Standard Template Library

Programmazione Avanzata e Parallela
2022/2023

Alberto Casagrande

Programmare in C++ Senza Librerie

Si può fare, ma:

- richiede tempo
- si re-inventa la ruota, e.g.
- aumenta la possibilità di bug e l'inefficienza

E.g., come implementare `set<T>`? Con i **Red-Black Tree**

Avete voglia/tempo di implementare la versione più efficiente possibile dei *RBTree*?

La *Standard Template Library*

È una libreria che aggiunge delle funzionalità avanzate al C++

È standard perché *è inclusa nello standard ISO C++*

È distribuita assieme a ogni copilatore C++ che aderisce allo standard

Ci sono varie implementazioni (funzionalmente) equivalenti

Delle Vecchie Conoscenze...

Abbiamo già usato alcune funzionalità della STL:

- l'header `iostream` per l'I/O standard
- il namespace `std::chrono`
- la classe `std::string`
- le classi `std::less<T>` e `std::greater<T>`
- l'header `cmath` per l'utilizzo delle funzioni matematiche standard
- l'header `typeinfo` per l'utilizzo di `typeid()` e `type_info`

Le Componenti della *STL*

La STL ha 4 componenti principali:

- **Contenitori:** strutture dati quali insiemi, array a dimensione variabile, dizionari *à la* Python, etc.
- **Iteratori:** oggetti per visitare una struttura dati o produrre dati
- **Algoritmi:** ordinamento, ricerca binaria, unione, intersezione e differenza di insiemi di dati in contenitori
- **Funzionali:** operatori algebrici, logici e relazionali, e.g., `std::plus<T>`, `std::logical_and<T>`, e `std::less<T>`

I Contenitori della STL

Sono delle strutture dati per... contenere dei dati

Si dividono in:

- **Sequenziali:** forniscono accesso a una sequenza di elementi
- **Associativi:** consentono la ricerca tramite una *chiave* e sono:
  - **ordinati**
  - **non-ordinati**

I Contenitori Sequenziali

- `std::array<T, size_T N>` (C++11): array di dimensione fissa
- `std::vector<T,A>`: array a dimensione variabile
- `std::forward_list<T,A>` (C++11): liste concatenate semplici
- `std::list<T,A>`: liste doppiamente concatenate
- `std::deque<T,A>`: code in cui gli elementi possono essere aggiunti sia in testa che in coda

`std::array`

È definita nell'header `array` come

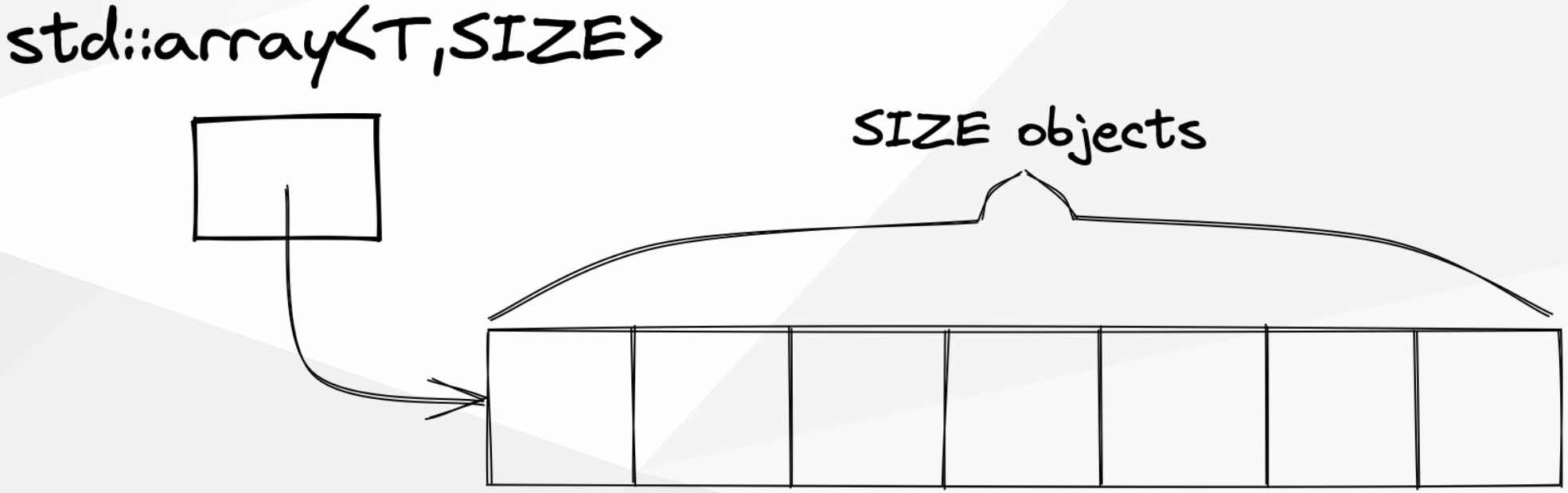
```
template<
    class T,
    std::size_t N
> class vector;
```

Il secondo parametro del template è la dimenisone dell'array

Gli elementi sono contigui in memoria, i.e., `&(a[i])+1==&(a[i+1])`

Tra gli altri, fornisce i metodi `front()`, `back()`, `size()`, `fill(T& t)`, `swap(std::array<T, M>&)`, e `operator[](const size_t)`

Una Implementazione per `std::array`



`std::array`: esempio di utilizzo

```cpp
#include<array>

int main() {
  std::array<int, 3> a{ 0, -5, 3};

  std::cout  << "a.size(): " << a.size()    // stampa "a.size(): 3"
             << " a.front(): " << a.front()  // stampa " a.front(): 0"
             << " a.back(): " << a.back()    // stampa " a.back(): 3"
             << std::endl;

  a.fill(5);   // scrive 5 in tutte le celle dell'array
  return 0;
}
```

