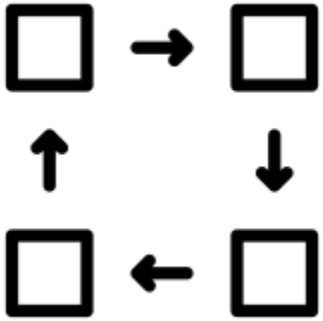


Iteratori

Accedere in modo indiretto ad una sequenza di valori

Iteratori

- Un iteratore è una struttura dati dotata di stato, in grado di generare una sequenza di valori
  - I valori possono essere estratti da un contenitore, di cui l'iteratore detiene un riferimento, o generati programmaticamente, come nel caso di un intervallo di valori
- Un iteratore offre tipicamente un modo per verificare se sono presenti ulteriori valori da generare (*hasNext() → bool*) ed un altro per accedere al valore successivo (*next() → E*)
  - In Rust le due operazioni sono combinate (*next() → Option<E>*)
- Un iteratore può offrire ulteriori metodi che consentono di derivare un nuovo iteratore
  - Che trasforma la sequenza di valori in un'altra sequenza (accorpando, eliminando, trasformando, …, i singoli elementi originali
- Pressoché tutti i linguaggi "moderni" offrono il concetto di iteratore come parte della propria libreria standard
  - Essi permettono di accedere ai valori contenuti all'interno di collezioni come liste, insiemi, mappe, scorrere i caratteri presenti all'interno di una stringa o leggere il contenuto di un file di testo estraendo una riga alla volta

Uso degli iteratori

- L'uso degli iteratori si sovrappone concettualmente a quello dei cicli for/while
  - Ma offre svariati vantaggi in termini di compattezza del codice, leggibilità, manutenibilità ed efficienza, nascondendo i dettagli necessari a generare / accedere ai singoli elementi
  - I vantaggi sono maggiormente evidenti se l'operazione svolta all'interno del ciclo è complessa e richiede, ad esempio, di scartare alcuni valori e di trasformare i restanti

```
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"]
```

Caratteristiche degli iteratori

- Un iteratore offre un modo uniforme di accedere agli elementi, indipendentemente da come essi siano generati o da dove siano prelevati
  - Permettendo al codice che utilizza tali valori di ignorare la fonte ed essere più generico
- Gli iteratori operano in modalità pigra (**lazy**)
  - Solo a fronte della richiesta di un valore successivo, si occupano di generarlo / prelevarlo
- Gli iteratori possono abilitare l'elaborazione parallela dei dati ospitati in una collezione
  - Permettendo di sfruttare la presenza di più core nell'elaboratore
- La possibilità di derivare un iteratore da un altro iteratore aumenta la flessibilità del codice
  - E' facile inserire all'interno di una catena di elaborazione passi ulteriori che iniettano nuovi valori, ne eliminano altri, combinano più valori tra loro, modificano l'ordine di visita, …, senza dover intervenire sul codice che utilizza i valori generati

Iteratori in C++

- Il C++ considera gli iteratori come una **versione generalizzata dei puntatori**
  - L'astrazione di partenza è legata a come vengono visitati gli array in C/C++, sfruttando l'aritmetica dei puntatori

```
class C;
C array[10];

C *iter = array; //puntatore al primo elemento
C *end = array + 10;//puntatore oltre l'ultimo

for( ; iter != end; iter++){
        C& elem = *iter;
        //… opero su elem
}
```

```
class C;
vector<C> v(10);

vector<C>::iterator iter = v.begin();
vector<C>::iterator end = v.end();

for( ; iter != end; iter++) {
        C& elem = *iter;
        //… opero su elem
}
```

Iteratori in C++

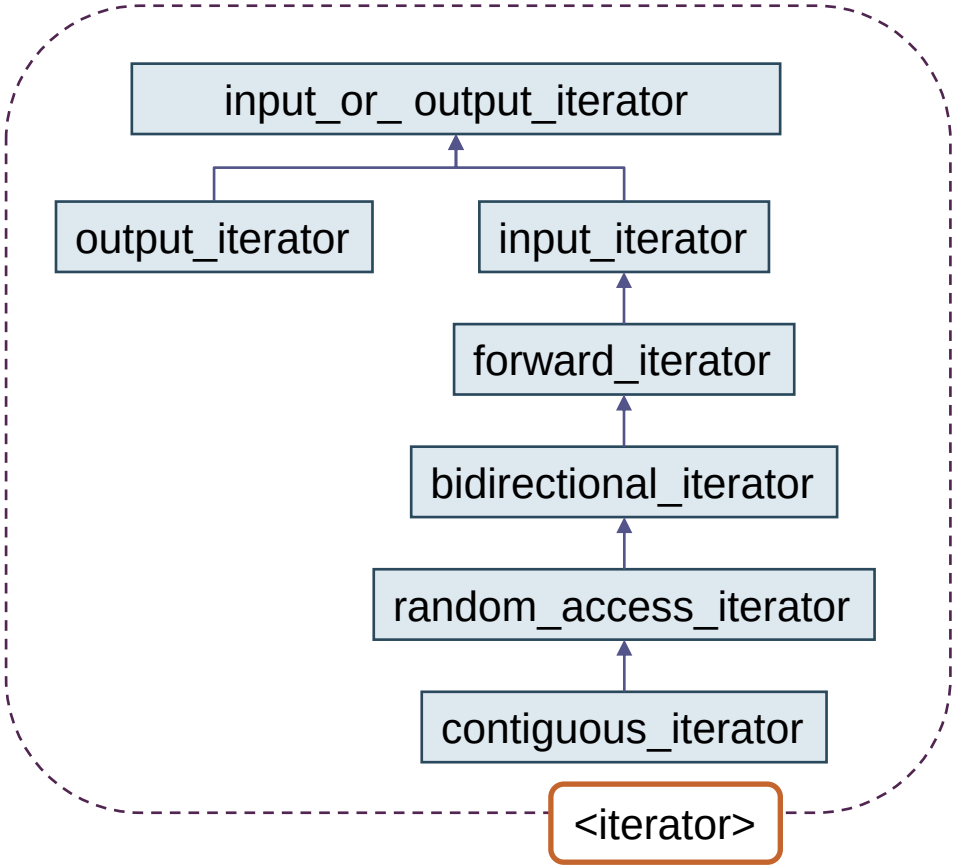
- In C++, un iteratore è definito in modo “implicito”, senza assegnargli un tipo specifico
  - Un iteratore, nel caso più semplice, è un oggetto che può essere dereferenziato (per accedere al valore corrente), incrementato (per passare al successivo) e confrontato con un’istanza della stessa classe (per sapere se si è raggiunto il fondo)
- Di conseguenza, ogni contenitore mette a disposizione il proprio tipo specifico di iteratore
  - Operatori di tipo diverso non sono tra loro interoperabili
  - Tecnica del “duck typing”: *if it walks like a duck and it quacks like a duck, then it must be a duck*
- E' possibile creare iteratori su misura per una specifica struttura dati avendo cura di definire una classe che implementi i metodi base necessari all'astrazione
  - `operator*()`, `operator->()`, `operator==(…)`, `operator!(…)`, `operator++()`

C++

```
template<long FROM, long TO>
class MyRange { // rappresenta l'intervallo di valori tra FROM e TO (escluso)
public:
    class iterator {
        using iterator_category = std::input_iterator_tag;
        using value_type = long;
        using difference_type = long;
        using pointer = const long*;
        using reference = long;

        long num = FROM; // STATO DELL'ITERATORE
    public:
        explicit iterator(long _num) : num(_num) {}
        iterator& operator++() {num = TO >= FROM ? num + 1: num - 1; return *this;}
        iterator operator++(int) {iterator retval = *this; ++(*this);
                                  return retval;}
        bool operator==(iterator other) const {return num == other.num;}
        bool operator!=(iterator other) const {return !(*this == other);}
        reference operator*() const {return num;}
    };
    iterator begin() {return iterator(FROM);}
    iterator end() {return iterator(TO >= FROM? TO+1 : TO-1);}
};
```

