralasciato dall’esecuzione
 - È possibile includere tali test con il comando **cargo test -- --ignored**

Per saperne di più

- Testing in Rust
 - <https://anismousse.medium.com/testing-in-rust-22f27136b433>
- Writing Automated Tests
 - <https://doc.rust-lang.org/book/ch11-00-testing.html>
- Rust Mock Shootout!
 - https://asomers.github.io/mock_shootout/
- Formatting, Linting, and Documenting with Rust
 - <https://blog.devgenius.io/formatting-linting-and-documenting-with-rust-eb7b189ade65>