`std::vector`

È definita nell'header `vector` come

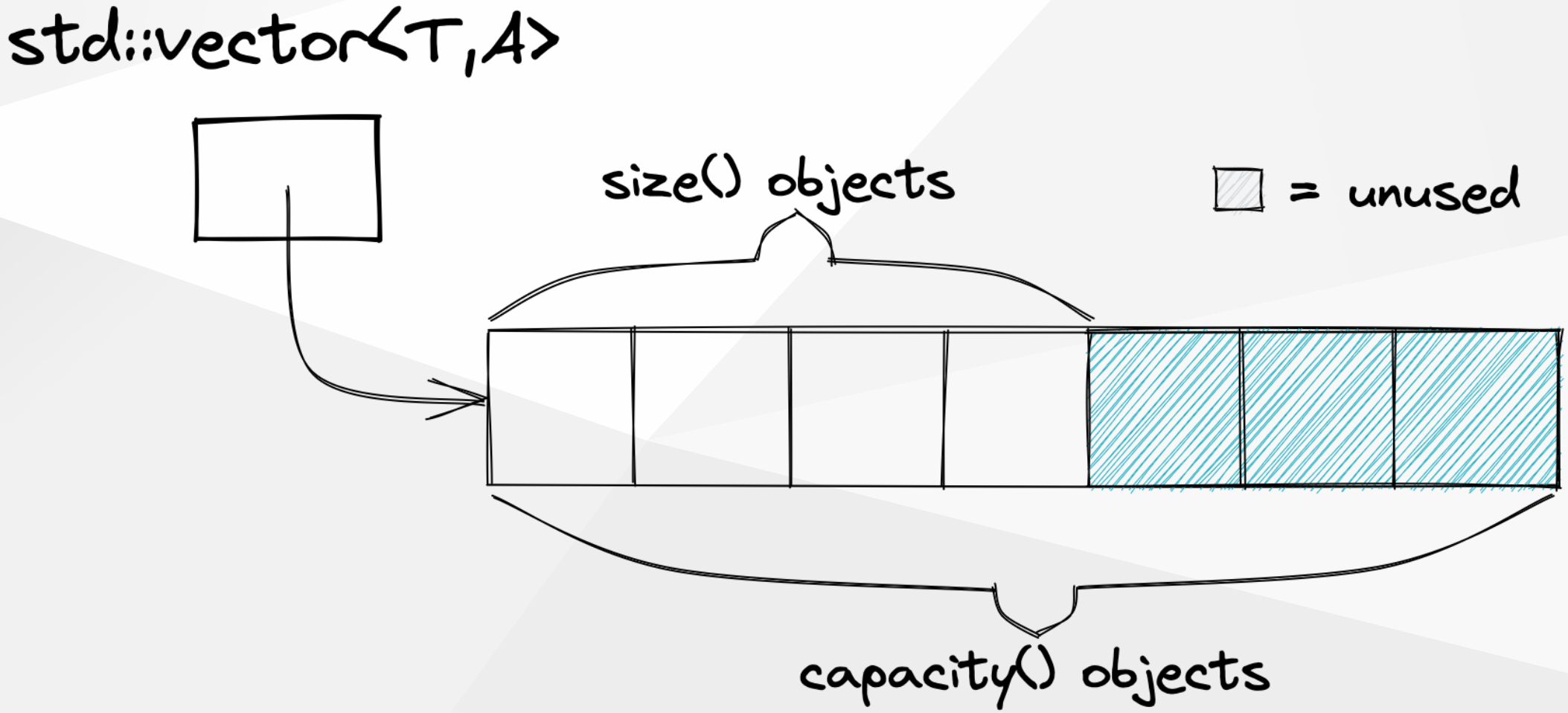
```
template<
    class T,
    class Allocator = std::allocator<Key>
> class vector;
```

`std::allocator<T>` allocare e deallocare oggetti del tipo `T[]`

Gli elementi sono contigui in memoria

Ha anche i metodi `push_back(const T&)`, `pop_back()`, e `capacity()`

Una Implementazione per `std::vector`



Una Implementazione per `push_back(const T&)`

1. se `size()==capacity()`, alloca un nuovo array di dimensione `2*capacity()` e copia i valori del vecchio array nel nuovo array (Complessità ammortizzata $O(1)$)
2. copia `value` nella posizione `size()` e incrementa `size()` ($O(1)$)

Complessità totale ammortizzata $O(1)$

`std::vector`: esempio di utilizzo

```cpp
#include<vector>

int main() {
  std::vector<int> v{ 0, -5, 3};

  v.push_back(7);

  std::cout  << "v.size(): " << v.size()     // stampa "a.size(): 4"
             << " v.front(): " << v.front()  // stampa " v.front(): 0"
             << " v.back(): " << v.back()    // stampa " v.back(): 7"
             << std::endl;

  return 0;
}
```

`std::forward_list` (C++11)

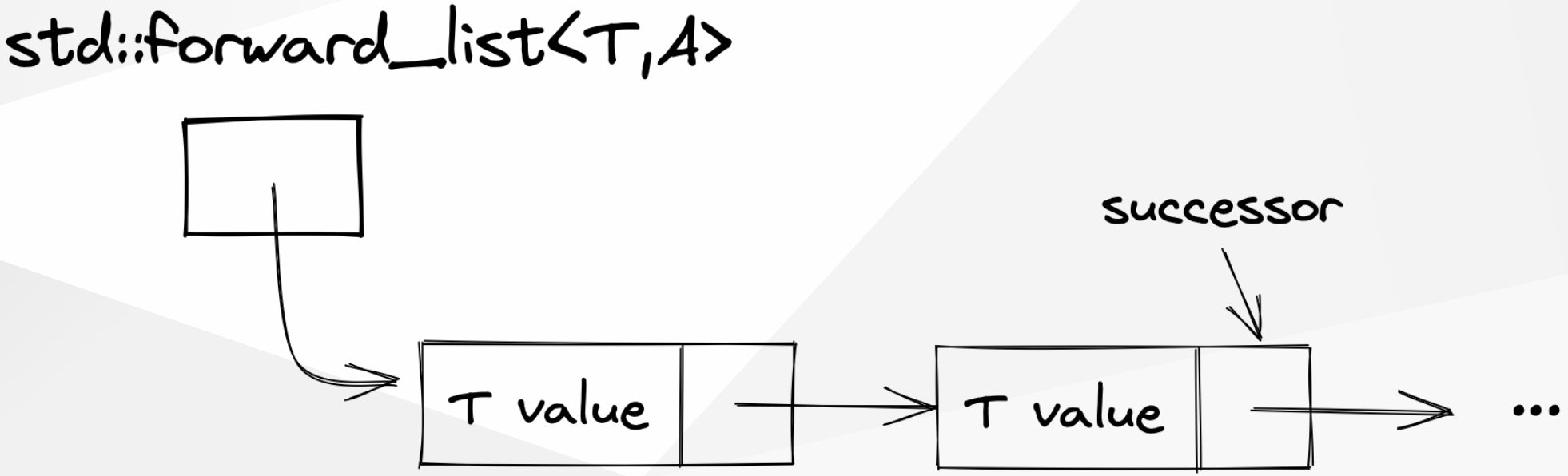
È definita nell'header `forward_list` come

```
template<
    class T,
    class Allocator = std::allocator<Key>
> class forward_list;
```

Rappresenta liste concatenate

Tra gli altri, fornisce i metodi `push_front(const T&)`, `pop_front()`, `sort()`, `unique()` e `size()`

