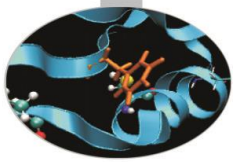
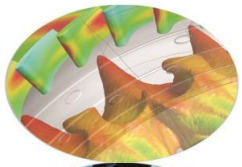
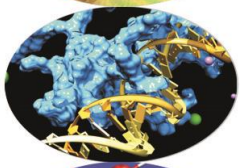
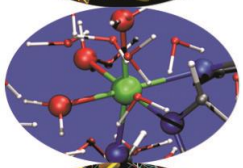
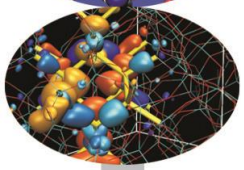
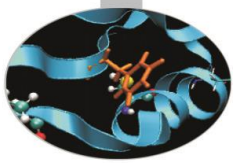


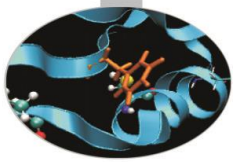
STL



Indice

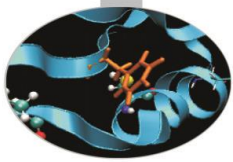
- Le componenti fondamentali
- I container
- Gli iteratori
- Gli algoritmi
- Il container sequenziale vector
- Vector e polimorfismo
- I container sequenziali deque e list
- I container associativi set, multiset, map e multimap
- Gli adattatori di container stack, queue



La Standard Template Library

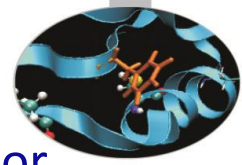
- E' una componente estremamente utile del linguaggio C++ ed appare come una libreria di ***strutture dati*** ed ***algoritmi*** che agiscono su tali strutture.
- Poiché fa uso dei *template*, tale libreria è del tutto *generica.*
- Le funzioni presenti nell'STL sono, generalmente, *inline*: questa caratteristica garantisce un certo grado di efficienza ma, come sappiamo, *aumenta* le dimensioni del codice eseguibile.
- L'uso dei template può richiedere un *considerevole* tempo di *compilazione*
- Tutte le funzionalità dell'STL sono contenute all'interno del *namespace std*.

STL: le componenti fondamentali



- L'STL è costituita da tre componenti fondamentali: i **container**, gli **iteratori** e gli **algoritmi**.
- I **container** sono dei *template* di classi, ovvero strutture di dati che possono assumere qualsiasi tipo, con proprie funzioni membro. La memoria occupata dai container è allocata in maniera automatica dai cosiddetti *allocatori* dell'STL: non si rende più necessario l'utilizzo di new e delete, né la specifica di una dimensione tramite una costante numerica.
- Gli **iteratori** sono classi (contenute nei container) i cui oggetti servono per *accedere* agli elementi dei container stessi. In un certo senso possiamo dire che gli iteratori stanno ai container come i puntatori agli array.
- Gli **algoritmi** sono funzioni che servono per manipolare i container. Essi sono separati dai container (ovvero *non sono metodi*) ed accedono agli elementi dei container indirettamente, tramite gli iteratori che ricevono come argomenti.

I container



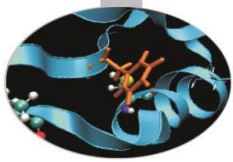
- I container si dividono in due gruppi: *container di prima classe* e *adattatori di container*. I container di prima classe mostrano un maggior numero di funzioni membro e supportano gli iteratori che sono, invece, preclusi agli adattatori di container.
- Fra i *container di prima classe* si distinguono, poi, i *container sequenziali* ed i *container associativi.*
- Esistono anche i *"quasi container"*, come le string, che mostrano funzionalità simili a quelle dei container di prima classe, ma non ne condividono tutte le caratteristiche.
- La dichiarazione di un container segue la sintassi:

  tipo_container<tipo_dato> nome_container (lista_argomenti);

