Esercitazione 1

Esercizio 1

Obiettivi:

- gestione stringhe e slice
- differenza tra “caratteri” (char) e “byte” (u8) in Rust
- lettura parametri da command line

Con il termine “slug” si intende una stringa convertita in formato leggibile, composta solo dai caratteri [a-z][0-9]-.
Nella stringa originale i caratteri non ammissibili vengono convertiti seguendo queste regole:

- tutti i caratteri accentati riconosciuti vengono convertiti nell’equivalente non accentato
- tutto viene convertito in minuscolo
- ogni altro carattere rimanente che non sia in [a-z][0-9] viene convertito in “-”
- due “-” consecutivi non sono ammessi, solo il primo viene tenuto
- un “-” finale non è ammesso a meno che non sia l’unico carattere nella stringa

L’obiettivo dell’esercizio è fare un funzione “slugify” che converta una stringa generica in uno slug.

PASSI PER LA SOLUZIONE

1. Creare con cargo un nuovo package chiamato **slugify** e, in main.rs, definire la funzione

   ```
   fn slugify(s: &str) -> String {}
   ```

2. Per convertire le lettere accentate definire una funzione che esegua la conversione:

   ```
   fn conv(c: char) -> char {}
   ```

   Conv restituisce il carattere c se è uno ammesso, la lettera non accentata corrispondente se viene trovata, o “-” negli altri casi

   All’interno usare questa tabella di conversione, dove il carattere nella posizione i (come carattere) in SUBS_I corrisponde al carattere nella posizione i in SUBS_O:

   ```
   const SUBS_I : &str =
   "àáâäæãåāăąçćčđďèéêëēėęěğǵḧîïíīįìıİłḿñńňôöòóœøõőṕŕřßśšşșťțûüùúūǔůűųẃẍÿýžźż";
   const SUBS_O: &str  =
   "aaaaaaaaaacccddeeeeeeeegghiiiiiiiilmnnnnoooooooooprrsssssttuuuuuuuuuwxyyzz
   z";
   ```

   ATTENZIONE: SUBS_I e SUBS_O essendo degli slice di stringa non possono essere indicizzati direttamente con [pos] (**perché?**), tutto quello che si può assumere è che il carattere corrispondente alla posizione i-esima di SUBS_I è nella stessa

posizione in SUBS_O.

3. Scrivere degli unit test per le funzioni create
   (riferimento: https://doc.rust-lang.org/rust-by-example/testing/unit_testing.html)

   a. creare una sezione in main.rs per ospitare i test
   ```
   #[cfg(test)]
   mod tests {
         use super::*;
   }
   ```
   b. i test sono funzioni così definite all'interno:
   ```
   #[test]
   fn my_first_test() {
         // valore = preparazione test
         assert_eq!(valore, valore_atteso)
   }
   ```
   c. definire almeno questi test:
      i. conversione lettera accentata
      ii. conversione lettera non accentata
      iii. conversione lettera non ammessa sconosciuta
      iv. conversione lettera accentata non compresa nella lista (es ῶ)
      v. stringa con più di una parola separata da spazio
      vi. stringa con caratteri accentati
      vii. stringa vuota
      viii. stringa con più spazi consecutivi
      ix. stringa con con più caratteri non validi consecutivi
      x. stringa con solo caratteri non validi
      xi. stringa con spazio alla fine
      xii. stringa con più caratteri non validi consecutivi alla fine
   d. i test possono venire lanciati con

      ```
      cargo test
      ```

4. Rendere la funzione invocabile dalla command line

   Nel main() leggere una sequenza di parole come argomento da command line, invocare la funzione e stampare il risultato.

   Esempio:
   ```
   cargo run -- "Questo che slug    sarà???"
   ```

   risultato: "slug: questo-che-slug-sara"

   (Attenzione: doppio "--" dopo run, serve per separare i parametri di cargo da quelli del comando; la stringa tra apici doppi viene letta come unico parametro e non come n parametri. Non usate copia incolla dal testo, perché l'editor usato per comporre il doc sostituisce gli apici doppi con altri caratteri)

Per il parsing degli argomenti di command line si suggerisce di inserire nel progetto la libreria clap: https://docs.rs/clap/latest/clap/