Una Implementazione per `std::forward_list`



`std::forward_list`: esempio di utilizzo

```cpp
#include<forward_list>

int main() {
  std::forward_list<int> v{ 0, -5, 3};

  v.push_front(7);

  std::cout  << " v.front(): " << v.front()  // stampa " v.front(): 7"
             << std::endl;

  return 0;
}
```

`std::list`

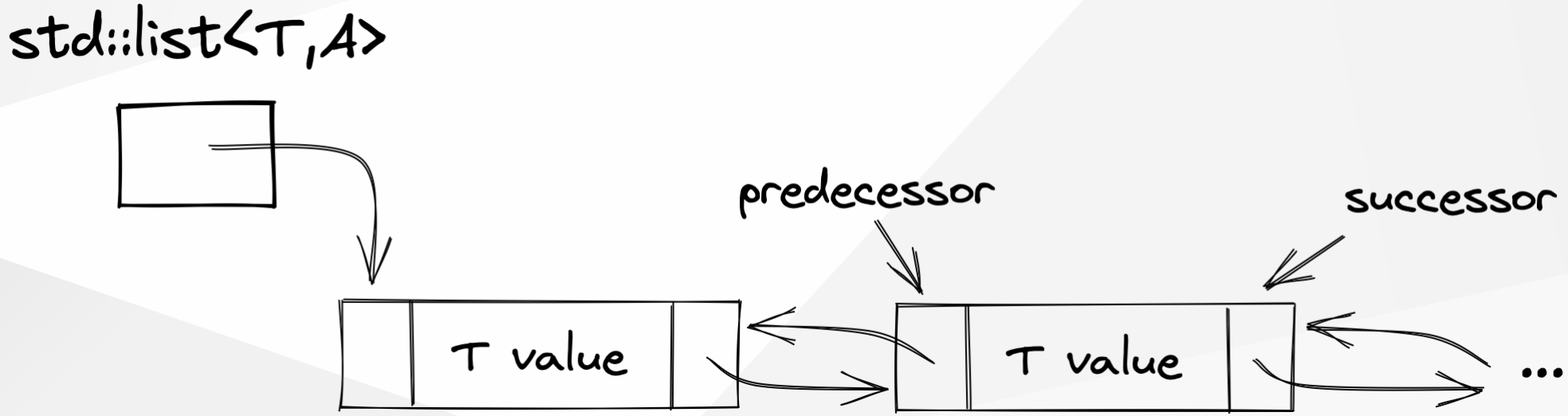
È definita nell'header `list` come

```
template<
    class T,
    class Allocator = std::allocator<Key>
> class list;
```

Rappresenta liste doppiamente concatenate

Tra gli altri, fornisce i metodi `push_front(const T&)`, `pop_front(const T&)`, `reverse()`, e `sort()`

Una Implementazione per `std::list`



`std::list`: esempio di utilizzo

```cpp
#include<list>

int main() {
  std::list<int> l{ 0, -5, 3};

  l.push_front(-2);
  l.push_back(7);

  std::cout  << "l.front(): " << l.front()  // stampa "l.front(): -2"
             << " l.back(): " << l.back()    // stampa " l.back(): 7"
             << std::endl;

  return 0;
}
```

`std::deque`

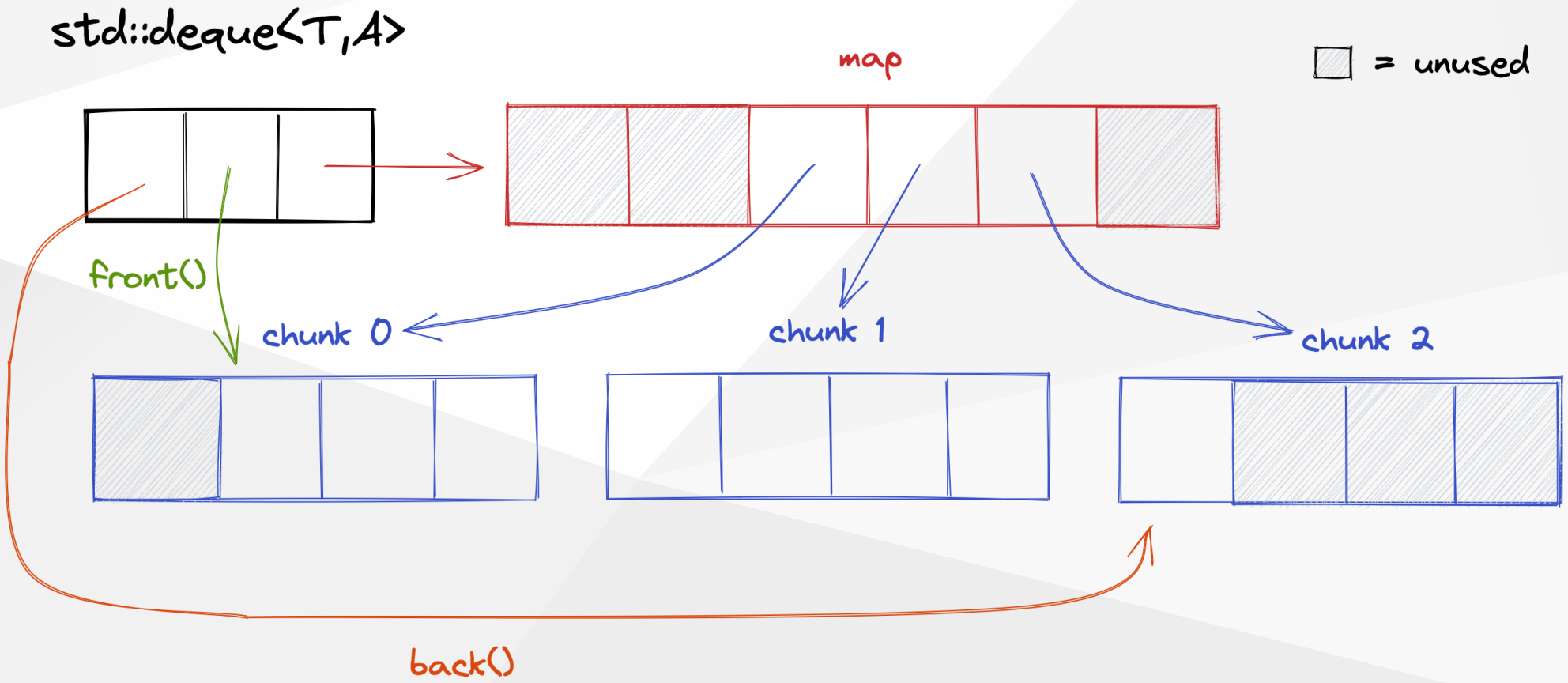
È definita nell'header `deque` come

```
template<
    class T,
    class Allocator = std::allocator<Key>
> class deque;
```