Iteratori in C++20

- A partire dalla versione C++20, è stata introdotta una tassonomia di concept, volta a descrivere requisiti via via più stringenti su cosa possa essere considerato un iteratore
  - Il caso più generico richiede l'incrementabilità (`it++`) e la dereferenziabilità (`*it`)
  - **`input_iterator`** si distingue da **`output_iterator`** perché il valore dereferenziato (`T v= *it;`) può essere letto piuttosto che assegnato (`*it = v;`)
  - **`forward_iterator`** aggiunge la confrontabilità tra iteratori della stessa classe (`it1==it2`)
  - **`bidirectional_iterator`** aggiunge la decrementabilità (`it--`)
  - **`random_access_iterator`** permette di far avanzare e retrocedere (in un tempo costante) la poszione dell'iteratore di più unità (`it+=n;`)



Iteratori nella libreria standard C++

- La libreria standard offre molteplici classi contenitore (**`std::array<T>`**, **`std::list<T>`**, **`std::vector<T>`**, …) ciascuna caratterizzata da una diversa strategia di implementazione
  - Ed in grado di offrire differenti compromessi / prestazioni nelle funzionalità di accesso e modifica dei dati contenuti al loro interno
- Nonostante l'esistenza di profonde differenze implementative, tali classi sono accomunate da un uso coerente dei relativi iteratori, mediante i quali è possibile scrivere funzionalità facilmente portabili e interscambiabili a livello di codice sorgente
  - Un'intera sezione della libreria standard, descritta nel file intestazione <algorithms>, offre funzionalità indipendenti dal tipo di contenitore proprio grazie all'uso degli iteratori
  - Al suo interno sono raccolti algoritmi per ricerca e ricerca binaria, trasformazione dei dati, partizionamento, ordinamento, merge, operazioni insiemistiche, operazioni su heap, comparazioni lessicografiche…

Iteratori in Rust

- In Rust, un **iteratore** è una qualsiasi struttura dati che implementa il tratto **`std::iter::Iterator`**

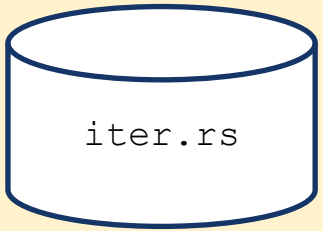
```
trait Iterator{
  type Item;
  fn next(&mut self) -> Option<Self::Item>;
  …// molti altri metodi con implementazione di default
}
```

- **`Item`**: tipo del valore che l'iteratore produce
- **`next()`**: ritorna un `Option`.

Iteratori e possesso

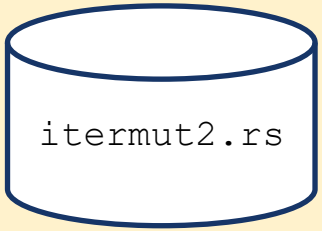
- I **contenitori** presenti nella libreria standard mettono normalmente a disposizione tre metodi per ricavare un iteratore ai dati contenuti al loro interno
  - `iter()`, crea un iteratore che restituisce oggetti di tipo `&Item` e non consuma il contenuto del contenitore
  - `iter_mut()`, crea un iteratore che restituisce oggetti di tipo `&mut Item` e permette di modificare gli elementi all'interno del contenitore
  - `into_iter()`, crea un iteratore che prende possesso del contenitore (e lo consuma) e restituisce oggetti di tipo Item estraendoli dal contenitore.

```rust
fn main() {
    let numbers = [1, 2, 3, 4, 5];

    // Itera attraverso ciascun elemento dell'array e stampalo

    for num in numbers.iter() {
        println!("{}", num);
    }

    for num in &numbers {
        println!("{}", num);
    }
}
```



```rust
fn main() {
    let mut numbers = vec![1, 2, 3, 4, 5];

    // Incrementa ogni numero nel vettore di 1 utilizzando iter_mut()
    for num in numbers.iter_mut() {
        *num += 1;
    }

    // Stampa il vettore dopo l'incremento
    println!("{:?}", numbers); // Output: [2, 3, 4, 5, 6]

    // Incrementa ogni numero nel vettore di 1 utilizzando &mut
    for num in &mut numbers {
        *num += 1;
    }

    // Stampa il vettore dopo l'incremento
    println!("{:?}", numbers); // Output: [3, 4, 5, 6, 7]
}
```

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

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

    for s in v.iter_mut() {
        s.push_str("bis");
    }

    for s in &v {
        println!("{}", s); // Output: a1bis, b1bis, c1bis
    }
}
```

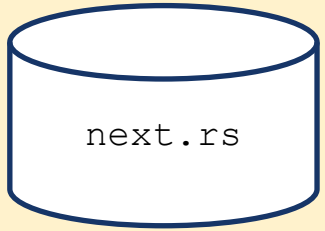


```rust
fn main() {

    let v = vec![10.0, 30.0, 50.0, 90.0];

    let mut sum=f64::default();

    // into_iter consuma il contenitore
    for num in v.into_iter() {
        sum += num;
    }
    println!("{}", sum);

    let v2 = vec![1, 3, 5, 9];

    let mut sum2=i32::default();

    for num2 in v2 {
        sum2 += num2;
    }

    println!("{}", sum2);
}
```

```rust
fn main() {
    let numbers = vec![1, 2, 3, 4, 5];

    // Consuma il vettore e calcola la somma dei suoi elementi utilizzando into_iter()
    let sum: i32 = numbers.into_iter().sum();

    // Questo non è consentito, poiché numbers è già stato consumato
    // println!("{:?}", numbers);

    // Stampa la somma dei numeri
    println!("Somma: {}", sum); // Output: Somma: 15
}
```

```
fn main() {
    let mut v = vec!["a".to_string(), "b".to_string(), "c".to_string()];

    for s in v.iter_mut()
                {
                    s.push('1');
                }

    for s in v.iter() {
        println!("{:?} ", s)
    }

    println!("Oppure {:?} ", v);

    let v2 = vec![1, 2, 3, 4, 5];
    let mut sum = 0;
    for n in v2.into_iter(){
        sum += n;
    }
    println!("{:?}", sum);

}
```

```rust
fn main() {
    let mut numbers = vec![1, 2, 3, 4, 5];

    // Ottiene il primo elemento dall'iteratore
    if let Some(first_num) = numbers.iter().next() {
        println!("Il primo numero e' {} ", first_num);
    } else {
        println!("Nessun elemento trovato");
    }

    if let Some(first_num) = numbers.iter_mut().next() {
        *first_num += 1;
    } else {
        println!("Nessun elemento trovato");
    }
    for num in numbers.iter() {
        println!("{}", num);
    }
}
```



```rust
// Definiamo una struttura che implementerà il tratto Iterator
struct Contatore {
    count: usize,
    max: usize,
}
// Implementiamo il tratto Iterator
impl Iterator for Contatore {
    type Item = usize;
    fn next(&mut self) -> Option<Self::Item> {
        if self.count < self.max {
            let val = self.count;
            self.count += 1;
            Some(val)
        } else {
            None
        }
    }
}
fn main() {
    let mut contatore = Contatore { count: 0, max: 10 };

    // Usiamo l'iteratore per stampare i valori
    while let Some(i) = contatore.next() {
        println!("{}", i);
    }
}
```

```rust
trait Iterator{
  type Item;
  fn next(&mut self) -> Option<Self::Item>;
  …
}
```

