ITERATORE

Un iteratore è una struttura dati dotata di stato, in grado di generare una sequenza di valori

- Estrae i valori uno alla volta
- **Stato**: perché deve capire a che punto è arrivato

offre il metodo **Next**:

- Ritorna una **Option<E>**
  - *Null → non c'è nulla*
  - *Altrimenti valore*

Nota:

- Posso attaccare un iteratore ad un altro
- Posso avere un iteratore che filtra
- Posso avere un iteratore che trasforma

Uso degli iteratori:

Offre metodi di compatezza, leggibilità, manutenibilità ed efficienza.

```
let v1 = vec![1,2,3,4,5,6];
let mut v2 = Vec::<String>::new();

for i in 0..v1.len() {
    if v1[i] %2 != 0 { continue; }
    v2.push(format!("a{}",v1[i]))
}
println!("{:?}", v2); // ["a2","a4","a6"]
```

```
let v1 = vec![1,2,3,4,5,6];
let mut v2: Vec<String>;

v2 = v1.iter()
    .filter(|val|{ *val %2 == 0})
    .map(|val| format!("a{}",val))
    .collect();

println!("{:?}", v2); // ["a2","a4","a6"]
```

V1.**iter** → prepara per dare i numeri
.filter → quando avrai i valori, passami solo i pari
.map → usa questa lambda per trasformare in stringa
.collect → parti e raccogli il tutto (dunque chiede a map che chiede a filter che chiede a iter che si sveglia)

Nota: iteratore non fa niente finchè non ci metto qualcosa che consuma

Caratteristiche iteratori:

- Offre un modo uniforme per accedere agli elementi in maniera indipendente da come siano generati o dove siano prelevati
  - Posso avere un codice generico
- Sono pigri → fanno qualcosa solo se chiediamo l'elemento successivo
- Possono operare in parallelo
- Da un iteratore è possibile derivare un altro iteratore
  - Codice molto flessibile

Nota: iteratori sono presenti anche in c++, esempi sulle slide.

Gli iteratori implementano il **tratto iterator**

std::iter::Iterator

```
trait Iterator{
  type Item;
  fn next(&mut self) -> Option<Self::Item>;
  …// molti altri metodi con implementazione di default
}
```

Item: tipo del dato ritornato

Next: da il successivo, se c'è

Nota: se un tipo **permette** di essere esplorato tirandone un elemento alla volta → implementa il tratto iterator

IntoIterator: un tipo che **vuole permettere** di iterare sugli oggetti senza dare il reference ma dando proprio i dai che gli oggetti contiene (quando iteratore eseguito → struttura sbriciolata) deve implementare **IntoIterator**

implementando il tratto **std::iter::IntoIterator**

```
trait IntoIterator where Self::IntoIter: Iterator<Item=Self::Item> {
  type Item;
  type IntoIter: Iterator;
  fn into_iter(self) -> Self::IntoIter;
}
```

esempio di implementazione

```
struct MyRange<const FROM: isize, const TO: isize> {}

impl<const FROM: isize, const TO: isize> IntoIterator for MyRange<FROM, TO> {
    type Item = isize;
    type IntoIter = MyRangeIterator<FROM, TO>;

    fn into_iter(self) -> Self::IntoIter {
        MyRangeIterator::<FROM,TO>::new()
    }
}

struct MyRangeIterator<const FROM: isize, const TO: isize> { val: isize }

impl<const FROM:isize, const TO:isize> MyRangeIterator<FROM, TO> {
    fn new() -> Self {
        MyRangeIterator{ val: FROM }
    }
}
```

```
impl<const FROM:isize, const TO:isize> Iterator for MyRangeIterator<FROM,TO> {
    type Item = isize;
    fn next(&mut self) -> Option<Self::Item> {
        if FROM < TO {
            if self.val < TO {
                let ret = self.val;
                self.val += 1;
                Some(ret)
            } else { None }
        } else {
            if self.val > TO {
                let ret = self.val;
                self.val -= 1;
                Some(ret)
            } else { None }
```

iteratori e possesso

tre metodi:

- **iter()** → prende in prestito la struttura → struttura non modificabile
- **iter_mut()** → prende in prestito la struttura e può modificarla
- **into_inter()** → prende possesso del contenitore e restituisce ogni di tipo Item estraendoli dal contenitore
  - Sbriciola la struttura, al termina dell'esecuzione dell'into_iter, la struttura originale sarà distrutta

Questi 3 metodi sono aiutati dal **tratto intoIterator** che ha 3 implementazioni:

- For x in v → into_iter
- For x in &v → iter
- For x in &mut v → iter_mut()

--codice lez24 pt2

```
let mut v = vec![String::from("a"), String::from("b"), String::from("c")];

for s in &v {
    println!("{}", s);     // s: &String
}

for s in &mut v {
    s.push_str("1") ;      // s: &mut String - Modifico il contenuto del
vettore
}

for s in v {
        println!("{}", s); // s: String - invalido il contenuto del vettore
}
```

```
let v = vec![String::new("a"), String::new("b"), String::new("c")];

let it = v.iter_mut();

for (s in it) { // it.into_iter() -> it
   // qui s ha tipo &mut String e opera sui valori contenuti in v
}
```

into_iter restituisce l'iteratore stesso

ADATTATORI

Metodi che consumano un iteratore e ne derivano uno differente in grado di offrire funzionalità ulteriori.