È una coda indicizzata in cui si può inserire sia in testa che in coda

Tra gli altri, fornisce i metodi `push_front(const T&)`, `push_back(const T&)`, `pop_front()`, `pop_back()`, `size()`, e `operator[](const size_t)`

Una Implementazione per `std::deque`



`std::deque`: esempio di utilizzo

```cpp
#include<deque>

int main() {
  std::deque<int> d{ 0, -5, 3};

  d.push_front(-3);
  d.push_back(7);

  std::cout  << "d.size(): " << d.size()    // stampa "d.size(): 5"
             << " d[0]: " << d[0]           // stampa " d[0]: -3"
             << " d.back(): " << d.back()   // stampa " d.back(): 7"
             << std::endl;

  return 0;
}
```

I Contenitori Associativi

- `std::set<T,CMP,A>`: insiemi di oggetti del tipo `T`
- `std::map<KEY,T,CMP,A>`: dizionari *á la* Python
- `std::multiset<T,CMP,A>`: multi-insiemi
- `std::multimap<KEY,T,CMP,A>`: multi-mappe

E le loro versioni *unordered*

- `std::unordered_set<T,HASH,KEY_EQUAL,A>`
- `std::unordered_map<KEY,T,HASH,KEY_EQUAL,,A>`

...

`std::set`

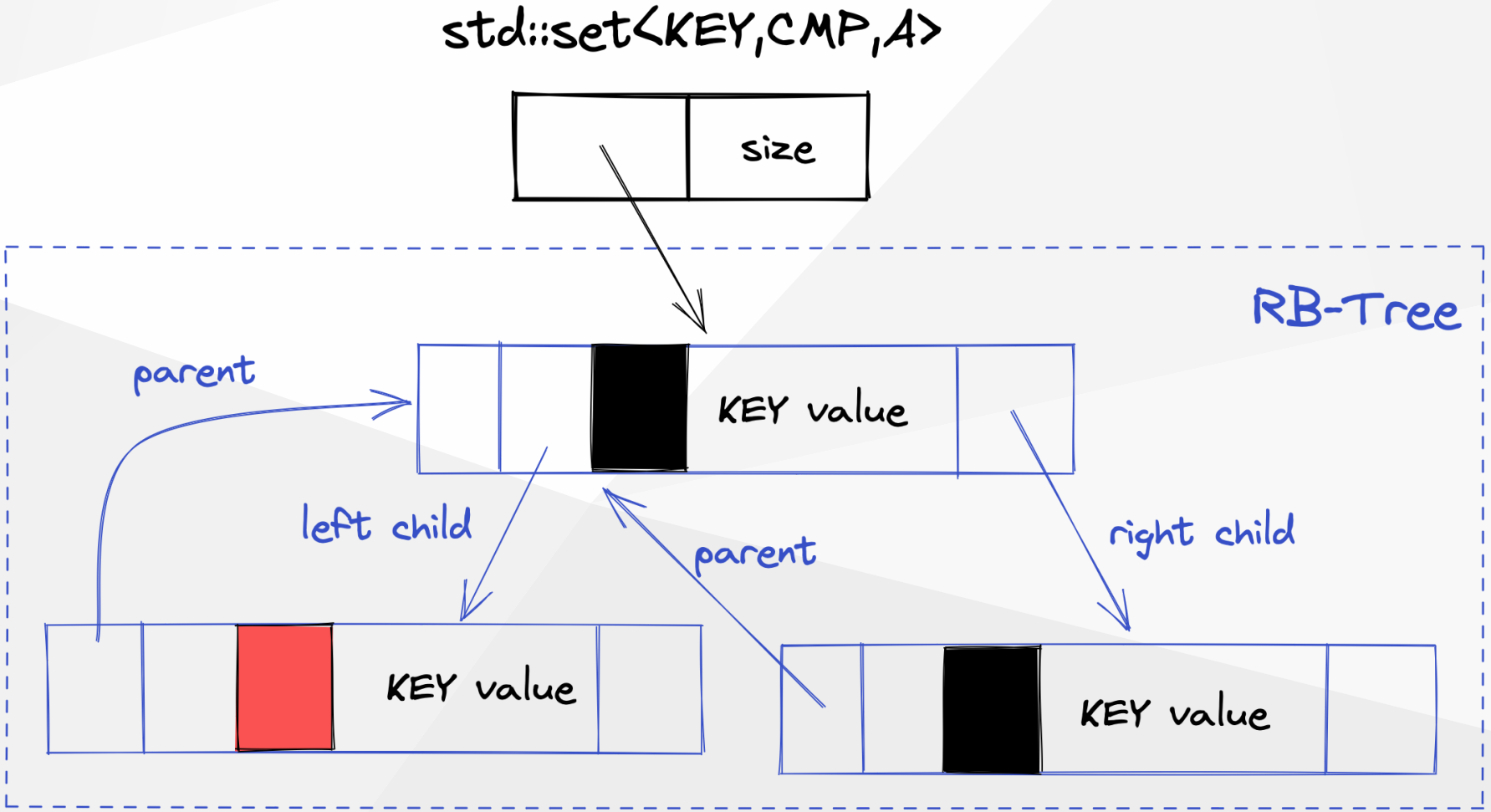
È definita nell'header `set` come

```
template<
    class Key,
    class Compare = std::less<Key>,
    class Allocator = std::allocator<Key>
> class set;
```

Rappresenta gli insiemi di oggetti

Tra gli altri, fornisce i metodi `insert(const T&)`, `count(const T&)`, `erase(const T&)`, `empty()`, e `size()`

Una Implementazione per `std::set`



`std::set`: esempio di utilizzo

```cpp
#include<set>

int main() {
  std::set<int> s{ 0, 5, 3};

  s.insert(-3);
  s.insert(5);    // 5 è già contenuto: non ha effetto su s
  s.erase(0);

  std::cout  << "s.size(): " << s.size()        // stampa "s.size(): 3"
             << " s.count(3): " << s.count(3)   // stampa " s.count(3): 1"
             << std::endl;

  return 0;
}
```