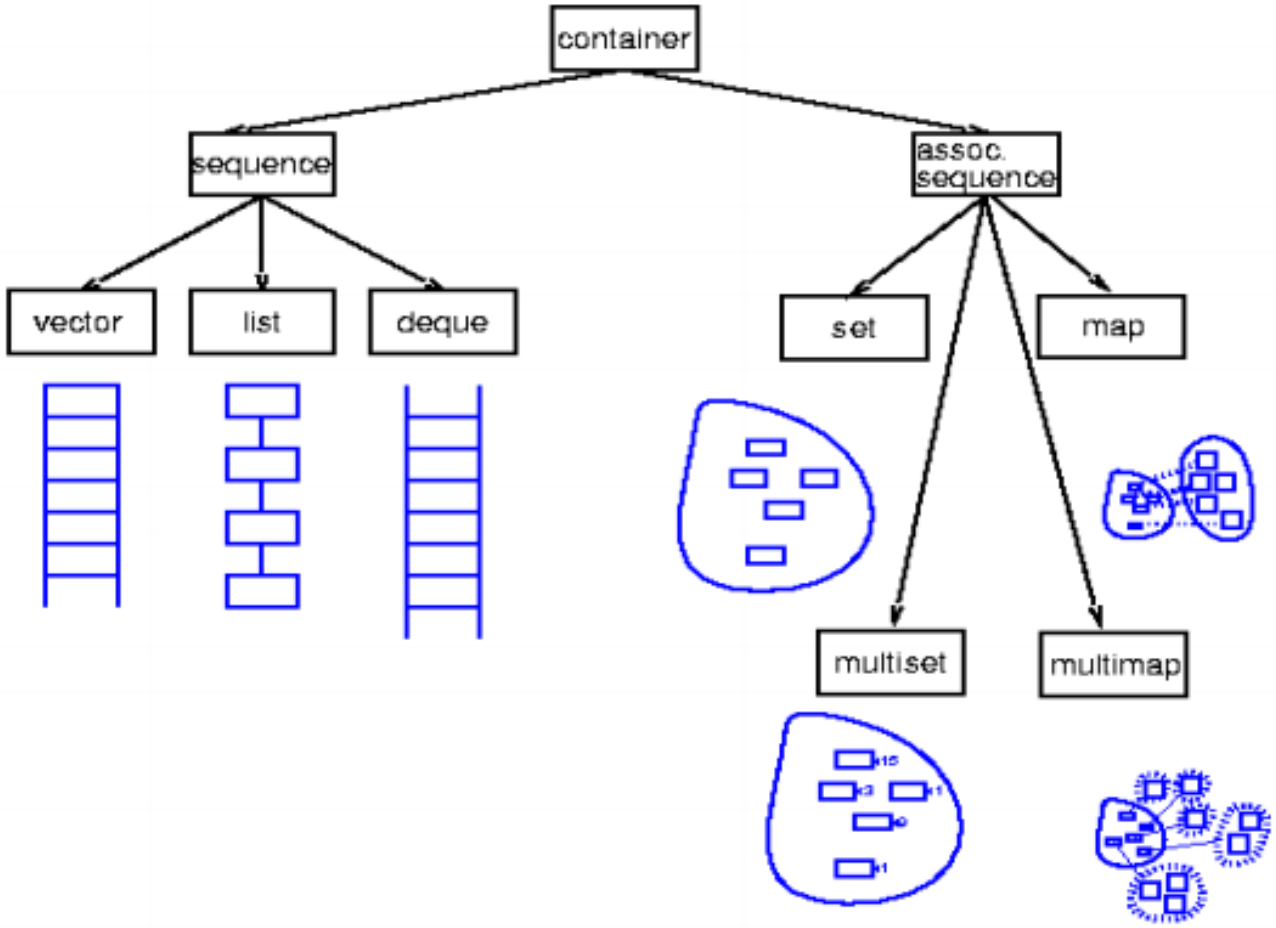
- ove la *lista_argomenti* comprende i parametri da inviare al *costruttore* del container.

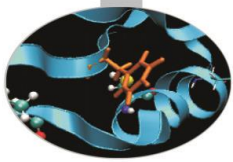
Esempi:

- vector<int> v1(6,2); // il container vector v è dato da 6 elementi tutti uguali a 2
- double arr[3]={2.4, 1.9, 7.5}; // questo è un vettore
- vector<double> v2(arr, arr+3); // il costruttore riceve due indirizzi di memoria