- possono essere combinati in catene al termine delle quali inserire un consumatore
- Sono pigri → non invocano il metodo next() tranne se avviene una richiesta da un consumatore

- `map<B, F>(self, f: F) -> Map<Self, F>`
  - Esegue la chiusura ricevuta come argomento su ogni elemento dell'iteratore ritornato
- `filter<P>(self, predicate: P) -> Filter<Self, P>`
  - Ritorna un iteratore che restituisce solo gli elementi per i quali l'esecuzione della chiusura ricevuta come argomento ritorna true
- `filter_map<B, F>(self, f: F) -> FilterMap<Self, F>`
  - Concatena in maniera concisa filter e map, l'iteratore risultante conterrà solo elementi per i quali la chiusura ritorna Some(B)
- `flatten(self) -> Flatten<Self>`
  - Ritorna un iteratore dal quale sono state rimosse le strutture annidate
  - `vec![vec![1,2,3,4],vec![5,6]].into_iter().flatten().collect::<Vec<u8>>()==&[1,2,3,4,5,6]`
- `flat_map<U, F>(self, f: F) -> FlatMap<Self, U, F>`
  - Concatena in maniera concisa map e flatten, esegue la chiusura ricevuta e rimuove le strutture annidate
- `take(self, n: usize) -> Take<Self>`
  - Ritorna un iteratore che contiene al più i primi n elementi dell'iteratore su cui viene eseguito (meno, se l'iteratore originale non contiene abbastanza elementi)
- `take_while<P>(self, predicate: P) -> TakeWhile<Self, P>`
  - Esegue la funzione ricevuta su tutti gli elementi dell'iteratore originale, conserva tutti gli elementi fino a quando la funzione ritorna true; dal momento in cui diventa false, scarta tutti i valori rimanenti
- `skip(self, n: usize) -> Skip<Self>`
  - Ritorna un iteratore che esclude i primi n elementi dell'iteratore su cui viene eseguito, se si raggiunge la fine ritorna un iteratore vuoto
- `skip_while<P>(self, predicate: P) -> SkipWhile<Self, P>`
  - Esegue la funzione ricevuta su tutti gli elementi dell'iteratore originale, esclude tutti gli elementi fino a quando la funzione ritorna false, dal momento in cui diventa true conserva tutti i valori rimanenti
- `peekable(self) -> Peekable<Self>`
  - Ritorna un iteratore sul quale è possibile chiamare i metodi `peek()` e `peek_mut()` per accedere al valore successivo senza consumarlo.

!

- `fuse(self) -> Fuse<Self>`
  - Ritorna un iteratore che termina dopo il primo None
- `rev(self) -> Rev<Self>`
  - Ritorna un iteratore con la direzione invertita
- `inspect<F>(self, f: F) -> Inspect<Self, F>`
  - Ogni volta che riceve una richiesta, preleva un elemento dall'iteratore a monte e la passa sia alla funzione, che ha possibilità di ispezionarlo, che al consumatore a valle
- `chain<U>(self, other: U) -> Chain<Self, <U as IntoIterator>::IntoIter>`
  - Prende come argomento un iteratore e lo concatena all'originale, ritorna uno nuovo iteratore
- `enumerate(self) -> Enumerate<Self>`
  - Ritorna un iteratore che restituisce una tupla formata dall'indice dell'iterazione e dal valore (i,val)
- `zip<U>(self, other: U) -> Zip<Self, <U as IntoIterator>::IntoIter>`
  - Combina due iteratori per ritornare un nuovo iteratore che ha come elementi le coppie composte dai valori dei primi due iteratori
- `by_ref(&mut self) -> &mut Self`
  - Prende in prestito un iteratore senza consumarlo, lasciando intatto il possesso dell'originale
- `copied<'a, T>(self) -> Copied<Self>`
  - Ritorna un nuovo iteratore, tutti gli elementi dell'iteratore originale vengono **copiati**
- `cloned<'a, T>(self) -> Cloned<Self>`
  - Ritorna un nuovo iteratore, tutti gli elementi dell'iteratore originale vengono **clonati**
- `cycle(self) -> Cycle<Self>`
  - Raggiunta la fine di un iteratore riparte dall'inizio, ciclando all'infinito

