# Esercitazione 4

21/23 Maggio 2025

# Esercizio 1

Obiettivi

- lifetime
- implementazione iteratori
- lambda function

Per questo esercizio fare riferimento al file **editor.rs**

Una struct **LineEditor** racchiude lo stato di un editor di testo, in cui il testo è memorizzato in un vettore di stringhe, una stringa ogni linea. L'approccio è comune in molte implementazioni di editor, in quanto racchiudere tutto il testo in una singola stringa richiederebbe continue allocazioni / deallocazioni ad ogni modifica, rendendolo estremamente inefficiente per file di grandi dimensioni.
Memorizzando il testo diviso in linee invece si minimizza la porzione di memoria che deve essere riallocata ogni volta, pur richiedendo più complessità del codice.

L'editor supporta una semplice interfaccia per implementare funzionalità come plugin:

- un metodo **all_lines()** restituisce un vettore con un riferimento a ciascuna linea di testo (non la copia!)
- un secondo metodo **replace()** permette di cambiare una linea dall'esterno

Di queste due funzioni è necessario effettuare l'implementazione.

Effettuata l'implementazione scrivere il plugin **FindReplace** (una struct), che dato un pattern permetta di cercarlo all'interno del testo e restituisca una lista di **Match**, con numero linea, offset inizio e fine e preview di ciascun Match.
Il plugin FindReplace inoltre permette di definire una funzione esterna custom, che può esaminare ciascun match e decidere se accettarlo ed infine può modificare il contenuto originale dell'editor.

Nel file editor.rs sono definite le interfacce delle struct da implementare, il file non compilerà in quanto mancano anche i lifetime necessari. Il suggerimento è commentare tutto, scommentare un pezzo per volta partendo dall'alto e sistemare i lifetime. Una volta fatto, pensare all'implementazione seguendo le indicazioni nei commenti.
**Per capire come utilizzare le funzioni definite guardare le funzioni di test inserite nel file**.

# Esercizio 2

Obiettivi:

- Iteratori
- Implementare un adattatore di un iteratore
- Uso di tratti per estendere oggetti
- Gestione errori

L'esercizio è diviso in due sotto problemi differenti, fare riferimento a **grep.rs**. Prima di iniziare a risolverlo leggere il pdf "Adapter Pattern…" fornito assieme all'esercizio.

Nella prima parte si chiede di realizzare un adattatore per iteratori su interi che permetta di iterare solo sui numeri pari. Seguire le indicazioni nel codice ed implementare le parti mancanti.
L'esercizio ha due step progressivi: prima si implementa per un solo tipo, i32, poi lo si generalizza per tutti gli interi.

Es:

```
let v = vec![1,2,3,4,5];
for item in v.into_iter().even() {
	println!("{}", item);
}
```

stamperà 2 e 4.

Nella seconda parte invece si deve realizzare un adattatore che esegue l'operazione grep (es: https://blog.keliweb.it/2021/11/linux-comando-grep-come-utilizzarlo/) in modo ricorsivo sui file contenuti in una directory.
Grep è un comando che permette di stampare tutte le occorrenze in un file di un pattern espresso con una regular expression.

Usando i crate **regex** e **walkdir**, che permette di iterare sul contenuto di una directory, realizzare un adattatore "grep" che permetta di eseguire codice come questo:

```
let walker = walkdir::WalkDir::new("/una_dir");
for m in walker.into_iter().grep(".*") {
	// gestisci match, possono esserci errori nell'aprire file
	// (es mancano permessi)
	// stampa errore o match
}
```

con output:

```
filexx:linea1:match
filexx:linea2:match
fileyy:linea1:match
filezz:errore
```

Anche per questo esercizio completare le funzioni inserite nel file, scrivere, completare e correggere le funzioni di test e implementare le classi necessarie.

Se l'implementazione risultasse complessa si consiglia di procedere a piccoli passi con multipli iterator adapter e scrivere test per valutare che il comportamento ottenuto sia quello atteso.

# Esercizio 3

Obiettivi:

- uso di collezioni
- gestione errori

Realizzare una struct Albero che gestisca le luci di un albero di Natale. L'albero è un albero proprio con una radice, no cicli, no link in avanti se non ai figli. Ogni nodo ha un nome, definito da una stringa arbitraria e un interruttore e una luce, che può essere accesa o spenta. Le operazioni supportate sono le seguenti:

```
impl Albero {
       // nota: aggiustare mutabilità dove necessario gestire errori in caso
       // di collisioni, valori mancanti

       // aggiungi un nodo figlio del nodo father
       pub fn add(&self, father: &str, node: &str) {}

       // togli un nodo e tutti gli eventuali rami collegati
       pub fn remove(&self, node: &str) {}

       // commuta l'interruttore del nodo (che può essere on off) e restituisci il
nuovo valore
       pub fn toggle(&self, node: &str) -> bool {}

       // restituisci se la luce è accesa e spenta
       pub fn peek(&self, node: &str) -> bool {}
}
```

Attenzione: la luce di un nodo è accesa se il suo interruttore è on e tutti gli interruttori dei nodi che lo collegano alla radice sono on, ne basta uno off perché non arrivi corrente.

Suggerimento: evitare di costruire in modo esplicito la struttura dell'albero con una struct del tipo Node {children: Vec<Node>, switch: bool } ma provare a memorizzare le relazioni fra i nodi e il loro stato in collezioni costruite in modo opportuno.