```
struct MyRange {
    count: usize,
}

impl Iterator for MyRange {
    type Item = usize;

    fn next(&mut self) -> Option<Self::Item> {
        if self.count == 0 {
            None
        } else {
            self.count -= 1;
            Some(self.count)
        }
    }
}

fn main() {
    let range_iter = MyRange {count: 20};

    for n in range_iter {
        println!("Next number: {}", n);
    }
}
```

```rust
struct ContatoreFibonacci {
    a: usize,
    b: usize,
    max: usize,
}
impl Iterator for ContatoreFibonacci {
    type Item = usize;
    // Definiamo il metodo next per ottenere il prossimo elemento
    fn next(&mut self) -> Option<Self::Item> {
        if self.a < self.max {
            let valore = self.a;
            let nuovo_valore = self.a + self.b;
            self.a = self.b;
            self.b = nuovo_valore;
            Some(valore)
        } else {
            None
        }
    }
}
fn main() {
    let fibonacci_iteratore = ContatoreFibonacci { a: 0, b: 1, max:1000 };
    for numero in fibonacci_iteratore {
        println!("Fibonacci: {}", numero);
    }
}
```

Tipo Iterable

- Un tipo **iterable** può segnalare la capacità di essere esplorato tramite un iteratore, implementando il tratto **`std::iter::IntoIterator`**
- Il tratto `IntoIterator` è un tratto di conversione che consente a un tipo di essere convertito in un iteratore
- Per implementare `IntoIterator` per un tipo personalizzato, è necessario definire un metodo `into_iter` che restituisca un iteratore per la collezione. Questo iteratore deve implementare il tratto Iterator e fornire un metodo `next` che restituisca gli elementi della collezione uno alla volta.

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

```rust
struct Pixel {
    r: i8,
    g: i8,
    b: i8,
}
impl IntoIterator for Pixel {
    type Item = i8;
    type IntoIter = std::array::IntoIter<i8, 3>;
    fn into_iter(self) -> Self::IntoIter {
        std::array::IntoIter::new([self.r, self.g, self.b])
    }
}
fn main() {
    let pixel = Pixel { r: 54, g: 23, b: 74 };
    let mut iter = pixel.into_iter();
    // Chiamare next restituirà il prossimo elemento dell'iteratore, se presente
    if let Some(component) = iter.next() {
        println!("Il primo componente è: {}", component);
    }
    // Puoi continuare a chiamare next per ottenere i successivi elementi
    if let Some(component) = iter.next() {
        println!("Il secondo componente è: {}", component);
    }
    if let Some(component) = iter.next() {
        println!("Il terzo componente è: {}", component);
    }
    // Se chiami next dopo che tutti gli elementi sono stati consumati, otterrai None
    if let Some(component) = iter.next() {
        println!("Questo non verrà stampato perché non ci sono più elementi.");
    } else {
        println!("Non ci sono più componenti.");
    }
}
```

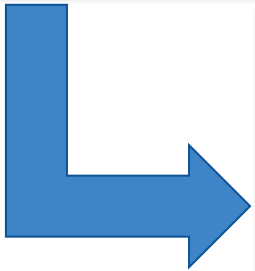
```rust
trait IntoIterator where Self::IntoIter: Iterator<Item=Self::Item> {
  type Item;
  type IntoIter: Iterator;
  fn into_iter(self) -> Self::IntoIter;
}
```

Terminologia

- **Iterator:** qualunque tipo che implementa il Tratto Iterator. I tipi iterator devono definire il metodo `next` che restituisce i valori successivi nella sequenza
- **Iterable**: qualunque tipo che implementa il Tratto IntoIterator
- Un **iterator** è ricavato da un tipo **iterable** chiamando il metodo `into_iter()`
- Un iterator produce **values**
- I singoli valori che l'iteratore produce si chiamano **items**
- Il codice che riceve gli items che un iteratore produce e genera un nuovo iteratore è detto **adapter**
- Il codice che consuma un iteratore e produce un risultato è detto **consumer**.

Iteratori e cicli for

```
let values = vec![1, 2, 3, 4, 5];
for x in values { println!("{}", x); }
```



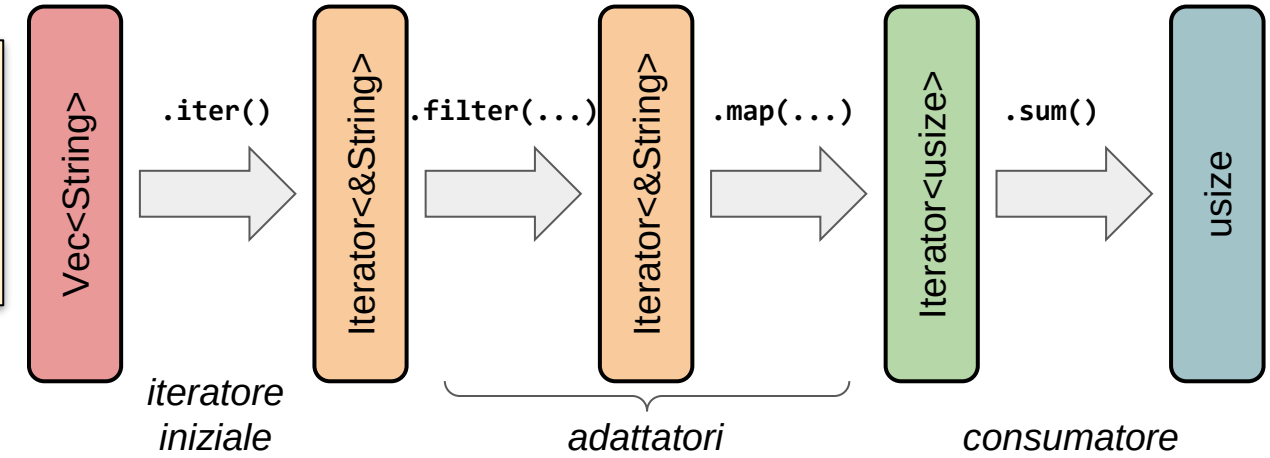
```
let values = vec![1, 2, 3, 4, 5];
{
    let result = match IntoIterator::into_iter(values) {
        mut iter => loop {
            let next;
            match iter.next() {
                Some(val) => next = val,
                None => break,
            };
            let x = next;
            let () = { println!("{}", x); };
        },
    };
    result
}
```

Il compilatore trasforma i cicli for in codice basato sugli iteratori

Adattatori

- Il tratto Iterator definisce un nutrito gruppo di metodi che consumano un iteratore e ne derivano uno differente, in grado di offrire funzionalità ulteriori
  - Possono essere combinati in catene più o meno lunghe al termine delle quali occorre porre un consumatore finale
  - Tutti gli adattatori sono infatti pigri (lazy) di natura e non invocano il metodo next() dell'oggetto a monte se non a seguito di una richiesta proveniente da un loro consumatore

```
let v: Vec<String> = … ;
v.iter()
   .filter(|x| x.len()<4)
   .map(|x|x.len())
   .sum();
```



Adattatori

- `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)

Adattatori

- `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

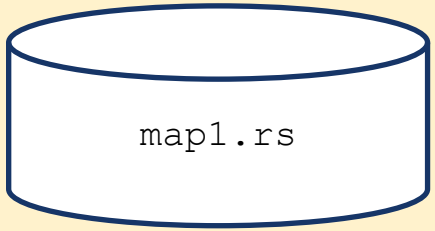
Adattatori

- `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
- `...`

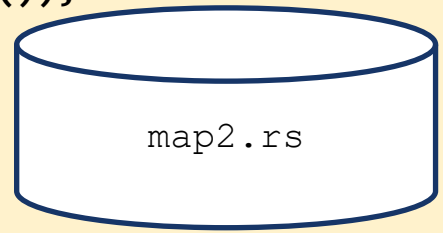
map()

- Trasforma gli elementi di un iteratore applicando una funzione a ciascun elemento e restituendo un nuovo iteratore che contiene i risultati della trasformazione.