`std::map`

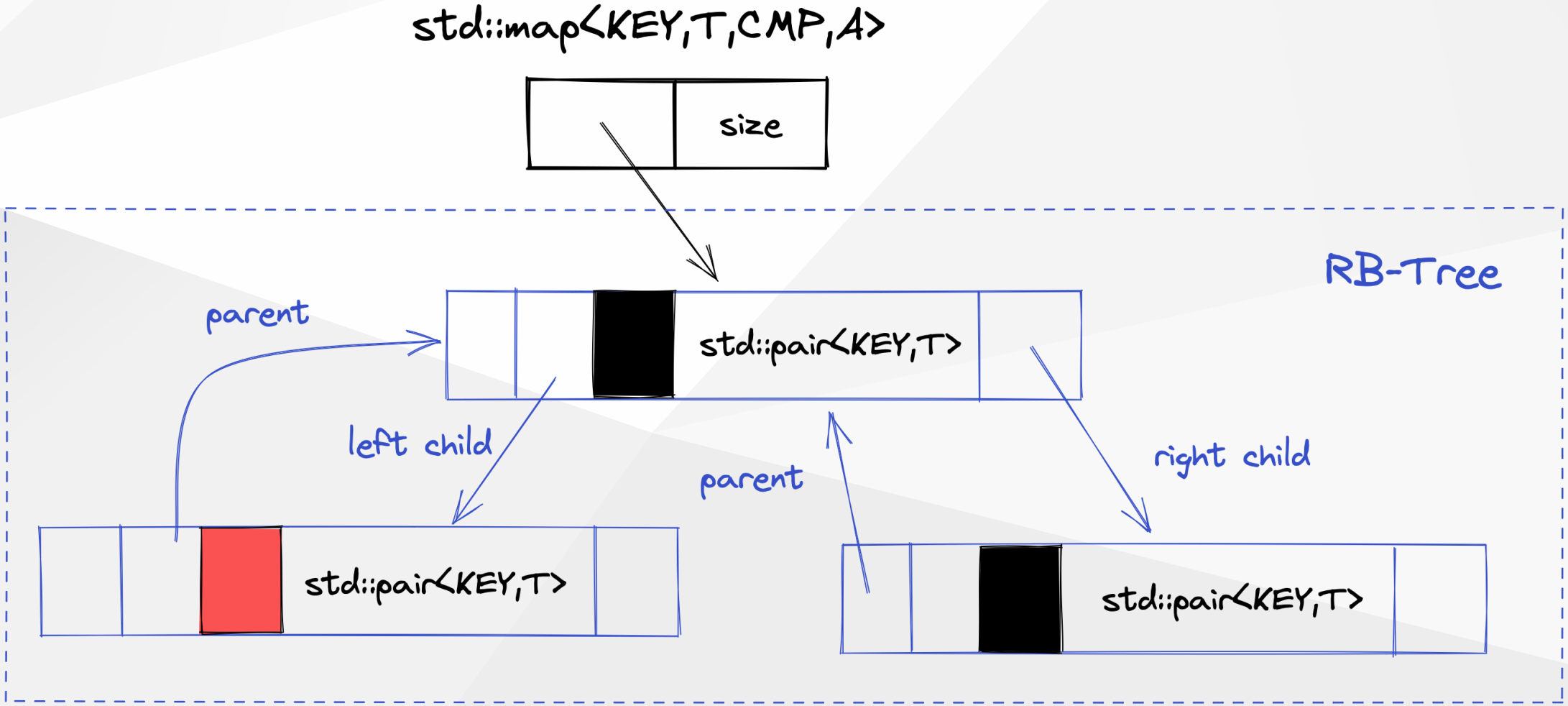
È definita nell'header `map` come

```
template<
    class Key, class T,
    class Compare = std::less<Key>,
    class Allocator = std::allocator<std::pair<const Key, T>>
> class map;
```

Rappresenta le mappe da `Key` a `T` (dizionari *á la* Python)

Tra gli altri, fornisce i metodi `insert(const std::pair<Key, T>&)`, `count(const Key&)`, `erase(const Key&)`, e `contains(const Key&)`

Una Implementazione per `std::map`



`std::map`: esempio di utilizzo

```cpp
#include<map>

int main() {
  std::map<std::string, std::vector<int>> m{{"first", {}},
                                            {"second", {1}}};

  m["first"].push_back(0);
  m.insert({ "second", {8,7} });  // non ha effetto
  m.erase("second");

  std::cout  << "m.size(): " << s.size()   // stampa "m.size(): 3"
             << " m.contains(\"first\"): "
             << m.contains("first")        // stampa " m.contains("first"): 1"
             << std::endl;

  return 0;
}
```

`std::multiset`

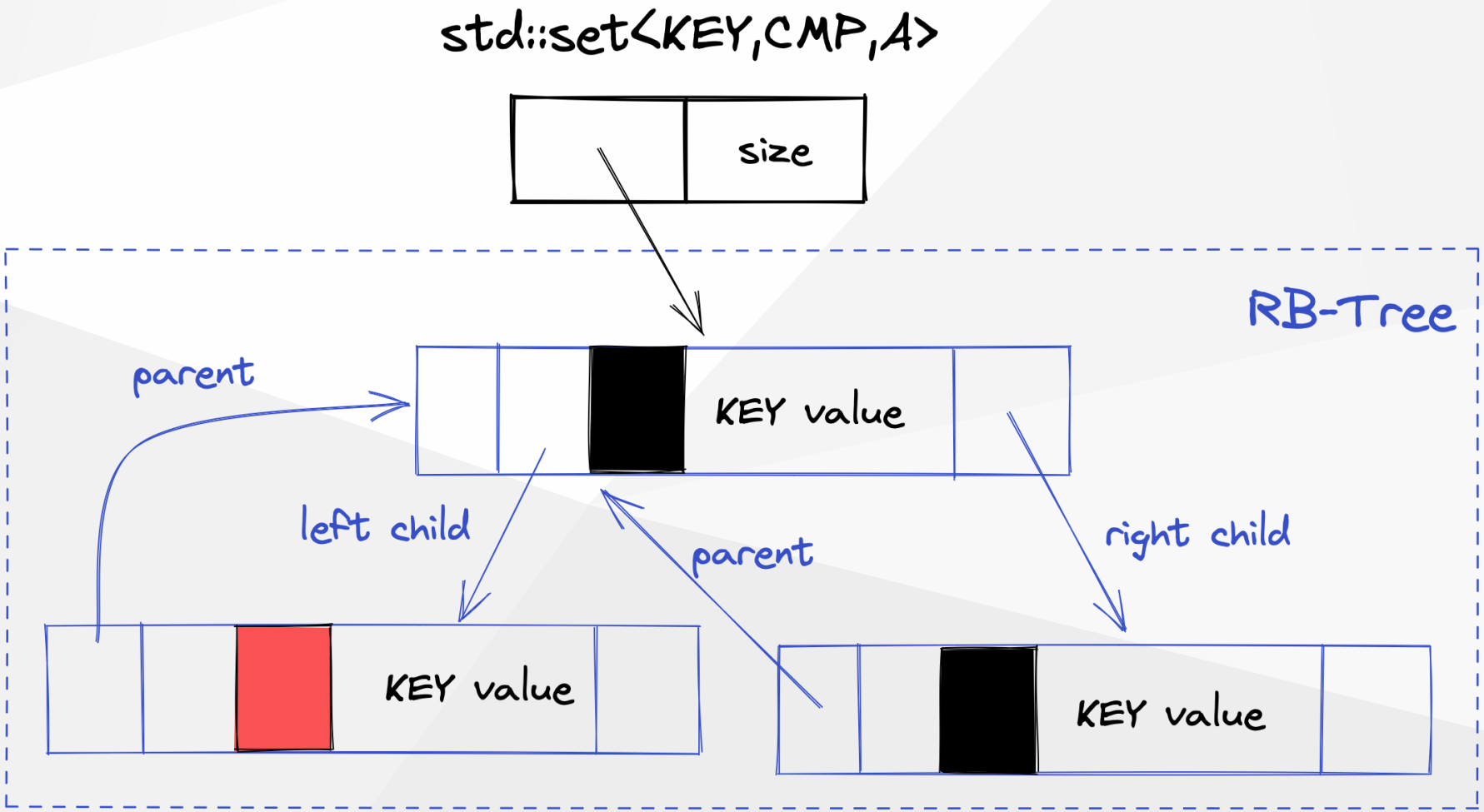
È definita nell'header `set` come

```
template<
    class Key,
    class Compare = std::less<Key>,
    class Allocator = std::allocator<Key>
> class multiset;
```