Container


container
sequence
assoc. sequence
vector
list
deque
set
map
multiset
multimap



Container

container sequenziali

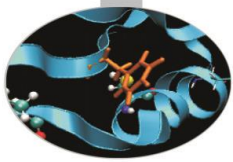
vector	inserimenti/eliminazioni rapidi in coda; accesso diretto a qualsiasi elemento
list	lista a doppio concatenamento; inserimenti ed eliminazioni rapidi ovunque
deque	inserimenti/eliminazioni rapidi in testa o in coda; accesso diretto a qualsiasi elemento

container associativi

set	ricerca rapida; non sono consentiti duplicati
multiset	ricerca rapida; sono consentiti duplicati
map	mapping uno-a-uno; non sono consentiti duplicati; ricerca rapida di una chiave
multimap	mapping uno-a-uno; sono consentiti duplicati; ricerca rapida di una chiave

adattatori di container

stack	l'ultimo inserito è il primo estratto: last-in-first-out (LIFO)
queue	il primo inserito è il primo estratto: first-in-first-out (FIFO)
priority_queue	l'elemento di priorità più alta è sempre il primo elemento estratto



Gli Iteratori

Gli iteratori servono per accedere agli elementi dei container di prima classe. Essi sono implementati in modo appropriato per ogni tipo di container che li supporta: alcuni *metodi* degli iteratori dipendono, dunque, strettamente dal container di cui fanno parte, altre operazioni, invece, hanno validità generale. Tra queste ce ne sono alcune, per esempio, tipiche dei puntatori, ovvero l'incremento (++), la dereferenziazione (*) ecc.

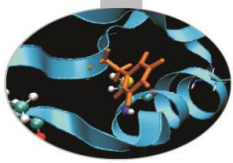
Un iteratore viene dichiarato come:

*tipo_container<tipo_dato>**::**tipo_iteratore nome_iteratore;*

La STL mette a disposizione quattro *tipi di iteratore* predefiniti, ognuno con una propria funzionalità ed un particolare comportamento relativamente all'operatore di incremento **++**:

tipo di iteratore	***direzione di ++***	***funzionalità***
iterator	avanti	lettura/scrittura
const_iterator	avanti	lettura
reverse_iterator	indietro	lettura/scrittura
const_reverse_iterator	indietro	lettura

Gli Iteratori



Esempio:

vector<int>::reverse_iterator p1; // dichiarazione di un iteratore p1 che itera
// all'indietro un container di tipo vector<int>

Esistono iteratori preposti alle operazioni di input ed output e non richiedono, in fase di dichiarazione, di essere associati ad alcun container:

tipo_iteratore_I/O<tipo_dato> nome_iteratore (lista_argomenti);

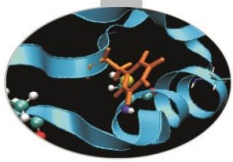
Gli iteratori abbinati, rispettivamente, alle funzionalità di input ed output sono: **istream_iterator** e **ostream_ iterator**.

Esempio:

```
istream_iterator<int> read_int(cin);
```
// legge valori interi da standard input attraverso cin

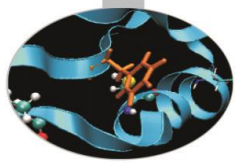
```
ostream_iterator<double> print_double(cout, " ");
```
// scrive valori double su standard output attraverso cout; i valori sono separati da uno spazio bianco (" ")

Gli Iteratori



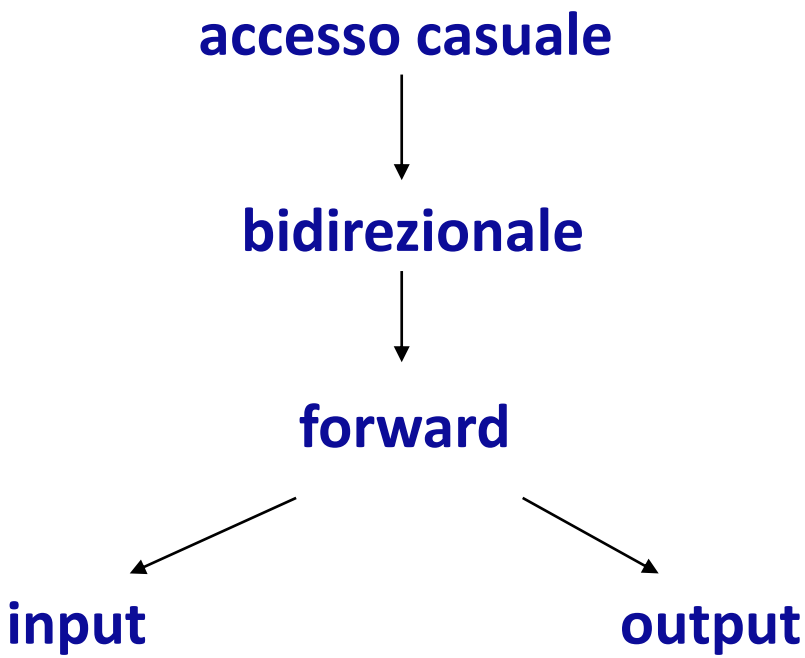
- Gli iteratori sono, inoltre, divisi in *categorie* legate, essenzialmente, alle modalità con cui iterano i container ed accedono ai loro elementi. La categoria di iteratore supportata da un container determina gli algoritmi che possono agire sul container stesso. La STL mette a disposizione cinque categorie:

categoria	**azioni**
input	Permette di leggere un elemento da un contenitore
output	Permette di scrivere un elemento in un contenitore
	Gli iteratori di input e output possono muoversi solo in avanti e di un solo elemento alla volta. Essi non possono essere utilizzati per attraversare più di una volta lo stesso container
forward	Combina le caratteristiche degli iteratori di input e output e in più permette di attraversare più volte lo stesso contenitore.
bidirezionale	Combina le caratteristiche di un forward iterator con la possibilità di muoversi all'indietro.
accesso casuale	Combina le caratteristiche di un bidirectional iterator con la possibilità di accedere direttamente a qualunque elemento del contenitore

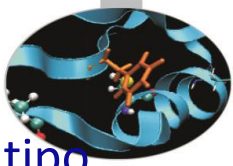


Gli Iteratori

Da quanto detto è chiaro che le cinque categorie di iteratori seguono una precisa gerarchia:

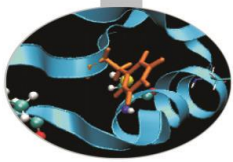


Gli Iteratori



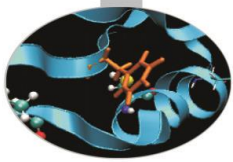
La seguente tabella mostra le categorie di iteratori supportate da ciascun tipo di container

container	**categoria di iteratore**
container sequenziali	
vector	accesso casuale
deque	accesso casuale
list	bidirezionale
container associativi	
set	bidirezionale
multiset	bidirezionale
map	bidirezionale
multimap	bidirezionale
adattatori di container	
stack	nessuna
queue	nessuna
priority_queue	nessuna



Iteratori

Operazioni sull'iteratore	Descrizione
Tutti gli iteratori	
`++p`	preincrementa un iteratore
`p++`	postincrementa un iteratore
Iteratori di input	
`*p`	dereferenzia un iteratore per utilizzare il risultato come rvalue
`p = p1`	assegna un iteratore a un altro
`p == p1`	verifica se due iteratori sono uguali
`p != p1`	verifica se due iteratori sono diversi
Iteratori di output	
`*p`	dereferenzia un iteratore per utilizzare il risultato come lvalue
`p = p1`	assegna un iteratore a un altro
Iteratori forward	gli iteratori forward hanno tutte le funzionalità degli iteratori di input e di output
Iteratori bidirezionali	
`--p`	predecrementa un iteratore
`p--`	postdecrementa un iteratore
Iteratori ad accesso casuale	
`p += i`	incrementa l'iteratore `p` di `i` posizioni
`p -= i`	decrementa l'iteratore `p` di `i` posizioni
`p + i`	dà come risultato un iteratore posizionato in `p` incrementato di `i` posizioni
`p - i`	dà come risultato un iteratore posizionato in `p` decrementato di `i` posizioni
`p[ i ]`	restituisce un riferimento all'elemento che si scosta da `p` di `i` posizioni
`p < p1`	restituisce true se l'iteratore `p` è minore dell'iteratore `p1` (l'iteratore `p` si trova prima di `p1` nel container); altrimenti false
`p <= p1`	restituisce true se l'iteratore `p` è minore o uguale all'iteratore `p1` (l'iteratore `p` si trova prima o alla stessa posizione di `p1` nel container); altrimenti false
`p > p1`	restituisce true se l'iteratore `p` è maggiore dell'iteratore `p1` (l'iteratore `p` si trova dopo `p1` nel container); altrimenti false
`p >= p1`	restituisce true se l'iteratore `p` è maggiore o uguale all'iteratore `p1` (l'iteratore `p` si trova alla stessa posizione di `p1` o dopo di esso nel container); altrimenti false



Gli Algoritmi

- Gli algoritmi forniti dalla STL sono utili strumenti per la manipolazione dei container. Essi *non* sono funzioni membro dei container ed operano sui loro elementi in maniera indiretta tramite gli iteratori, che ricevono come argomento. Un container può essere modificato da un algoritmo se e solo se supporta gli iteratori di cui fa uso l'algoritmo stesso.

