Esercitazione 3

Per questa esercitazione fare continuamente riferimento a questo link suggerito nelle slide *https://github.com/pretzelhammer/rust-blog/blob/master/posts/tour-of-rusts-standard-library-traits.md*

Contiene numerosi esempi e suggerimenti di come implementare i tratti richiesti in tutti e tre gli esercizi

Esercizio 1 - Tratto MySlug

Obiettivi:

- polimorfismo in Rust
- definire delle estensioni per i tipi di std
- implementazioni di default generiche dei tratti
- imporre dipendenze da altri tratti sui tipi generici

Modificando il codice della esercitazione precedente, slugify, definire un tratto **MySlug** per stringhe e slice. Il tratto deve definire i metodi che permettono di compilare il seguente codice e ottenere i risultati indicati:

```
let s1 = String::from("Hello String");
let s2 = "hello-slice";

println!("{}", s1.is_slug()); // false
println!("{}", s2.is_slug()); // true

let s3: String = s1.to_slug();
let s4: String  = s2.to_slug();

println!("s3:{} s4:{}", s3, s4); // stampa:  s3:hello-string s4:hello-slice
```

Passi soluzione

1. Aggiungere una funzione is_slug all'interno del modulo già creato che verifichi che uno slice sia uno slug (se una stringa è uno slug corrisponde allo slug generato dalla stringa stessa)
2. Come prima soluzione definire il tratto in modo esplicito per i tipi String e &str:
    a. definire il tratto MySlug
    b. fornire implementazioni separate sia per **String** che **&str** (nota: Rust tratta "&str" come un tipo a se stante, quindi si può definire un *impl xxx for &str*)
3. Come passo successivo provare a fare una singola implementazione generica, usando i constraint sui tipi a cui applicare il generic. Qui di seguito trovate come impostare la soluzione, sostituire "xxx" con un constraint e fornire l'implementazione.

```
impl<T> Slug for T
where
```

```
T : xxx {
      ….
}
```

Suggerimento per il constraint xxx: T deve essere un qualsiasi tipo che può essere derefrenziato come &str, quindi T deve implementare questo tratto per **str**: https://doc.rust-lang.org/std/convert/trait.AsRef.html)

Notare che con questa implementazione il tratto sarà automaticamente disponibile per tutti i tipi che permettono di ottenere un &str (anche tipi nuovi definiti dall'utente).

Esercizio 2 - ComplexNumber

Obiettivi

- panoramica estesa dei tratti in std
- organizzare un progetto in Rust con moduli e moduli di test

Implementare un tipo ComplexNumber che supporti gli operatori aritmetici di base (+, - ecc), che possa essere copiato, clonato, confrontato con se stesso e un numero reale, usato all'interno delle collezioni standard di Rust (vettori, hashmap, deque).

I tratti da implementare e le funzioni che deve realizzare sono definite dal file di test complex_numbers.rs fornito a parte. Come procedere:

1. creare un nuovo progetto rust (nome che useremo nell'esempio: cnumbers)
2. creare la directory tests nella radice del progetto se non esiste
3. copiare il file di tests in tests/
4. commentare tutti i test tranne il primo (altrimenti non compila)
5. realizzare in lib.rs[1] (dentro src) un modulo "solution" con all'interno la struct ComplexNumber
6. iniziare ad implementare ComplexNumber e i tratti richiesti per completare il primo test
7. scommentare il successivo test, seguire le indicazioni nei commenti e così via fino al completamento dei test (andare in ordine in quanto alcuni tratti da implementare sono dipendenti dai precedenti e potrebbe non compilare)

Per eseguire i test di un particolare modulo da command line (l'ide se il progetto è corretto dovrebbe permettere di eseguire direttamente i test con un pulsante "run" per ogni test):

```
cargo test --package [nome_package] --test [nome_modulo_di_test]
```

es per il package cnumbers (nome del crate) ed il modulo di test complex_numbers:

```
cargo test --package cnumbers --test complex_numbers
```

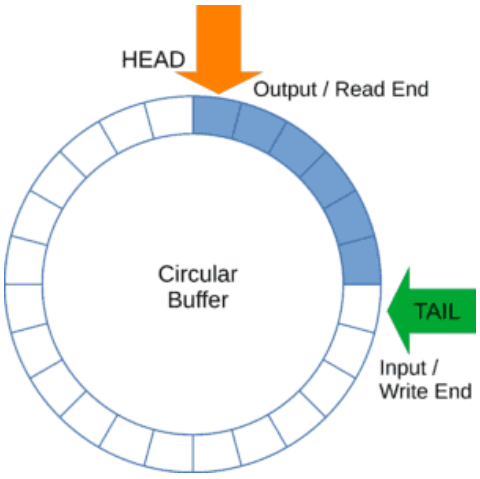
[1] lib.rs è il file di default in un cui creare i moduli di un crate per esportarli come libreria. Se definisco il modulo "**mymodule**" in lib.rs del crate "**mycrate**", da main.rs, dai test, da altri crate o dentro il crate stesso posso usare il modulo come **use mycrate::mymodule**;

Per una introduzione alla organizzazione dei crate vedere:
https://rust-classes.com/chapter_4_3

Esercizio 3 - Buffer Circolare

Obiettivi
- tipi complessi generici
- tratti Index e Deref
- analizzare il comportamento del borrow checker

Un buffer circolare è una struttura di dimensione fissa che permette di inserire elementi in coda ed estrarli dalla testa. Di solito viene implementato mediante un array. Idealmente può essere visto come un cerchio con due puntatori (indici): head e tail. Nella cella indicata da **tail** scrivo il valore successivo, in quella indicata da **head** leggo il valore corrente.