Rappresenta i multi-insiemi di oggetti

Tra gli altri, fornisce i metodi `insert(const T&)`, `count(const T&)`, `erase(const T&)`, `empty()`, e `size()`

Una Implementazione per `std::multiset`



`std::multiset`: esempio di utilizzo

```cpp
#include<set>

int main() {
  std::multiset<int> m{ 0, 5, 3};

  m.insert(-3);   // inserisco un nuovo valore
  m.insert(5);    // 5 è già contenuto, ma questo è un multiset
  m.erase(0);

  std::cout  << "m.size(): " << m.size()         // stampa "m.size(): 4"
             << " m.count(5): " << m.count(5)    // stampa " m.count(5): 2"
             << std::endl;

  return 0;
}
```

`std::multimap`

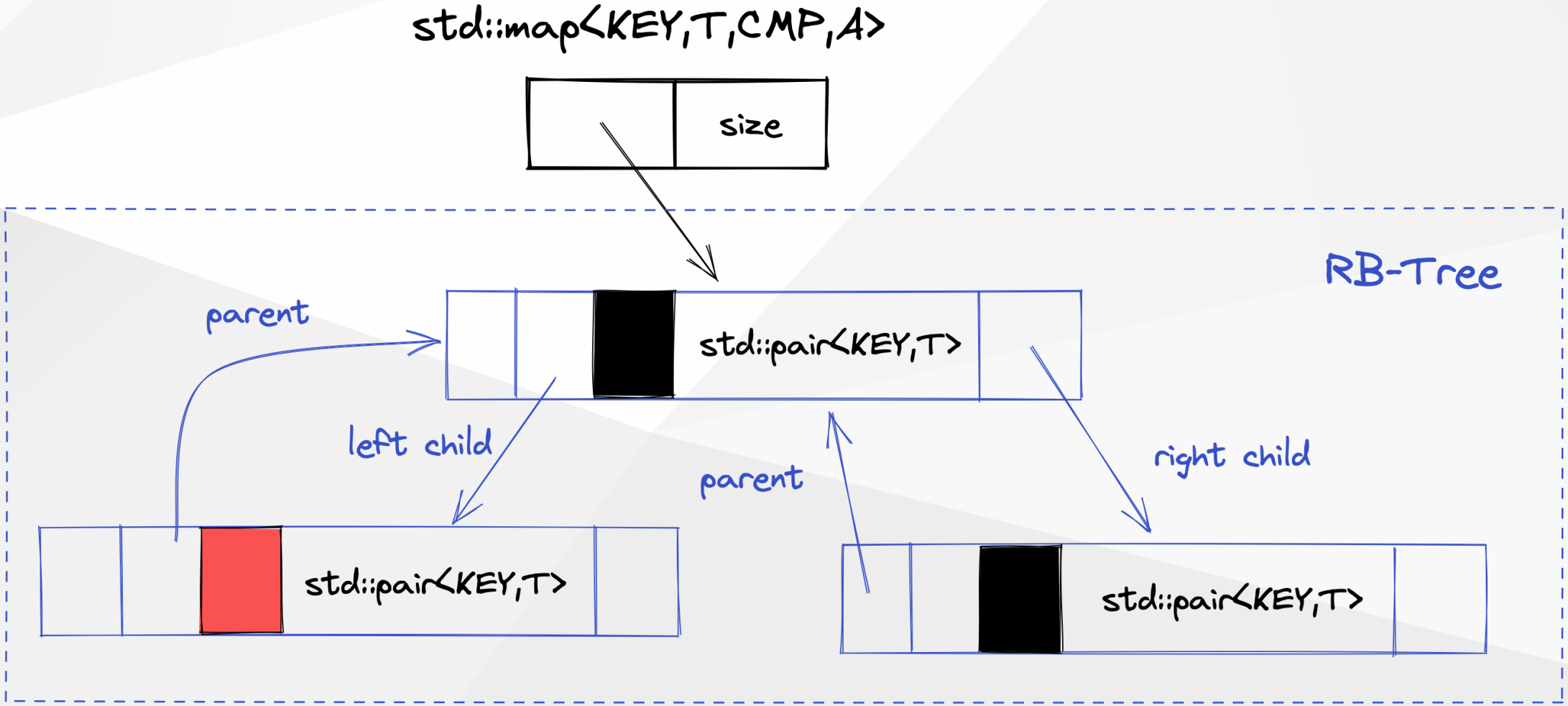
È definita nell'header `map` come

```
template<
    class Key, class T,
    class Compare = std::less<Key>,
    class Allocator = std::allocator<std::pair<const Key, T>>
> class multimap;
```

Rappresenta le multi-mappe da `Key` a `T`

Tra gli altri, fornisce i metodi `insert(const std::pair<Key, T>&)`, `count(const Key&)`, `erase(const Key&)`, e `contains(const Key&)`

Una Implementazione per `std::multimap`



`std::multimap`: esempio di utilizzo

```cpp
#include<map>

int main() {
  std::multimap<std::string, std::vector<int>> m{{"first", {}}, {"second", {1}},
                                                 {"third", {2,2}}};
  m.insert({ "second", {8,7} });
  m.erase("third");

  std::cout  << "s.size(): " << s.size()    // stampa "s.size(): 1"
             << " s.count(\"second\"): "
             << s.count("second")          // stampa " s.count("second"): 2"
             << std::endl;

  return 0;
}
```