```
fn main() {
    let numbers = vec![1, 2, 3, 4, 5];

    // Creiamo un iteratore dal vettore
    // applichiamo il metodo map per raddoppiare ogni elemento

    let doubled_iter = numbers.iter().map(|&x| x * 2);

    // Stampiamo i risultati
    for doubled in doubled_iter {
        println!("{}", doubled);
    }
}
```



```rust
fn main() {
    let words = vec!["hello", "world", "how", "are", "you"];

    // Creiamo un iteratore dal vettore
    // applichiamo il metodo map per ottenere la lunghezza di ogni parola

    let word_uppercase_iter = words.iter().map(|word| word.len());

    // Stampiamo i risultati
    for name in word_uppercase_iter {
        println!("{}", name);
    }

    // applichiamo il metodo map per ottenere la conversione di ogni parola
    let word_uppercase_iter = words.iter().map(|w| w.to_uppercase());

    // Stampiamo i risultati
    for name in word_uppercase_iter {
        println!("{}", name);
    }
}
```



filter()

- Crea un nuovo iteratore che include solo gli elementi che soddisfano un predicato.

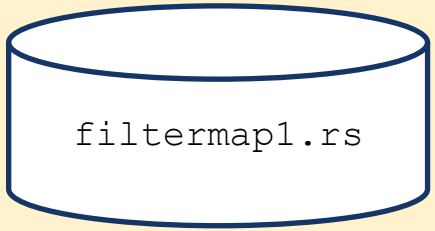
```
fn main() {
    let numbers = vec![1, 2, 3, 4, 5, 6, 7, 8, 9];


// Utilizzo di filtri per selezionare solo i numeri pari
let even_numbers = numbers.iter().filter(|&x| x % 2 == 0);

// Stampiamo i numeri pari
for n in even_numbers {
        println!("Even number: {}", n);
    }
}
```

```rust
fn main() {
    let nomi = vec!["Alice", "Bob", "Anna", "Carl", "David"];
    let nomi_filtrati = nomi.iter()
        .filter(|&nome| nome.starts_with("A"));

    for n in nomi_filtrati {
    println!("{:?}", n); // Stampa: ["Alice", "Anna"]
    }
}
```

filter_map()

- Applica una funzione a ciascun elemento e restituisce solo gli elementi risultanti che non sono None.

```
fn main() {
    let numbers = vec![1, 2, 3, 4, 5];

    // Utilizzo di filter_map per filtrare e mappare i numeri pari
    let even_numbers= numbers
        .iter()
        .filter_map(|&x| {
            if x % 2 == 0 {
                Some(x)
            } else {
                None
            }
        });

    for n in even_numbers {
        println!("{:?}", n);
    } // Stampa: 2, 4
}
```



```
fn main() {
    let dati = vec!["42", "93", "NaN", "42", "18", "77", "invalido"];
    let numeri_validi = dati.into_iter()
        .filter_map(|s| s.parse::<i32>().ok());

    for n in numeri_validi {
        println!("{:?}", n);
    } // Stampa: [42, 93, 18, 77]
}
```

flatten()

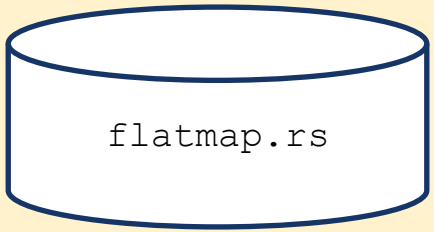
- Appiattisce una struttura dati composta da iterabili annidati e genera un singolo iteratore che contiene tutti gli elementi.

```
fn main() {
    let nested_numbers = vec![vec![1, 2, 3], vec![4, 5, 6], vec![7, 8, 9]];

    // Appiattisce la struttura nested di vettori in un singolo vettore
    let flattened_numbers = nested_numbers.into_iter().flatten();

    for n in flattened_numbers {
        println!("Numeri appiattiti: {:?}", n);
    }
}
```

flat_map()

- Consente di mappare ogni elemento dell'iteratore in un altro iteratore, quindi appiattisce il risultato in un singolo iteratore.

```
fn main() {
    let numbers = vec![1, 2, 3, 4];
    let new_numbers = numbers.iter()
        .flat_map(|&x| vec![x, x * x, x * x * x]);

    for n in new_numbers {
        println!("{:?}", n);
    } // Output: [1, 1, 1, 2, 4, 8, 3, 9, 27, 4, 16, 64]
}
```



take()

- Crea un nuovo iteratore che restituisce solo un numero specificato di elementi iniziali. Una volta raggiunto il limite specificato, l'iteratore smette di restituire elementi.

```
fn main() {
        let numbers = vec![1, 2, 3, 4, 5];

        // Prende solo i primi 3 numer i dall'inizio del vettore
        let first_three= numbers.iter().take(3);

        for n in first_three {
        println!("{:?}", n);
        } // Output: [1, 2, 3]
}
```

```
fn main() {
    let numbers = vec![10, 20, 30, 40, 50];

    // Prende i primi due numeri maggiori di 20 dall'inizio del vettore
    let first_two_over_twenty = numbers.iter()
        .filter(|&num| *num > 20)
        .take(2);
    for n in first_two_over_twenty {
    println!("Primi due numeri maggiori di 20: {:?}", n);
    }
}
```

```
fn main() {
    let numbers = vec![10, 20, 30, 40, 50];

    let first_two_over_twenty = numbers.iter()
        .take(3)
        .filter(|&num| *num > 10);

    for n in first_two_over_twenty {
    println!("Numeri maggiori di 10 tra i primi 3: {:?}", n);
    }
}
```

take_while()

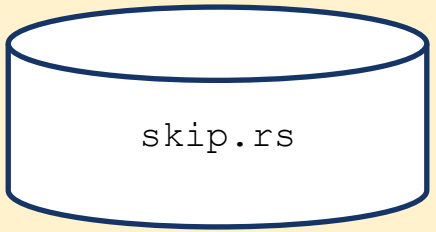
- Crea un nuovo iteratore che restituisce gli elementi finché una determinata condizione è vera. Una volta che la condizione diventa falsa, l'iteratore smette di restituire elementi.

```
fn main() {
    let numbers = vec![5, 10, 15, 20, 22, 30];

    // Prende numeri finché sono multipli di 5
    let multiples_of_five = numbers.iter()
        .take_while(|&num| *num % 5 == 0);

    for n in multiples_of_five {
    println!("Multipli di 5: {:?}", n);
    }
}
```

skip()

- Crea un nuovo iteratore che salta un numero specificato di elementi all'inizio dell'iteratore originale e restituisce tutti gli elementi rimanenti.

```
fn main() {
    let numbers = vec![1, 4, 5, 6, 7];

    // Salta i primi due numeri
    let skipped = numbers.iter().skip(2);

    for n in skipped {
    println!("Valori dopo i primi 2: {:?}", n);
    }
}
```

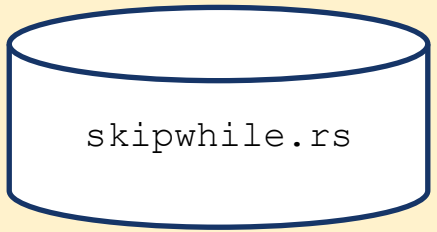


```
fn main() {
    let numbers = vec![1, 40, 5, 60, 7, 45, 34, 99];

    // Salta i primi due numeri
    let skipped = numbers.iter()
        .skip(2)
        .filter(|&num| *num > 10)
        .take(3);
    ;

    for n in skipped {
    println!("Dopo i primi 2, i primi 3 valori superiori a 10: {:?}", n);
    }
}
```

skip_while()

- Crea un nuovo iteratore che salta elementi fino a quando un predicato fornisce `false` o `None`. Una volta che il predicato restituisce true per la prima volta, tutti gli elementi successivi (compreso il primo per cui il predicato è diventato true) vengono inclusi nell'iteratore risultante.