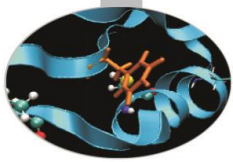
- La chiamata ad un algoritmo all'interno di un programma appare come:

```
nome_algoritmo( lista_argomenti );
```

- ove la *lista_argomenti* contiene iteratori e dati di varia natura (variabili, nomi di funzioni ecc.).

- Per far uso di algoritmi è necessario includere nel programma l'istruzione per il preprocessori

```
#include<algorithm>
```



Gli Algoritmi

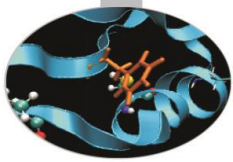
- Possiamo dire che esistono tre famiglie di algoritmi: algoritmi che *modificano* il contenuto dei container (es.: fill(), replace(), swap()), algoritmi che *non modificano* il contenuto dei container (es.: find(), search()) ed algoritmi *numerici* (es.: inner_product(), partial_sum()).

Esempio:

- replace**(** vec.begin(),vec.end(), 32, 20 **)**;

/* l'algoritmo replace sostituisce ogni occorrenza del valore 32 con il valore 20 all'interno del container vec, di tipo vector ad esempio. I primi due argomenti sono iteratori restituiti da funzioni membro di vector. */

Il container sequenziale vector

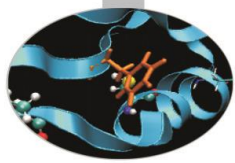


- Il container **vector** è una struttura di dati che occupano locazioni di memoria contigue e rappresenta un *miglioramento* del tipo di dati *array.*
- Come già accennato in precedenza, quando viene dichiarato un oggetto di tipo vector *non* è necessario specificarne la dimensione. La *memoria* occupata dal vector è allocata *dinamicamente*, in maniera *automatica*, dal compilatore.
- A differenza degli array, inoltre, i vector possono essere *assegnati tra di loro* grazie al costruttore di copia.
- I vector supportano iteratori ad accesso casuale, dunque possono essere manipolati da *tutti* gli algoritmi della STL.
- L'utilizzo di container vector richiede l'inserimento della direttiva

```
#include<vector>
```

all'interno del programma.

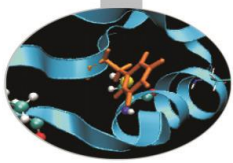
Esempio



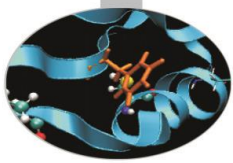
esempio1: dichiarazione di alcuni container vector, stampa del loro contenuto, l'algoritmo copy, i metodi begin, end ed empty.

```
#include<iostream.h>

#include<vector>

#include<algorithm>

int main(){

  vector<int> vi_one(5);     // il vector vi_one è composto
                            //da 5 elementi nulli per default

  vector<double> v_dbl(5,20.32);          //v_dbl contiene 5
                             //elementi tutti uguali a 20.32

  const int dim=5;

  int arr_d[dim]={1,2,3,4,5};

  vector<int> vi_two(arr_d, arr_d+dim);  // arr_d è
      //copiato in vi_two tramite passaggio di indirizzi

  ostream_iterator<int> out_int (cout, " ");  // iteratore
                                      //di tipo ostream per int
```