`std::unordered_set`

È definita nell'header `unordered_set` come

```
template<
    class Key,
    class Hash = std::hash<Key>,
    class KeyEqual = std::equal_to<Key>,
    class Allocator = std::allocator<Key>
> class set;
```

È un'implementazione degli insiemi basata sulle hash

`unordered_map`, `unordered_multiset`, e `unordered_multimap` sono analoghi

Container Adaptor

Forniscono un interfaccia diversa per i contenitori

- `stack<T,A>`: Pile (`forward_list`?)
- `queue<T,A>`: Code (`list`?)
- `priority_queue<T,CMP,A>`: Code di priorità

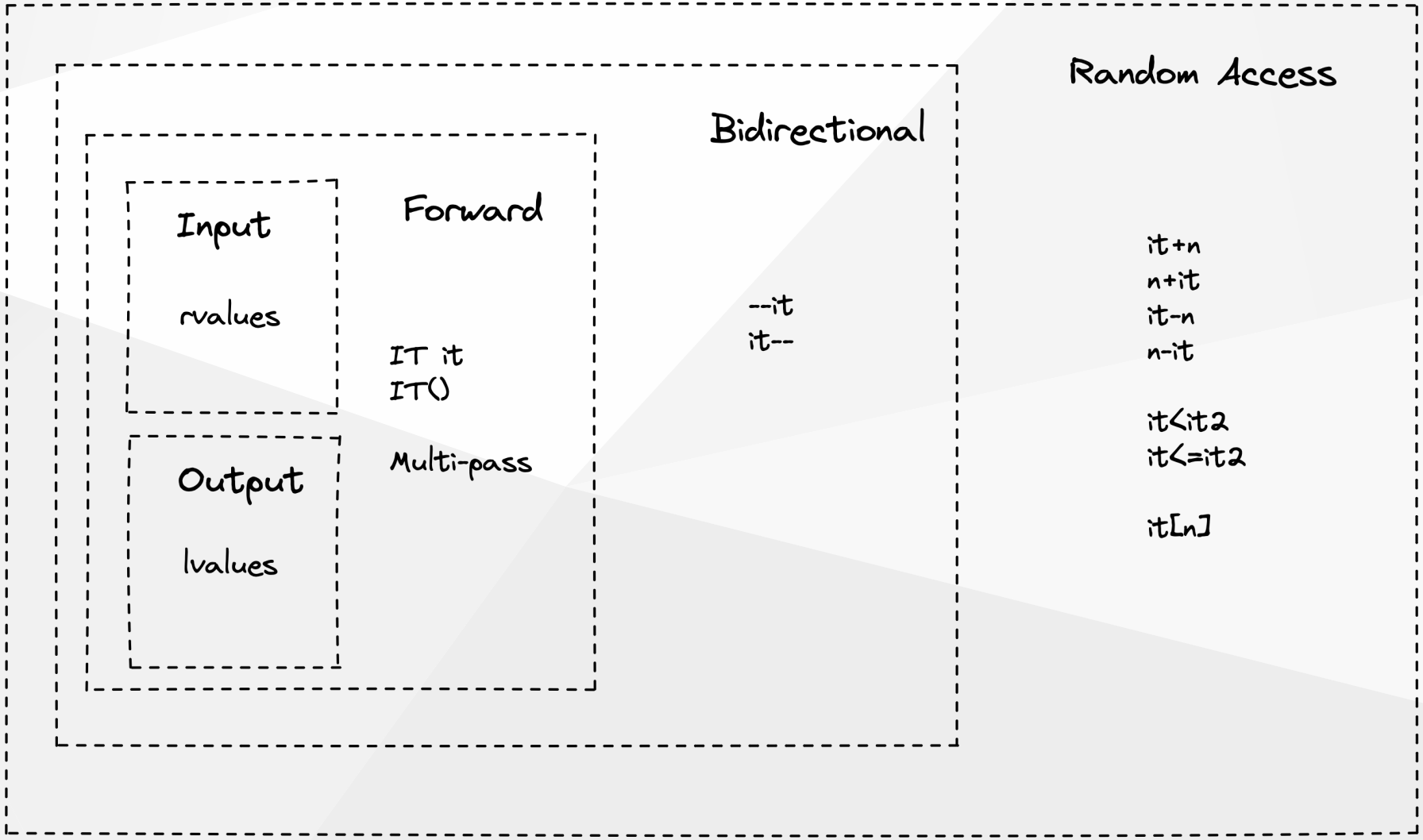
Gli Iteratori

Sono oggetti che puntano a valori in un contenitore o in una sequenza

Sono dotati almeno di:

- costruttore di copia
- l'assegnamento con semantica di copia
- incremento prefisso (`++a`)
- incremento postfisso (`a++`)
- operatore di dereferenziazione (`*a`)
- operatore freccia (`a->`)

Categorie di Iteratori



Gli Iteratori e i Contenitori

Ogni classe contenitore `C` è dotata di quattro sottoclassi:

- `C::iterator`: una classe iteratore per gli oggetti di `C`
- `C::const_iterator`: una classe iteratore costante per `C`
- `C::reverse_iterator`: un iteratore per scandire i valori negli oggeti di `C` ordine inverso
- `C::const_reverse_iterator`: come `reverse_iterator`, ma costante

Gli Iteratori e i Contenitori

I contenitori sono anche dotati dei metodi:

- `begin()` : restituisce un iteratore al "primo" valore del contenitore
- `end()` : restituisce l'iteratore successivo all'ultimo valore
- `rbegin()` : restituisce il primo valore di un iteratore rovesciato
- `rend()` : restituisce la fine di un iteratore rovesciato

Scandire un Vettore con gli Iteratori

```
std::vector<int> v={0, 1, 2, 3, 4};

for (std::vector<int>::iterator it = v.begin();
     it != v.end(); ++it) {
  *it *= 2;
}

for (std::vector<int>::const_iterator it = v.begin();
     it != v.end(); ++it) {
  std::cout << *it << std::endl;
}
```

Iteratori e for-"compatto"

```
std::vector<int> V={0, 1, 2, 3, 4};
std::vector<int> V2{V};

for (std::vector<int>::iterator it = V.begin();
     it != V.end(); ++it) {
  *it *= 2;
}

for (int& value: V) {  // questa sintassi è la forma contratta del precedente
                       //  "for". È possibile quando il contenitore è dotato
                       // dei metodi begin() e end()
  value *= 2;
}
```

Iteratori Predefiniti

- `reverse_iterator`: vista in ordine inverso una struttura
- `move_iterator`: dereferenzia l'*rvalue* prodotto
- `back_insert_iterator`: inserisce in coda
- `front_insert_iterator`: inserisce in testa
- `insert_iterator`: inserisce in una posizione indicata
- `istream_iterator`: legge i dati da un `istream`
- `ostream_iterator`: scrive i dati su un `ostream`
- `istreambuf_iterator`: legge i dati da un buffer di input
- `ostreambuf_iterator`: scrive i dati su un buffer di output