Clap è una libreria per leggere e validare i parametri da command line e ha due modalità di funzionamento, “derive” e “build”. La prima permette di definire i parametri da leggere mediante gli attributi di una struct, la seconda è più flessibile ma meno immediata da usare e consente di costruire i parametri da leggere in modo imperativo con una serie di istruzioni.
In questo progetto useremo la modalità “derive” (tutorial completo con esempi a questo indirizzo https://docs.rs/clap/latest/clap/_derive/_tutorial/chapter_0/index.html)

a. aggiungere la libreria cargo, editando il file cargo.toml od eseguendo:
   `cargo add clap --features derive`
   verrà aggiunto a cargo.toml:
   ```
   [dependencies]
   clap = { version = "4.5.3", features = ["derive"] }
   ```
b. in questo esempio non abbiamo opzioni con nome (es --verbose) e quindi leggiamo tutti i valori passati come una sola stringa, quindi definire una struct Args derivata da clap::Parser in questo modo:
   ```
   #[derive(Parser, Debug)]
   struct Args {
       // input string
       slug_in: String,
   }
   ```
c. la sintassi di clap è semplice: si definisce un attributo (in questo caso slug_in) per ogni parametro che si vuole leggere. Clap cercherà di fare la conversione automatica dei parametri passati al tipo indicato, mentre darà errore nel caso in cui non sia possibile
d. invocando nel main Args::parse() si effettua il parsing della command line e restituisce un oggetto di tipo Args con i parametri richiesti; clap aggiunge automaticamente l’opzione di --help e la gestione degli errori
e. provare ad invocare `cargo run -- --help` per verificare che clap funzioni in modo corretto
f. a questo punto l’invocazione `cargo run -- “Questo sarà uno slug?”` dovrebbe inserire in `args.slug_in` tutta la stringa “Questo sarà uno slug!” (notare i doppi apici intorno alla stringa, perché vogliamo che tutte le parole siano interpretate come un unico parametro)
g. per impratichirsi con clap aggiungere due parametri opzionali con nome --repeat=n e --verbose. Repeat è di tipo intero, verbose boolean.
   Usare l’annotazione #[arg()] come negli esempi al link indicato per impostare il comportamento del parser.
   Inoltre verificare, invocando il comando, che le conversioni, vengano lette in modo corretto, in particolare quella ad intero di --repeat
h. come potrei modificare il tipo di `slug_in` per leggere tutte le parole della stringa da convertire in un vettore di stringhe anziché in un’unica stringa?

Esercizio 2

Obiettivi

- Utilizzo di array
- parsing di stringhe
- mutabilità
- utilizzo di struct ed enum
- gestire i valori di ritorno e gli errori
- lettura/scrittura da file

ESERCIZI Propedeutici

Dopo avere creato un nuovo progetto rust provare questi tre brevi task (in funzioni dedicate) prima di passare alla soluzione dell'esercizio descritto in seguito.

1. Aprire, leggere e salvare un file: leggere un file "test.txt" con dentro del testo e salvare il testo ripetuto 10 volte nello stesso file
   Usare le funzioni read_to_string e write definite in std::fs (https://doc.rust-lang.org/std/fs/)
   a. Testare cosa capita quando il file o il path non esiste, gestire gli errori
   b. che differenza c'è tra read e read_to_string? Provare a leggere un file con delle lettere accentate con read e stampare in esadecimale l'array di byte letto con read, allineato con il testo sulla riga sopra.
      Esempio:
      c i   a o \n
      63 69 61 6f 0a
      Cosa notate se nel file è scritto "così\n" al posto di "ciao\n"?

2. Enum con "valore"
   A differenza del C le enum sono tipi che possono ospitare all'interno valore associato agli elementi della enum.
   Ogni elemento di una enum può essere di un tipo diverso e con `match` si può estrarre il contenuto della enum in una variabile.
   Definire quindi una `enum Error` con dentro due valori: `Simple(SystemTime)` e `Complex(SystemTime, String)` e fare una funzione `print_error(e: Error)` che stampi il tipo di errore e le informazioni contenute (senza usare {:?} debug, ma gestendo i valori della enum in modo opportuno)

3. Funzioni che possono restituire errori.
   In rust una funzione per segnalare un errore può usare la enum Result, che è definita in questo modo, analogamente a Option

```
pub enum Result<T, E> {
      Ok(T),
      Err(E)
}
```

T è il tipo del valore restituito come risultato, mentre E quello dell'errore, che può essere una enum qualsiasi
Implementare questa funzione:

```
pub mul(a: i32, b: i32) -> Result<u32, MulErr> {}
```

dove MulErr è

```
pub enum MulErr {Overflow, NegativeNumber};
```

mul quindi deve restituire:

- il risultato se possibile
- MyErr::NegativeNumber se uno tra a e b è negativo
- MyErr::Overflow se il risultato è più alto del valore massimo di u32

4. Uso di self, &self e &mut self
   Nei metodi di una struct si può utilizzare self per avere un riferimento all'oggetto su cui si sta operando: che differenza di comportamento c'è tra self, &self e &mut self?

   Ipotizziamo di avere la struttura

```
struct Node {
	name: String,
	size: u32,
	count: u32,
}
impl Node {
	pub fn new(name: String) -> Node  {
		Node {name, 0, 0}
	}
}
```

   Aggiungere due metodi size() e count() in modo che questo codice

```
let node = Node(String::new("nodo")).size(10).count(5);
```

   crei il nodo {"nodo", 10, 5}

   Quante struct di tipo Node costruisco in tutto? Ci sono penalità dal punto di vista dell'efficienza?