CONSUMATORI

- `for_each<F>(self, f: F)`
  - Esegue la chiusura ricevuta su tutti gli elementi dell'iteratore
- `try_for_each<F, R>(&mut self, f: F) -> R`
  - Esegue una chiusura che può fallire su tutti gli elementi dell'iteratore, si ferma dopo il primo fallimento
- `collect<B>(self) -> B`
  - Trasforma un iteratore in una collezione
- `nth(&mut self, n: usize) -> Option<Self::Item>`
  - Ritorna l'ennesimo elemento dell'iteratore
- `all<F>(&mut self, f: F) -> bool`
  - Verifica che la chiusura ricevuta restituisca true per tutti gli elementi restituiti dall'iteratore
- `any<F>(&mut self, f: F) -> bool`
  - Verifica che la chiusura ricevuta restituisca true per almeno un elemento restituito dall'iteratore
- `find<P>(&mut self, predicate: P) -> Option<Self::Item>`
  - Cerca un elemento sulla base della chiusura ricevuta come argomento e lo ritorna
- `count(self) -> usize`
  - Ritorna il numero di elementi dell'iteratore
- `sum<S>(self) -> S`
  - Somma tutti gli elementi di un iteratore e ritorna il valore ottenuto
- `product<P>(self) -> P`
  - Moltiplica tutti gli elementi di un iteratore e ritorna il valore ottenuto
- `max(self) -> Option<Self::Item>`
  - Ritorna il massimo tra gli elementi dell'iteratore, se trova due massimi equivalenti torna l'ultimo, se l'iteratore è vuotc viene ritornato None
- `max_by<F>(self, compare: F) -> Option<Self::Item>`
  - Ritorna il massimo tra gli elementi dell'iteratore sulla base della chiusura di confronto ricevuta come argomento
- `max_by_key<B, F>(self, f: F) -> Option<Self::Item>`
  - Esegue la chiusura ricevuta come argomento su tutti gli elementi e ritorna quello che produce il risultato massimo
- `min(self) -> Option<Self::Item>`
  - Ritorna il minimo tra gli elementi dell'iteratore, se trova due minimi equivalenti torna l'ultimo, se l'iteratore è vuoto viene ritornato None
- `min_by<F>(self, compare: F) -> Option<Self::Item>`
  - Ritorna il minimo tra gli elementi dell'iteratore sulla base della chiusura di confronto ricevuta come argomento
- `min_by_key<B, F>(self, f: F) -> Option<Self::Item>`
  - Esegue la chiusura ricevuta come argomento su tutti gli elementi e ritorna quello che produce il risultato minimo
- `position<P>(&mut self, predicate: P) -> Option<usize>`
  - Cerca un elemento sulla base della chiusura ricevuta come argomento e ritorna la posizione
- `rposition<P>(&mut self, predicate: P) -> Option<usize>`
  - Cerca un elemento sulla base della chiusura ricevuta come argomento, partendo da destra e ritornando la posizione
- `fold<B, F>(self, init: B, f: F) -> B`
  - Esegue la chiusura ricevuta accumulando i risultati sul primo argomento ricevuto
- `try_fold<B, F, R>(&mut self, init: B, f: F) -> R`
  - Esegue la chiusura ricevuta fino a quando ritorna con successo, accumulando i risultati sul primo argomento ricevuto
- `last(self) -> Option<Self::Item>`
  - Ritorna l'ultimo elemento dell'iteratore
- `find_map<B, F>(&mut self, f: F) -> Option<B>`
  - Esegue la chiusura ricevuta su tutti gli elementi e ritorna il primo risultato valido
- `partition<B, F>(self, f: F) -> (B, B)`
  - Consuma un iteratore e ritorna due collezioni sulla base del predicato ricevuto
- `reduce<F>(self, f: F) -> Option<Self::Item>`
  - Riduce l'iteratore ad un singolo elemento eseguendo la funzione ricevuta
- `cmp<I>(self, other: I) -> Ordering`
  - Confronta gli elementi di due iteratori
- `eq<I>(self, other: I) -> bool`
  - Verifica se gli elementi di due iteratori sono uguali
- `ne<I>(self, other: I) -> bool`
  - Verifica se gli elementi di due iteratori sono diversi
- `lt<I>(self, other: I) -> bool`
  - Verifica se gli elementi di un iteratore sono minori rispetto a quelli di un secondo iteratore
- `le<I>(self, other: I) -> bool`
  - Verifica se gli elementi di un iteratore sono minori o uguali rispetto a quelli di un secondo iteratore
- `gt<I>(self, other: I) -> bool`
  - Verifica se gli elementi di un iteratore sono maggiori rispetto a quelli di un secondo iteratore
- `ge<I>(self, other: I) -> bool`
  - Verifica se gli elementi di un iteratore sono maggiori o uguali rispetto a quelli di un secondo iteratore
- `...`