`std::move_iterator<ITER>` : Esempio di Utilizzo

```
#include <iostream>
#include <iterator>
#include <vector>
#include <algorithm>    // per std::copy

int main () {
  std::vector<double> v1(2);   // creo un array di 2 elementi
  std::vector<double> v2{-5.9, 8.2};

  typedef std::vector<double>::iterator VIter;
  std::copy ( std::move_iterator<VIter>(v2.begin()),
              std::move_iterator<VIter>(v2.end()),
              v1.begin() );
  return 0;
}
```

`std::back_inserter<CONTAINER>` : Esempio di Utilizzo

```cpp
#include <iostream>
#include <iterator>
#include <vector>
#include <algorithm>    // per std::copy

int main () {
  std::vector<double> v1{2.1, 3.0, -2.1};
  std::vector<double> v2{-5.9, 8.2};

  std::copy(v2.begin(), v2.end(), std::back_inserter(v1));

  return 0;
}
```

`std::insert_iterator<CONTAINER>` : Esempio

```cpp
#include <iterator>
#include <list>
#include <algorithm>    // per std::copy

int main () {
  std::list<int> l1{2, 3, -5, 7, 8};
  std::list<int> l2{4,5};

  std::list<int>::iterator l_it = l1.begin()+3;

  std::insert_iterator< std::list<int> > insert_it (l1,l_it);

  std::copy (l2.begin(),l2.end(),insert_it);

  return 0;
}
```

`std::ostream_iterator<...>`: Esempio di Utilizzo

```cpp
#include <iostream>
#include <iterator>
#include <vector>
#include <algorithm>    // per std::copy

int main () {
  std::vector<double> v1{2.1, 3.0, -2.1};

  std::ostream_iterator<int> out_it (std::cout,", ");
  std::copy ( v1.begin(), v1.end(), out_it );

  return 0;
}
```

`std::istream_iterator<...>`: Esempio di Utilizzo

```cpp
#include <iostream>
#include <iterator>
#include <vector>

int main() {
  std::vector<int> v;
  std::cout << "Inserisci quanti valori vuoi (0 per finire): ";

  std::istream_iterator<int> eos;              // fine dell'iteratore di input
  std::istream_iterator<int> in_it (std::cin);   // iteratore di input su std::cin

  while (in_it!=eos && *in_it != 0) {
    v.push_back(*(in_it++));
  }

  return 0;
}
```

Funzioni per gli Iteratori

STL fornisce anche una serie di funzioni per gli iteratori

- `advance(it, n)`: fa avanzare l'iteratore `it` di `n` passi
- `distance(it, it2)`: misura il numero di passi tra `it` e `it2`
- `begin(container)`: restituisce l'iteratore iniziale di `container`
- `end(container)`: restituisce l'iteratore iniziale di `container`
- `prev(it,n=1)`: restituisce l'iteratore ai valori precedenti
- `next(it,n=1)`: restituisce l'iteratore al valori successivi

I Funzionali di STL

STL introduce classi e funzioni per trattare le funzioni come oggetti

- `std::function` (C++11): tipo funzione
- `std::invoke` (C++17): invoca una funzione
- `std::identity` (C++20): una classe per la funzione identità
- `std::not_fn` (C++17): restituisce la negazione di funzione
- `std::bind` (C++11): consente valutazione parziali delle funzioni

`std::function`: Esempio di Utilizzo

```
#include<iostream>
#include<functional>  // per function

int somma(const int& a, const int& b) {
    return a+b;
}

int main() {
  std::function<int(const int&, const int&)> f = somma;

  std::cout << f(3, 4) << std::endl;

  return 0;
}
```

`std::bind`: Esempio di Utilizzo

```cpp
#include<functional>  // per bind e function (auto)

int somma(const int& a, const int& b) { return a+b; }

int main() {
  using namespace std::placeholders;  // _1, _2, _3... sono i futuri
                                      // parametri attuali

  auto f = std::bind(somma, _3, 2);   // la chiamata a f(a,b,c,...)
                                      // invocherà somma(c,2)

  std::cout << f(1,2,3,4)             // tutti i parametri tranne il terzo
            << std::endl;             // vengono scartati

  return 0;
}
```

`std::not_fn`: Esempio di Utilizzo

```cpp
#include<iostream>
#include<functional>  // per not_fn

bool uguali(const int& a, const int& b) {
    return a==b;
}

int main() {
  auto f = std::not_fn(uguali);

  std::cout << f(3,5) << std::endl;

  return 0;
}
```

Gli Algoritmi di STL

STL fornisce l'implementazione di molti algoritmi

- `all_of`, `any_of` e `none_of` per testare una condizione
- `for_each` per applicare una funzione
- `sort`, `stable_sort`, `partial_sort` per ordinare delle seguenze
- `push_heap`, `pop_heap`, `make_heap` e `sort_heap` per gestire una heap
- `min`, `max`, `minmax`
- `copy`, `move`, `swap`, `unique`, `rotate`, `remove_if`, ...

`std::any_of` : Esempio di Utilizzo

```cpp
#include <list>
#include <algorithm>     // per any_of

int main () {
  std::list<int> L = {0,5,-7};

  auto P = [](int i){ return i<0; };   // funzione lambda

  if (std::any_of(L.begin(),L.end(),P)) // richiede iteratori di input
    std::cout << "L contiene dei valori negativi";

  return 0;
}
```

`std::for_earch`: Esempio di Utilizzo

```
#include <list>
#include <algorithm>     // per for_each

int main () {
  std::list<int> L = {0,5,-7};

  auto F = [](int& i){ i = 2*i; };    // funzione lambda

  std::for_each(L.begin(),L.end(), F);  // richiede iteratori di input

  return 0;
}
```

`std::sort` : Esempio di Utilizzo

```
#include <list>
#include <algorithm>     // per sort

int main () {
  std::vector<int> V = {0,5,-7};

  std::sort(V.begin(), V.end());  // it richiede iteratori ad accesso casuale

  return 0;
}
```

`std::rotate`: Esempio di Utilizzo

```cpp
#include <list>
#include <algorithm>    // per rotate

int main () {
  std::list<int> L = {0,5,-7,6,4};

  auto n_begin = L.begin();    // "n_begin" punta la testa di L

  std::advance(n_begin, 3);    // "n_begin" ora punta il quarto elemento di L

  std::rotate(L.begin(), n_begin, L.end());  // la lista L viene ruotata in
                                             // modo che il primo elemento
                                             // sia quello puntato da "n_begin"
  return 0;
}
```