```
fn main() {
        let numbers = vec![1, 3, 5, 7, 2, 7, 8, 9, 10];

        // Salta numeri fino a quando non trova un numero pari
        let skipped = numbers.iter()
                             .skip_while(|&num| *num % 2 != 0);

    for n in skipped {
    println!("Tutti i numeri a partire dal primo pari: {:?}", n);
    }
}
```

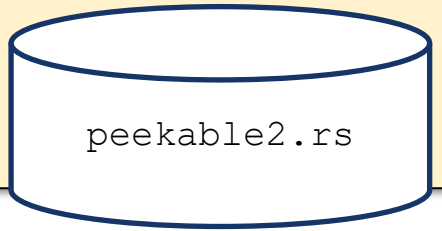


`peekable()`

- Crea un iteratore che permette di guardare l'elemento successivo senza che l'iteratore proceda al prossimo elemento
- I metodi `peek()` e `peek_mut()` permettono di accedere al valore successivo senza consumarlo.

```
fn main() {
    let numbers = vec![1, 2, 3, 4, 5];
    let mut peekable_numbers = numbers.iter().peekable();
    // Controlla se c'è un elemento successivo e lo stampa senza consumarlo
    if let Some(&next_number) = peekable_numbers.peek() {
        println!("Elemento successivo: {}", next_number);
    }
    // Ora si può consumare l'elemento
    if let Some(next_number) = peekable_numbers.next() {
        println!("Elemento consumato: {}", next_number);
    }
    for number in peekable_numbers {
        println!("Elemento successivo: {}", number); // Stampa gli elementi rimanenti
    }
}
```

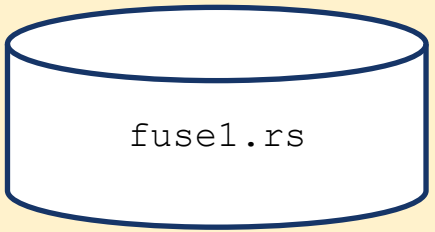
```rust
fn main() {
    let numeri = vec![1, 2, 3, 4];
    let mut iteratore_peekable = numeri.iter().peekable();

    while let Some(numero) = iteratore_peekable.next() {
        println!("Valore dall'iteratore: {}", numero);
        match iteratore_peekable.peek() {
            Some(prossimo) => println!("Sbirciando il prossimo valore: {}", prossimo),
            None => println!("Non ci sono altri valori"),
        }
    }
}
```



fuse()

- Crea un iteratore che si ferma al primo None che incontra e continuerà a restituire None.

```
fn main() {
    let numbers = vec![1, 2, 3];
    let mut iter = numbers.iter().fuse();

    // Stampa i primi due numeri
    println!("First: {:?}", iter.next());  // Some(1)
    println!("Second: {:?}", iter.next()); // Some(2)
    println!("Third: {:?}", iter.next());  // Some(3)

    // Stampa None e Continua a restituire None
    println!("Fourth: {:?}", iter.next()); // None
    println!("Fifth: {:?}", iter.next());  // None
}
```



```
fn main() {
    let numbers = vec![1, 2, 3];
    let mut iter = numbers.iter().fuse();

    // Consuma l'iteratore completamente
    println!("Iteratore 1:");
    while let Some(&number) = iter.next() {
        println!("Numero: {}", number);
    }

    // Riprova a consumare l'iteratore
    println!("Iteratore 2:");
    while let Some(&number) = iter.next() {
        println!("Numero: {}", number);
    }
}
```

rev()

- Utilizzato per invertire l'ordine degli elementi.

```
fn main() {
        let numbers = vec![1, 2, 3, 4, 5];
        let reversed_numbers= numbers.iter().rev();

        println!("Numeri originali: {:?}", numbers);

        for n in reversed_numbers {
                println!("Numeri invertiti: {:?}", n);
        }
}
```

```
fn main() {
        // Creazione di un range da 1 a 5
        let range = 1..=5;

        // Inversione dell'ordine del range utilizzando il metodo rev
        let reversed_range= range.rev();

        for n in reversed_range {
                println!("Nuovo Range: {:?}", n);
        }
}
```

inspect()

- Utilizzato per scopi di debug: applica una chiusura all'item senza modificare il valore

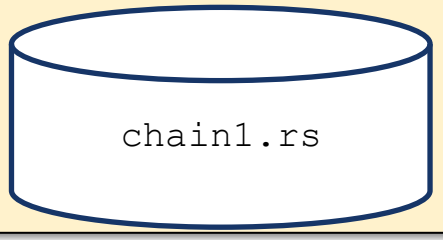
```
fn main() {
    let numbers = vec![1, 2, 3, 4, 5];

    // Utilizziamo inspect per stampare ogni elemento prima di moltiplicarlo per 2
    let doubled_numbers = numbers.iter()
        .inspect(|&x| println!("Elemento: {}", x))
        .map(|&x| x * 2);

    for n in doubled_numbers {
        println!("Raddoppiato: {:?}", n);
        }
    }
```

```
fn main() {
    let numbers = vec![1, 2, 3, 4, 5];

    // Utilizziamo inspect per controllare se un elemento è pari o dispari
    let parity_check= numbers.iter()
        .inspect(|&x| {
            if x % 2 == 0 {
                println!("{} è pari", x);
            } else {
                println!("{} è dispari", x);
            }
        });

    for n in parity_check {
        println!("Numero: {:?}", n);
        }
    }
```

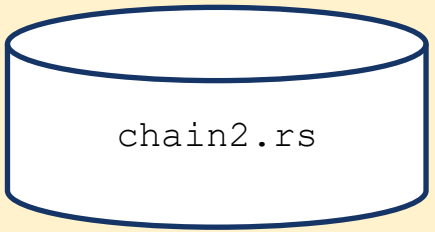
chain()

- Appende un iteratore ad un altro

```
fn main() {
    let numbers1 = vec![1, 2, 3];
    let numbers2 = vec![4, 5, 6, 7];

    // Concateniamo i due vettori utilizzando il metodo chain
    let chained_numbers = numbers1
            .iter()
            .chain(numbers2.iter());
    for n in chained_numbers {
        println!("Numeri concatenati: {:?}", n);
        }
}
```



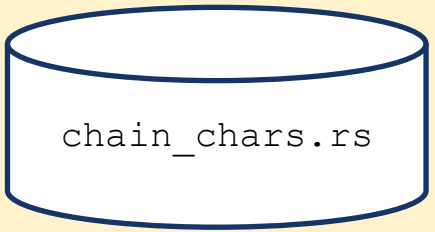
```
fn main() {
    let strings = vec!["hello", "world"];
    let numbers = vec![1, 2, 3];

    // Concateniamo il vettore di stringhe con il vettore di numeri
    let concatenated = strings.iter()
        .map(|s| s.to_string())
        .chain(numbers.iter().map(|&n| n.to_string()));

    for s in concatenated {
        println!("elemento: {:?}", s);
    }

}
```



chars()

- Chiamato su una stringa restituisce un iteratore che produce sequenzialmente i caratteri all'interno della stringa

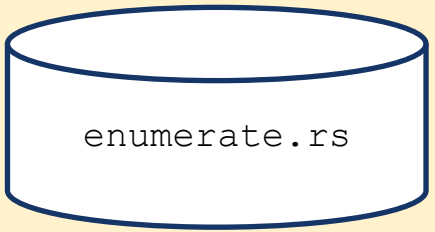
```
fn main() {
    let words = vec!["hello", "world"];
    let chars = "123";

