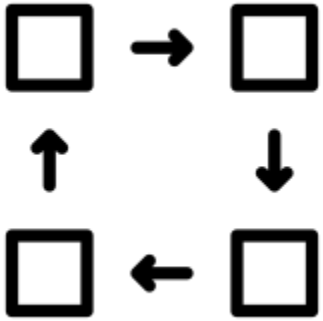


Iteratori

Accedere in modo indiretto ad una sequenza di valori

Iteratori

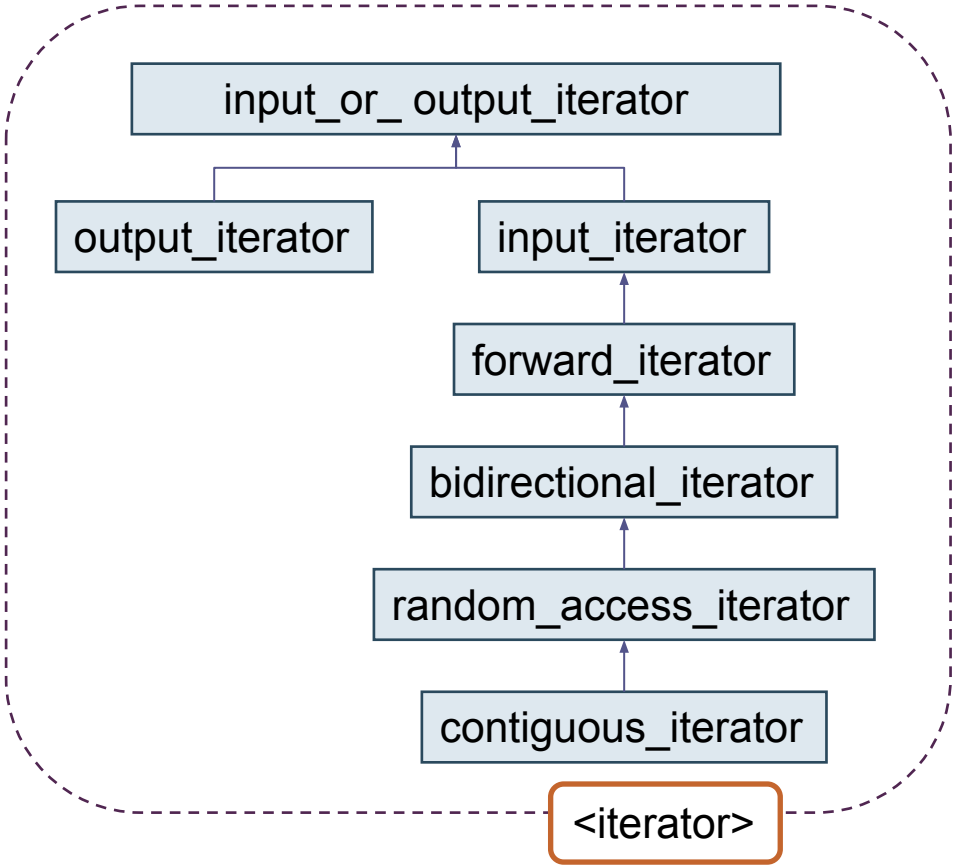
- Un iteratore è una struttura dati dotata di stato, in grado di generare una sequenza di valori
  - I valori possono essere estratti da un contenitore, di cui l'iteratore detiene un riferimento, o generati programmaticamente, come nel caso di un intervallo di valori
- Pressoché tutti i linguaggi "moderni" 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
- In C++, un iteratore è definito in modo "implicito", senza assegnargli un tipo specifico
  - Tecnica del "duck typing": *if it walks like a duck and it quacks like a duck, then it must be a duck*
  - 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)

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>;
  …// altri metodi con implementazione di default
}
```

- Un tipo **può** segnalare la capacità di essere esplorato tramite un iteratore, 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;
}
```

Implementare un iteratore in Rust

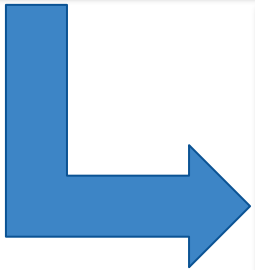
```
struct MyRange<const FROM: isize, const TO: isize> {}

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 }
    }
}
```

Implementare un iteratore in Rust

```
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 cicli for

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



Il compilatore trasforma i cicli for in codice basato sugli iteratori

```
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
}
```

http://xion.io/post/code/rust-let-unit.html

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()`, che restituisce oggetti di tipo `&Item` e non consuma il contenuto del contenitore
  - `iter_mut()`, che restituisce oggetti di tipo `&mut Item` e permette di modificare gli elementi all'interno del contenitore
  - `into_iter()`, che prende possesso del contenitore e restituisce oggetti di tipo Item estraendoli dal contenitore
- E' comune, per tali contenitori, dichiarare tre implementazioni distinte del tratto `IntoIterator`
  - Una per il tipo ***Container*** vero e proprio, che richiama il metodo `into_iter()`
  - Una per il tipo ***&Container***, che richiama `iter()`
  - Una per il tipo ***&mut Container***, che richiama `iter_mut()`
  - In alcuni casi (HashSet<T>, HashMap<T>, …) la terza implementazione non è fornita perché romperebbe le astrazioni

Iteratori e possesso

```
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
}
```

Derivare un iteratore

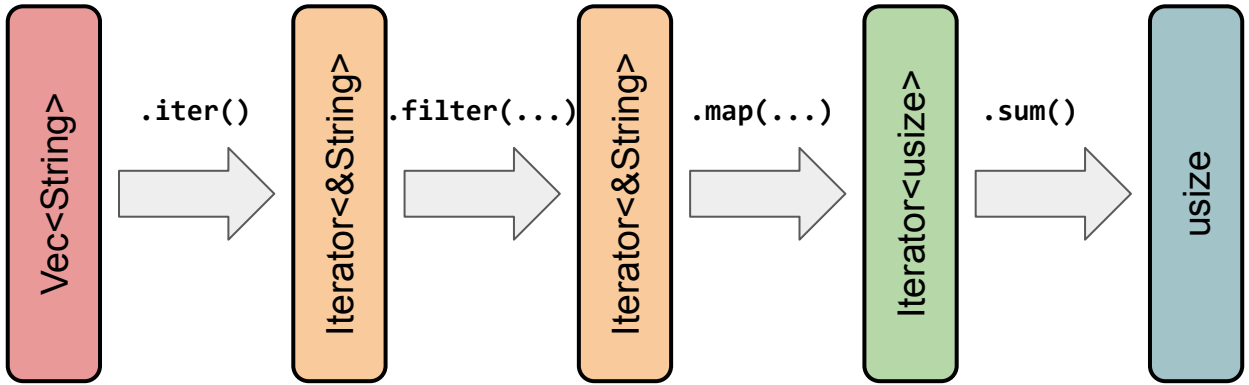
- Tra i metodi dotati di un'implementazione di default del tratto **`Iterator`** c'è anche il metodo **`into_iter()`** che si limita a restituire l'iteratore stesso
  - Questo significa che è possibile utilizzare la sintassi del ciclo for … indicando direttamente un iteratore

```
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
}
```

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
- `...`

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

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<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

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
- `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

Consumatori

- `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
- `...`

Risorse utili

- Rust Iterator Cheat Sheet
  - https://danielkeep.github.io/itercheat_baked.html