Quando è pieno non è più possibile inserire valori e una *write* restituisce errore, quando è vuoto invece la lettura restituisce None.
Fisicamente può essere realizzato mediante un **array o un Vector** preallocato che memorizza i valori: quando leggo restituisco il valore puntato da head e avanzo head di 1, così la casella prima occupata diventa libera, quando inserisco un valore lo scrivo nella cella puntata da tail e aggiungo 1 a tail.
Occorre stare attenti alle seguenti condizioni:
- quando head o tail vengono incrementati oltre oltre la lunghezza dell'array tornano a zero (è come se l'array fosse chiuso su se stesso in un cerchio)
- quando scrivo e tail viene a coincidere con head il buffer è pieno
- quando leggo e head viene a coincidere con tail buffer è vuoto

- per gli effetti di queste operazioni in certi momenti tail può essere minore di head e i valori nel buffer non sono contigui (una parte in coda e i successivi in cima al buffer)

Implementare un buffer circolare in grado di ospitare tipi generici che abbia la seguente interfaccia (da completare rendendola generica, è possibile aggiungere metodi se servono per la soluzione):

```
pub struct CircularBUffer { /*..*/ };

impl CircularBuffer {
	pub fn new(capacity: usize) -> Self {};
	pub fn write(&mut self, item: …) -> Result<..., …> {};
	pub fn read(&mut self) -> Option(...) {};
	pub fn clear(&mut self) {};
	pub dn size(&self) -> usize;
	// può essere usata quando il buffer è pieno per forzare una
	// scrittura riscrivendo l'elemento più vecchio
	pub fn overwrite(&mut self, item: …) {};
	// vedi sotto*
	pub fn make_contiguous(&mut self) {};
}
```

[*] Quando tail < head (tail ha raggiunto la fine ed è ritornato a zero) i valori nel buffer sono spezzati in due segmenti separati, una parte all'inizio, una parte alla fine dell'array, con lo spazio vuoto in mezzo. Non è quindi contiguo e **make_contiguous**() riorganizza il buffer, copiando in cima all'array tutti gli elementi mantenendo l'ordine di lettura, rendendolo così di nuovo contiguo.

Passi per risolvere l'esercizio:

1. rendere generica la struct, indicando quali sono i tratti richiesti per T (ad esempio se devo allocare un vettore di T senza sapere a priori i valori, T dovrà fornire un valore di default)
2. implementare i metodi generici
3. scrivere i test base:
    a. inserire elemento e controllare dimensione buffer
    b. inserire elemento, leggerlo e controllare che sia lo stesso
    c. ripetere per n elementi e leggerli
    d. controllare che head e tail ritornino correttamente a zero
    e. leggere da buffer vuoto
    f. scrivere su buffer pieno
    g. fare overwrite su buffer pieno (se non è pieno si deve comportare come write)
    h. rendere contiguo buffer non contiguo e controllare posizione di head e tail
4. Provarlo con il tipo complex realizzato nel punto precedente (opzionale, passare prima ai punti successivi)
5. Un buffer circolare generico può ospitare ogni tipo T, ma i tipi devono essere omogenei. Che escamotage posso usare in Rust per ospitare tipi eterogenei senza

modificare l'implementazione del Buffer? Quali sono le limitazioni? Come varia la occupazione di memoria?

6. Implementare i seguenti tratti aggiuntivi per il buffer circolare
    a. **Index** e **IndexMut**, in questo modo **buf[0]** permette di leggere e modificare l'elemento in testa e così via fino a tail (notare: l'indice non è l'offset reale nell'array, ma relativo ad head!)
    Nel caso di index out of bounds deve andare in panic!
    b. **Deref**: dereferenzia il buffer circolare come uno slice di T **&[T]**, con inizio e fine che coincidono con head e tail. **NB**: se il buffer non è contiguo deve fallire con panic!
    Perché il tratto Deref non può chiamare internamente make_contiguous() e non fallire? Attenzione ai vincoli di mutabilità
    c. **TryDeref**: si comporta come Deref, ma è più gentile: se non è contiguo restituisce un errore senza andare in panic!
    (*notare: implementando TryDeref ha poco senso implementare anche Deref, lo scopo di implementare entrambi è solo per vedere la differenza*)
    d. **DerefMut**: si comporta come Deref ma restituisce **&mut[T]**. Qui è possibile fare una implementazione che eviti di fallire?

Osservazioni generali sulla importanza del borrow checker di Rust: i tratti come Deref sono molto "pericolosi" potenzialmente, perché se qualcuno legge/inserisce elementi nel buffer mentre ci si accede come slice, i dati potrebbero essere non più contigui o tail e head non allineati con lo slice.
Ma questo è possibile con Rust? Provare con degli esempi che cercano di modificare il buffer mentre si ha un riferimento ad esso come slice e osservare come il compilatore agisce per prevenire i potenziali problemi di questo tipo.