   Questo modo di creare l'oggetto, un pezzo per volta a partire dai default, viene definito "**builder pattern**". Ciò permette di ovviare al problema della mancanza di valori default e del polimorfismo nei metodi rust: nel caso di molti parametri opzionali posso usare il pattern per creare una versione "base" dell'oggetto con i valori di default e modificare solo gli attributi che hanno un valore diverso.

   Aggiungere un metodo to_string() che lo trasformi in stringa "name:node size:10 count:5". Come deve essere definito self in questo caso? Con o senza "&"?
   Infine aggiungere due metodi grow() e inc() che aumentano rispettivamente la size e il count di 1 senza creare un nuovo oggetto. Qui self come va definito?

TESTO ESERCIZIO

Un programma deve gestire la costruzione di uno schema di battaglia navale 20x20 salvato su file.
Il formato del file è il seguente (21 righe):

- **LINEA 1**: N1 N2 N3 N4, 4 interi separati da spazio che indicano il numero di navi rispettivamente di lunghezza 1, 2 , 3 e 4, che si possono ancora aggiungere alla board
- **LINEE 2..21**, 20 righe di 20 caratteri con " " (spazio) per le caselle vuote e "B" per quelle con navi

La costruzione della board avviene per passi, invocando il programma con dei parametri

- **cargo run -- new board.txt 4,3,2,1**
  questo crea una nuova board vuota nel file board.txt e può ospitare 4 navi da 1, 3 navi da 2, 2 navi da 3, e una da 4.
- **cargo run -- add board.txt 3V 10,10**
  legge la board in board.txt, aggiunge una nave da 3 caselle in verticale, partendo dalla casella (10,10) e andando giù 3 caselle, fino a (12,10). Possibili direzioni: H e V. Aggiunta la nave, salva il risultato in board, aggiornando anche le navi disponibili nella prima linea.
  Gli indici iniziano da 1 fino a 20

L'operazione di add deve essere "safe" e stampare errore, senza panic e senza modificare il file nel caso in cui non si possa aggiungere la nave. Casi in cui la nave non si può aggiungere:

- sono già state aggiunte tutte le navi possibili
- una nave si sovrappone o ha almeno un casella che "tocca" una nave esistente (anche di angolo)
- la nave va fuori dallo schema

(per la gestione della command line utilizzare sempre clap, come nell'esercizio precedente, notare gli spazi che separano file, dimensione boat e posizione start: provare a leggere tutti gli arg in un vettore di stringhe)

Per gestire la board utilizzare la seguente struttura ed implementare i metodi indicati **senza modificare la signature dei metodi presenti**:

```
const bsize: usize = 20;
pub struct Board {
    boats: [u8; 4],
    data: [[u8; bsize]; bsize],
}

pub enum Error {
    Overlap,
    OutOfBounds,
    BoatCount,
}
```

```
pub enum Boat {
    Vertical(usize),
    Horizontal(usize)
}

impl Board {
      /** crea una board vuota con una disponibilità di navi */
      pub fn new(boats: &[u8]) -> Board {}

      /* crea una board a partire da una stringa che rappresenta tutto
      il contenuto del file board.txt  */
      pub fn from(s: String)->Board {}

      /* aggiunge la nave alla board, restituendo la nuova board se
      possibile */
      /* bonus: provare a *non copiare* data quando si crea e restituisce
      una nuova board con la barca, come si può fare?  */
      pub fn add_boat(self, boat: Boat, pos: (usize, usize))
            -> Result<Board, Error> {}

      /* converte la board in una stringa salvabile su file */
      pub fn to_string(&self) -> String
}
```

Bonus

1. identificare e scrivere i test per struct Board
2. modificare Board in modo che add_boat alteri la struttura Board anziché crearne una nuova; per fare questo prestare attenzione al parametro self, come va modificato?
3. modificare il parse di clap in modo che gestisca i parametri in modo più espressivo e provare ad utilizzare la modalità builder anziché derive (https://docs.rs/clap/latest/clap/_tutorial/chapter_0/index.html):
   cargo run -- add --file=board.txt --boat=3V --start=10,10

Esercizio 3

Un sito che vi propone numerosi esercizi in rust, con test case per verificare se sono corretti è Exercism.
Per partecipare occorre:

- registrarsi gratuitamente
- scaricare l'esercizio con git
- leggere la descrizione dell'esercizio e completare il codice in modo che i test definiti funzionino senza dover modificare i test

Questa settimana vi segnaliamo (sarà fornita la soluzione fra 15 giorni assieme all'esercizio 1 e 2):

https://exercism.org/tracks/rust/exercises/clock