Esempio



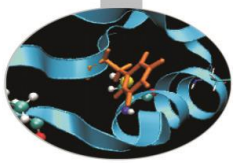
```
ostream_iterator<double> out_dbl (cout, " ");  //iteratore di
                         tipo ostream per double

cout << "Vector v_dbl contains: ";
  copy(v_dbl.begin(), v_dbl.end(), out_dbl);  // algoritmo copy() //per la
   stampa su standard output; fa uso dei metodi della classe vector
   begin() ed end()
  cout << endl;

  cout << "Vector vi_two contains: “;
 copy(vi_two.begin(), vi_two.end(), out_int);
 cout << endl;

 cout << "Is vi_one empty?" ;
 if(vi_one.empty()==1)          // metodo empty() della classe
                            vector
    cout << "true" << endl;
 else{
    cout << "false" << endl;
    cout << "Vector vi_one contains: “;
    copy(vi_one.begin(), vi_one.end(), out_int);
    cout << endl;}
 return 0;}
```



Esempio

L'output del programma è il seguente:

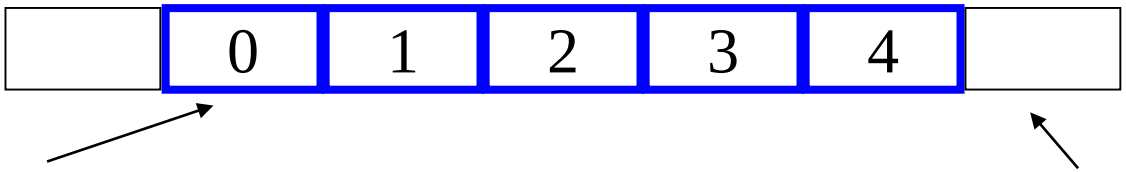
```
Vector v_dbl contains: 20.32 20.32 20.32 20.32 20.32
Vector vi_two contains: 1 2 3 4 5
Is vi_one empty? false
Vector vi_one contains: 0 0 0 0 0
```



Commenti

Alcuni commenti:

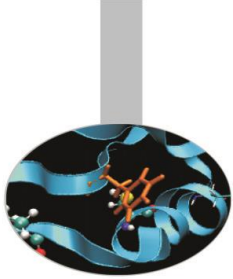
- **vector<**int**>** vi_two(arr_d, arr_d+dim); dice al compilatore di copiare il contenuto delle locazioni di memoria, comprese fra gli indirizzi *arr_d* e *arr_d+dim* escluso, all'interno del vector *vi_two*;
- **copy(**v_dbl.**begin()**, v_dbl.**end()**, out_dbl**)**; dice al compilatore di copiare nella posizione di memoria specificata dall'iteratore di output *out_dbl*, il contenuto del container *v_dbl* presente tra le celle di memoria puntate dagli iteratori restituiti da *begin()* ed *end()* (quest'ultima esclusa).
- Infatti:



- v_dbl.**begin()** v_dbl.**end()**

- if(vi_one.**empty()**==1) : il metodo *empty()* restituisce "true" se e solo se il container al quale si riferisce non contiene alcun valore, compreso lo zero. Ciò si verifica quando il container è stato dichiarato senza specificarne nemmeno la dimensione.

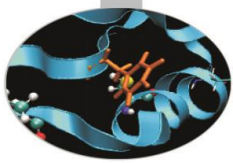
Esempio



esempio2: assegnamento di vector, l'algoritmo replace, i metodi reserve, capacity, size, back, push_back, at, operator[], pop_back, resize e swap.

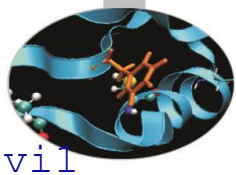
```
#include<iostream.h>
#include<vector>
#include<algorithm>
  int main() {
  vector<int> vi1;
  const int dim=5;
  int arr_i[dim]={1,5,9,14,19};
  vector<int> vi2(arr_i, arr_i+dim);

  vi1.reserve(10);     // metodo reserve(): prealloca memoria per il
                          vector vi1
  cout << "The capacity of vi1 is: " << vi1.capacity() << endl;
   // metodo capacity: restituisce la memoria allocata per il vector
   //vi1
  cout << "The size of vi1 is: " << vi1.size() << endl;
   // metodo size: ritorna l'effettiva dimensione del vector vi1
  cout << "The last element of vi2 is: " << vi2.back() << endl;
   // metodo back: dà l'ultimo elemento del vector vi2
  ostream_iterator<int> out_int (cout, " ");
```