    // Concateniamo l'iteratore delle parole con l'iteratore dei caratteri
    let chained_sequence = words.iter()
        .flat_map(|word| word.chars())
        .chain(chars.chars());

    for n in chained_sequence{
        println!("Carattere: {:?}", n);
    }

}
```



enumerate()

- Genera un iteratore che produce la tupla (indice, valore) per ciascun elemento di una sequenza

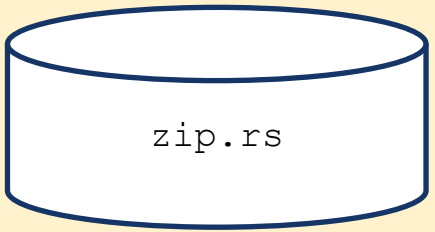
```
fn main() {
    let my_vec = vec!["a", "b", "c"];

    // Otteniamo un iteratore che produce coppie (indice, valore)
    let mut iter = my_vec.iter().enumerate();

    // Iteriamo attraverso ogni coppia (indice, valore)
    while let Some((index, value)) = iter.next() {
        println!("Indice: {}, Valore: {}", index, value);
    }
}
```



zip()

- Combina 2 iteratori in un solo iteratore che produce una sequenza di tuple che contengono un elemento da ciascun iteratore.

```
fn main() {
    let numbers = vec![1, 2, 3];
    let letters = vec!['a', 'b', 'c'];

    // Otteniamo un iteratore che produce tuple (numero, lettera)
    let mut iter = numbers.iter().zip(&letters);

    // Iteriamo attraverso ogni coppia (numero, lettera)
    while let Some((number, letter)) = iter.next() {
        println!("Numero: {}, Lettera: {}", number, letter);
    }
}
```



unzip()

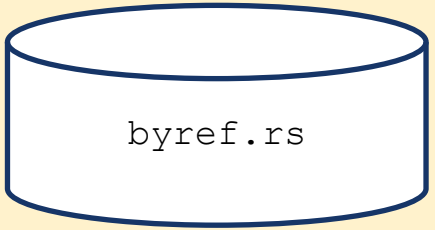
- Scompatta una collezione di tuple in 2 collezioni separate, una contenente tutti gli elementi nella prima posizione delle tuple e l'altra contenente tutti gli elementi nella seconda posizione.

```
fn main() {
    let data = vec![(1, 'a'), (2, 'b'), (3, 'c')];

    // Unzip della collezione di tuple in due iteratori separati
    let (numbers, characters): (Vec<_>, Vec<_>) = data.into_iter().unzip();

    println!("Numeri: {:?}", numbers);
    println!("Caratteri: {:?}", characters);
}
```

by_ref()

- Utilizzato per ottenere un riferimento all'iteratore corrente senza consumarlo
- Utile quando si desidera utilizzare l'iteratore più volte senza consumarlo o per evitare di dover possedere l'iteratore.

```
fn main()
{
    let mut numeri = vec![1, 2, 3].into_iter();

    // Utilizziamo by_ref per prendere i primi due elementi
    // senza consumare completamente l'iteratore
    let primi_due = numeri.by_ref().take(2);

    for n in primi_due {
        println!("Primi 2: {}", n);
    }
    for n in numeri {
        println!("Tutti i numeri: {}", n);
    }
}
```



cloned()

- Prende un iteratore che produce riferimenti e restituisce un iteratore che produce valori clonati da questi riferimenti

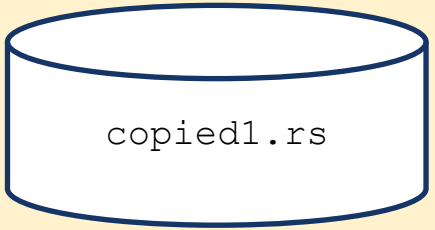
```rust
fn main() {
    let words = vec!["hello", "world", "rust"];

    // Otteniamo un iteratore che produce copie delle stringhe
    let cloned_iter = words.iter().cloned();

    // Stampiamo ogni parola nel nuovo iteratore
    for word in cloned_iter {
        println!("Parola: {}", word);
    }
    for word in words {
        println!("Parola: {}", word);
    }

}
```

```rust
fn main() {
    let numbers = vec![1, 2, 3, 4, 5];

    // Crea un iteratore che restituisce valori clonati
    let mut iter = numbers.iter().cloned();

    // Filtra e mappa i valori clonati
    let doubled_even_numbers = iter
        .filter(|n| n % 2 == 0)
        .map(|n| n * 2);

    // Stampa i numeri pari raddoppiati
    println!("Doubled even numbers:");
    for n in doubled_even_numbers {
        println!("{}", n);
    }

    // L'iteratore originale non è consumato, quindi possiamo utilizzarlo di nuovo
    println!("Original numbers:");
    for n in numbers {
        println!("{}", n);
    }
}
```

copied()

- Genera un iteratore che produce una copia dei valori originali.

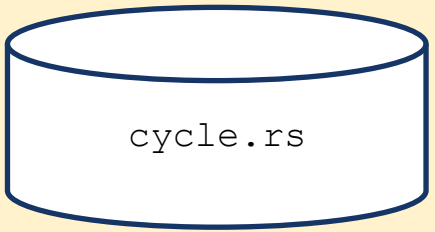
```
fn main() {
    let tuple_vec = vec![(1, 'a'), (2, 'b'), (3, 'c')];

    // Otteniamo un iteratore che produce copie delle tuple originali
    let copied_iter = tuple_vec.iter().copied();

    // Stampiamo ogni tupla nel nuovo iteratore
    for tuple in copied_iter {
        println!("Tuple: {:?}", tuple);
    }
    for tuple in tuple_vec{
        println!("Tuple: {:?}", tuple);
    }
}
```



cycle()

- Genera un iteratore che può ciclare ripetutamente attraverso gli elementi di una sequenza

```
fn main() {
    let numbers = vec![1, 2, 3];

    // Otteniamo un iteratore che cicla ripetutamente attraverso i numeri
    let mut cycle_iter = numbers.iter().cycle();

    // Stampiamo i primi 5 numeri del ciclo
    for _ in 0..5 {
        if let Some(num) = cycle_iter.next() {
            println!("Numero: {}", num);
        }
    }
}
```



Consumatori

- `collect<B>(self) -> B`
  - Trasforma un iteratore in una collezione
- `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
- `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

Consumatori

- `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 vuoto viene ritornato None
- `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_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

Consumatori

- `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
- `last(self) -> Option<Self::Item>`
  - Ritorna l'ultimo elemento dell'iteratore
- `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
- `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

collect()

- Consuma un iteratore e crea una collezione.

```
fn main() {
    let numbers = vec![1, 2, 3, 4, 5];

    // Filtriamo solo i numeri pari e li raccogliamo in un nuovo vettore
    let even_numbers: Vec<_> = numbers.iter()
        .filter(|&x| x % 2 == 0)
        .collect();

    println!("Numeri pari: {:?}", even_numbers);
}
```

for_each()

- Applica una chiusura su ciascun elemento

```
fn main() {
    let numbers = vec![1, 2, 3, 4, 5];

    // Stampiamo ogni numero del vettore
    numbers.iter().for_each(|&num| {
        println!("Numero: {}", num);
    });
}
```

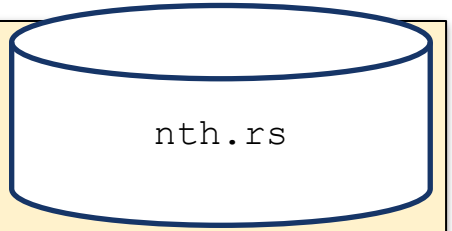
try_for_each()

- Applica una chiusura (che può fallire) su ciascun elemento


```
fn main() {
    let strings = vec!["programming", "computer"];

    // Stampa la lunghezza di ciascuna stringa, gestendo gli errori
    let result = strings.iter().try_for_each(|s| -> Result<(), ()> {
        if s.len() > 5 {
            println!("La lunghezza di '{}' è maggiore di 5.", s);
            Ok(())
        } else {
            Err(()) // Interrompe l'iterazione se la lunghezza è minore o uguale a 5
        }
    });
    // Controlla il risultato dell'iterazione
    match result {
        Ok(()) => println!("Tutte le stringhe hanno una lunghezza maggiore di 5."),
        Err(()) => println!("Almeno una stringa ha una lunghezza minore o uguale a 5."),
    }
}
```

nth()

- Utilizzato per ottenere l'elemento all'indice specificato.




```
fn main() {
    let numbers = vec![10, 20, 30, 40, 50];

    // Otteniamo il terzo elemento del vettore
    match numbers.iter().nth(2) {
        Some(&number) => println!("Terzo elemento: {}", number),
        None => println!("Nessun elemento trovato all'indice specificato."),
    }

    // Otteniamo il secondo elemento del vettore
    match numbers.iter().nth(1) {
        Some(&number) => println!("Secondo elemento: {}", number),
        None => println!("Nessun elemento trovato all'indice specificato."),
    }
}
```

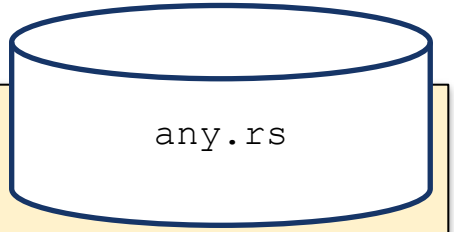
all()

- Verifica che tutti gli elementi di un iteratore soddisfano una determinata condizione.


```
fn main() {
    let numbers = vec![2, 4, 6, 8, 10];

    // Verifichiamo se tutti gli elementi sono pari
    let all_even = numbers.iter().all(|&x| x % 2 == 0);

    if all_even {
        println!("Tutti gli elementi sono pari.");
    } else {
        println!("Almeno un elemento non è pari.");
    }
}
```

any()

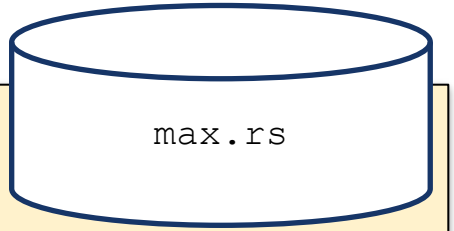
- Verifica se almeno un elemento soddisfi una condizione.




```
fn main() {
    let words = vec!["hello", "world", "rust", "programming"];

    // Verifichiamo se almeno una stringa ha una lunghezza maggiore di 5 caratteri
    let any_long_word = words.iter().any(|&word| word.len() > 5);

    if any_long_word {
        println!("Almeno una parola ha una lunghezza maggiore di 5 caratteri.");
    } else {
        println!("Nessuna parola ha una lunghezza maggiore di 5 caratteri.");
    }
}
```

max()

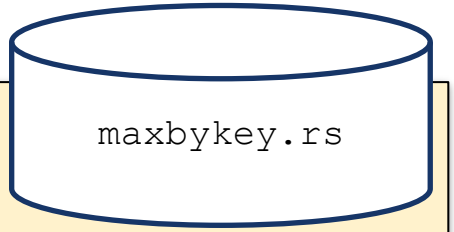
- Applicato su un iteratore che contiene elementi confrontabili, restituisce un Option, che in caso di successo riporta il valore massimo.




```
fn main() {
    let numbers = vec![10, 30, 20, 50, 40];

    // Trova il massimo elemento nell'iteratore
    let max_number = numbers.iter().max();

    match max_number {
        Some(max) => println!("Il massimo elemento è: {}", max),
        None => println!("L'iteratore è vuoto."),
    }
}
```

max_by_key()

- Applicato su un iteratore per calcolare il massimo di un iteratore in base al risultato di una funzione di confronto personalizzata.

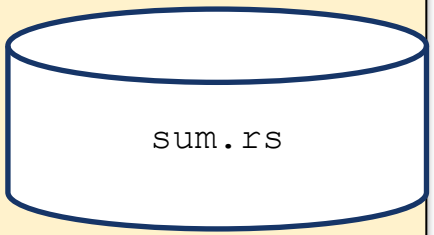



```
fn main() {
    let words = vec!["hello", "world", "rust", "programming"];

    // Trova la stringa con il maggior numero di caratteri
    let longest_word = words.iter().max_by_key(|word| word.len());

    match longest_word {
        Some(longest) => println!("La stringa più lunga è: {}", longest),
        None => println!("L'iteratore è vuoto."),
    }
}
```

sum()

- Applicato su un iteratore calcola la somma di tutti gli elementi dell'iteratore.

```
fn main() {
    let numbers = vec![1, 2, 3, 4, 5];

    // Calcola la somma di tutti gli elementi nell'iteratore
    let sum: i32 = numbers.iter().sum();

    println!("La somma di tutti gli elementi è: {}", sum);
}
```



find()

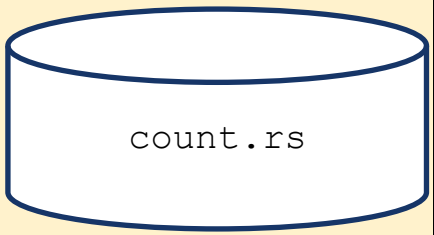
- Applicato su un iteratore cerca il primo elemento che soddisfa una determinata condizione. Restituisce Some(elemento) se l'elemento è stato trovato, altrimenti restituisce None.

```
fn main() {
    let numbers = vec![2, 4, 6, 7, 8, 9];

    // Trova il primo numero dispari nell'iteratore
    let first_odd = numbers.iter().find(|&x| *x % 2 != 0);

    match first_odd {
        Some(odd) => println!("Il primo numero dispari è: {}", odd),
        None => println!("Nessun numero dispari trovato."),
    }
}
```

count()

- Applicato su un iteratore conta quanti elementi appartengono alla sequenza

```
fn main() {
    let numbers = vec![2, 4, 6, 7, 8, 9];

    let odd = numbers.iter().filter(|&x| *x % 2 != 0).count();

    println!("Il numero di dispari è: {}", odd);

}
```



product()

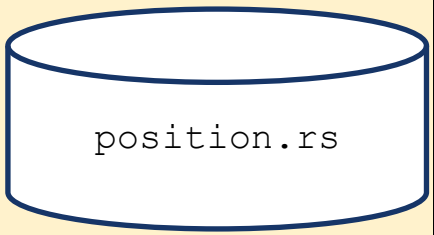
- Calcola il prodotto tra gli elementi di una sequenza.

```
fn main() {
    let numbers = vec![1, 2, 3];

    // Calcola il prodotto di tutti gli elementi dell'iteratore
    let product: u32 = numbers.iter().product();

    println!("Il prodotto di {:?} è {}", numbers, product);
}
```

position()

- Metodo utilizzato per trovare l'indice del primo elemento che soddisfa un determinato predicato

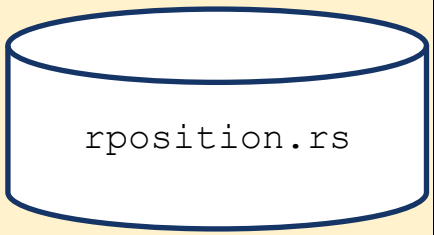
```
fn main() {
    let numbers = vec![1, 2, 3, 4, 5];

    // Find the index of the first even number
    let index = numbers.iter().position(|&x| x % 2 == 0);

    match index {
        Some(i) => println!("The first even number is at index {}", i),
        None => println!("No even number found"),
    }
}
```



rposition()

- Metodo utilizzato per trovare l'indice del primo elemento che soddisfa un determinato predicato partendo dal fondo

```
fn main() {
    let numbers = vec![1, 3, 5];

    // Find the index of the first even number
    let index = numbers.iter().rposition(|&x| x % 2 == 0);

    match index {
        Some(i) => println!("The last even number is at index {}", i),
        None => println!("No even number found"),
    }
}
```



last()

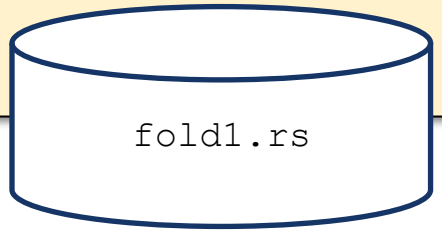
- Metodo utilizzato per trovare l'ultimo elemento di una sequenza

```
fn main() {
    let numbers = vec![1, 2, 3, 4, 5];

    // Ottieni l'ultimo numero pari
    let last_even = numbers
                    .iter()
                    .filter(|&x| x % 2 == 0)
                    .last();

    match last_even {
        Some(&number) => println!("L'ultimo numero pari è: {}", number),
        None => println!("Non ci sono numeri pari"),
    }
}
```

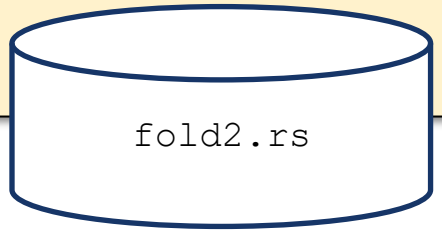
fold()

- Esegue la chiusura ricevuta accumulando i risultati sul primo argomento ricevuto

```
fn main() {
    let numbers = vec![1, 2, 3, 4, 5];

    // Somma tutti gli elementi della collezione
    let sum = numbers.iter().fold(0, |accumulatore, &x| accumulatore + x);

    println!("La somma di tutti gli elementi è: {}", sum);
}
```



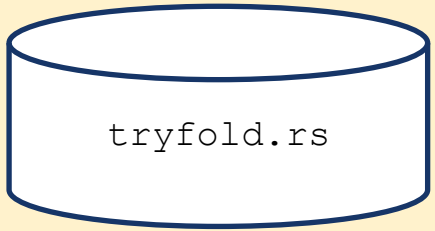
```
fn main() {
    let numbers = vec![1, 2, 3, 4, 5];

    // Calcola il prodotto di tutti gli elementi utilizzando il metodo fold()
    let product = numbers.iter().fold(1, |acc, n| acc * n);

    println!("Il prodotto di {:?} è {}", numbers, product);
}
```



try_fold()

- Simile a fold, ma gestisce anche gli errori.

```
fn main() {
    let numbers = vec![2, 3, 4, 5];

    // Calcola il prodotto di tutti gli elementi della collezione
    let product = numbers.iter().try_fold(1, |acc, &x| {
        if x != 0 {
            Ok(acc * x)
        } else {
            Err("Zero")
        }
    });

    match product {
        Ok(result) => println!("Il prodotto di tutti gli elementi è: {}", result),
        Err(err) => println!("Errore: {}", err),
    }
}
```



find_map()

- Cerca il primo elemento che soddisfa un predicato e restituisce il risultato di una funzione di mapping applicata a quell'elemento.

```
fn main() {
    let numbers = vec![1, 2, 3, 4, 5];

    // Cerca il primo numero pari e raddoppia il valore
    let result = numbers.iter().find_map(|&x| {
        if x % 2 == 0 {
            Some(x * 2)
        } else {
            None
        }
    });

    match result {
        Some(value) => println!("Il primo numero pari raddoppiato è: {}", value),
        None => println!("Nessun numero pari trovato"),
    }
}
```

partition()

- Divide una collezione in due parti, in base a un predicato fornito
- Restituisce una tupla contenente gli elementi che soddisfano il predicato e l'altra quelli che non lo soddisfano.

```
fn main() {
    let numbers = vec![1, 2, 3, 4, 5];

    // Dividi la collezione in numeri pari e dispari
    let (even_numbers, odd_numbers): (Vec<_>, Vec<_>) =
         numbers
         .into_iter()
         .partition(|&x| x % 2 == 0);

    println!("Numeri pari: {:?}", even_numbers);
    println!("Numeri dispari: {:?}", odd_numbers);
}
```

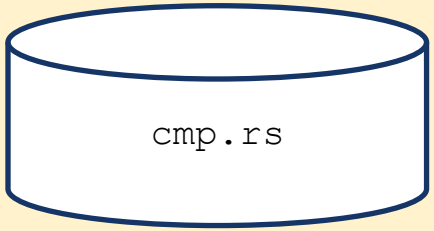
```rust
fn main() {
    let people = vec![
        ("Alice", 30),
        ("Bob", 25),
        ("Charlie", 35),
        ("David", 40),
    ];

    // Dividi la collezione in due parti in base all'età
    let (young, old): (Vec<_>,Vec<_>) = people.into_iter().partition(|&(_, age)| age < 35);

    println!("Giovani: {:?}", young);
    println!("Anziani: {:?}", old);
}
```

Confronti tra iteratori

- `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
- `...`

cmp()

- 2 iteratori possono essere confrontati convertendoli prima in collezioni di valori comparabili e poi confrontando le collezioni utilizzando il metodo cmp
- Il confronto avviene confrontando gli elementi corrispondenti delle 2 collezioni, considerando l'ordine degli elementi e il numero di elementi.

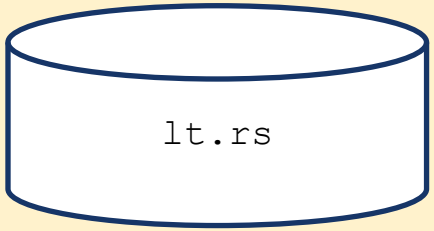



```
fn main() {
    let numbers1 = vec![1, 3, 2];
    let numbers2 = vec![1, 2, 4];

    // Confronto dei due vettori
    let comparison = numbers1.iter().cmp(numbers2.iter());

    match comparison {
        std::cmp::Ordering::Less => println!("Il primo vettore è minore"),
        std::cmp::Ordering::Equal => println!("I due vettori sono uguali"),
        std::cmp::Ordering::Greater => println!("Il primo vettore è maggiore"),
    }
}
```

eq(),ne(),lt(),le(),gt(),ge()

- 2 iteratori possono essere confrontati convertendoli prima in collezioni di valori comparabili e poi confrontando le collezioni utilizzando i metodi di confronto
- Il confronto avviene confrontando gli elementi corrispondenti delle 2 collezioni, considerando l'ordine degli elementi e il numero di elementi.

```
fn main() {
    let numbers1 = vec![1, 2, 3];
    let numbers2 = vec![4, 5, 6];

    // Confronto dei due iteratori di numeri interi
    let result = numbers1.iter().lt(numbers2.iter());

    if result {
        println!("Il primo iteratore è minore del secondo");
    } else {
        println!("Il primo iteratore non è minore del secondo");
    }
}
```



Risorse utili

- The Rust Programming Language book - Chapter 13: "Functional Language Features: Iterators and Closures"
  - https://doc.rust-lang.org/book/ch13-00-functional-features.html
- Rust by Example - Iterators
  - https://doc.rust-lang.org/rust-by-example/trait/iter.html
- Rust Iterators: A Guide
  - https://www.newline.co/@uint/rust-iterators-a-guide--80e35528
- Daily Rust: Iterators
  - https://adventures.michaelfbryan.com/posts/daily/iterators/