Esempio



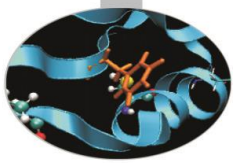
```
cout << "vi2 is: ";
copy(vi2.begin(),vi2.end(),out_int);
cout << endl;

 cout << "Is vi1 empty? ";
 if(vi1.empty()==1)
   cout << "true" << endl;
 else{
   cout << "false" << endl;
   cout << "Vector vi1 contains: ";
   copy(vi1.begin(), vi1.end(), out_int);
   cout << endl; }

 cout << "The dimension of vi2 is: " << vi2.size() << endl;

 cout << "Doing the assignment vi1=vi2" << endl;
 vi1 = vi2;              // assegnamento del vector vi2 al vector
                            vi1
 cout << "The size of vector vi1 is: " << vi1.size() << endl;
 cout << "Vector vi1 contains: ";
 copy(vi1.begin(), vi1.end(), out_int);
 cout << endl;
```

Esempio



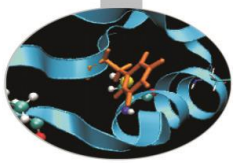
```
// segue
  vi1.push_back(20);
     // metodo push_back: aggiunge un elemento in coda al vector vi1
  vi1.push_back(32);
  vi1.at(2)=14;
     //metodo at: accede all'elemento del vector vi1 in
       posizione 2 e gli assegna il valore 2
  vi1[6]=4;       // metodo operator[ ]: funziona come at
  vi1.push_back(21);

  cout << "Three elements have been inserted" << endl;
  cout << "The size of vector vi1 is: " << vi1.size() << endl;
  cout << "Vector vi1 contains: ";
  copy(vi1.begin(), vi1.end(), out_int);
  cout << endl;

  replace(vi1.begin()+1, vi1.end(), 14, 75); //algoritmo replace
  cout << "After replacing 14 with 75, vector vi1 contains: ";
  copy(vi1.begin(), vi1.end(), out_int);
  cout << endl;
  // continua
```

Esempio

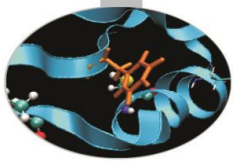


```
// segue
  vi1.pop_back();  // metodo pop_back(): dealloca l'ultimo
   elemento del vector vi1
  vi1.pop_back();
  cout << "The last two elements have been cancelled; vi1 is: ";
  copy(vi1.begin(), vi1.end(), out_int);
  cout << endl;

  if(vi1.size > 4)
     vi1.resize(4); // metodo resize: modifica la dimensione del
                    vector vi1
  cout << "Now vi1 has been resized to " << vi1.size() << endl;
  cout << "After swapping vi1 with vi2, vi1 becomes: ";
  vi1.swap(vi2);  // metodo swap: scambia gli elementi di vi1
                    con quelli di vi2
  copy(vi1.begin(), vi1.end(), out_int);
  cout << endl;
  cout << "and vi2 becomes: ";
  copy(vi2.begin(), vi2.end(), out_int);
  cout << endl;

  return 0; }
```

Esempio



```
Abbiamo, ora, come output:
The capacity of vi1 is: 10
The size of vi1 is: 0
The last element of vi2 is: 19
vi2 is: 1 5 9 14 19
Is vi1 empty? true
The dimension of vi2 is: 5
Doing the assignment vi1=vi2
The size of vector vi1 is: 5
Vector vi1 contains: 1 5 9 14 19
Three elements have been inserted
The size of vector vi1 is: 8
Vector vi1 contains: 1 5 14 14 19 20 4 21
After replacing 14 with 75, vector vi1 contains: 1 5 75 75 19 20 4 21
The last two elements have been cancelled; vi1 is: 1 5 75 75 19 20
Now vi1 has been resized to 4
After swapping vi1 with vi2, vi1 becomes: 1 5 9 14 19
and vi2 becomes: 1 5 75 75
```

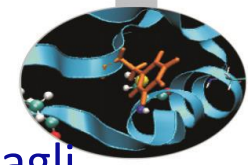


Commenti

Alcuni commenti:

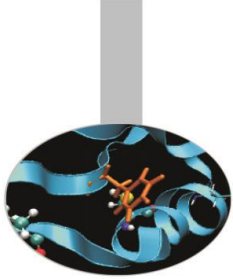
- vi1.**reserve(**10**)**; il metodo *reserve()* alloca 10 locazioni di memoria destinate al vector *vi1*. Può essere utile preallocare memoria perché l'uso dei metodi push_back e pop_back può allungare i tempi di compilazione e rallentare l'esecuzione del programma;
- vi1.**capacity()** : il metodo *capacity()* restituisce la memoria allocata per *vi1* con il metodo *reserve()*;
- vi2.**size()** : il metodo *size()* restituisce la dimensione corrente del vector *vi2* definita nella dichiarazione del vector stesso;
- vi2.**back()** : il metodo *back()* restituisce un reference all'ultimo elemento del vector vi2;
- **vi1= vi2**; operazione di *assegnamento*. E' permessa grazie al costruttore di copia presente nella classe vector.
- vi1.**push_back(**20**)**; il metodo *push_back()* aggiunge una nuova locazione di memoria in coda al vector *vi1* e vi salva il valore intero 20. La dimensione di *vi1* aumenta passando da 5 a 6;

Commenti



- vi1.at(2)=14; e vi1[6]=4; sono due istruzioni equivalenti per accedere agli elementi del vector *vi1*. La prima si avvale del metodo *at* e, rimpiazza l'intero 9 con 14; la seconda utilizza un altro metodo, *operator[ ]*, e scrive 4 nella cella di memoria che prima conteneva 32;
- replace(vi1.begin()+1, vi1.end(), 14, 75); l'algoritmo *replace* sostituisce il valore 14 con il valore 75 in tutte le celle di memoria del vector *vi1* comprese tra gli iteratori restituiti da *begin()+1* ed *end()* (quest'ultima, al solito, esclusa);
- vi1.pop_back(); il metodo *pop_back()* dealloca l'ultima locazione in coda al vector *vi1*, cancellandone il contenuto, ovvero l'intero 21. La dimensione di *vi1* passa, automaticamente, da 8 a 7;
- vi1.resize(4); il metodo *resize()* ridimensiona il vector *vi1* da 6 a 4;
- vi1.swap(vi2); il metodo *swap()* scambia gli elementi del vector *vi1* con quelli del vector *vi2.* Per svolgere questa operazione non è necessario che i due vector abbiano la stessa dimensione: la memoria destinata a loro è allocata automaticamente dal compilatore.

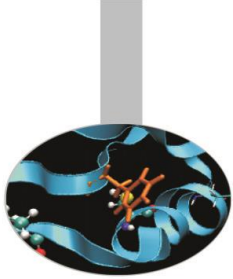
Esempio



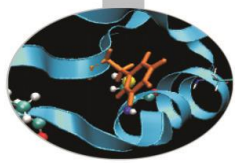
Esempio3: uso degli iteratori ed i metodi rbegin, rend, insert ed erase.

```
#include <iostream>
#include <iterator>
#include <vector>
#include <algorithm>
using namespace std;
  int main(){

  const int dim=5;
  int arr[dim]={1,2,3,4,5};
  vector<int> vi(arr, arr+5);

  ostream_iterator<int> out_int (cout, " ");

  cout << "Vector vi contains: ";
  copy(vi.begin(), vi.end(), out_int);
  cout << endl;
  vector<int>::reverse_iterator ptr_r;  // dichiarazione dell'iteratore
                                           reverse ptr_r
  for(ptr_r = vi.rbegin(); ptr_r != vi.rend(); ptr_r++)  // metodi
                                           reverse rbegin() e rend()
    *ptr_r = *ptr_r+6;   // operatore di dereferenziazione * applicato
                         ad un iteratore
  // continua
```

Esempio



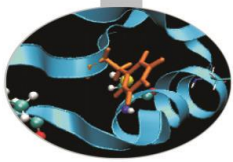
```
cout << "Now vector vi contains: ";
vector<int>::const_iterator ptr_c; // dichiarazione dell'iterator const

for(ptr_c = vi.begin(); ptr_c != vi.end(); ptr_c++)
  cout << *ptr_c << " ";
cout << endl;

vi.insert(vi.begin()+2, arr, arr+dim); // metodo insert(), per porre
                                       nuovi elementi dentro un
                                       vector
cout << "After inserting arr into vi, we have: ";
copy(vi.begin(), vi.end(), out_int);
cout << endl;

cout << "After erasing the first three elements of vi, we have: ";
vi.erase(vi.begin(), vi.begin()+3); // metodo erase(), per cancellare
                                    elementi di un vector
copy(vi.begin(), vi.end(), out_int);
cout << endl;
return 0;}
```



Esempio

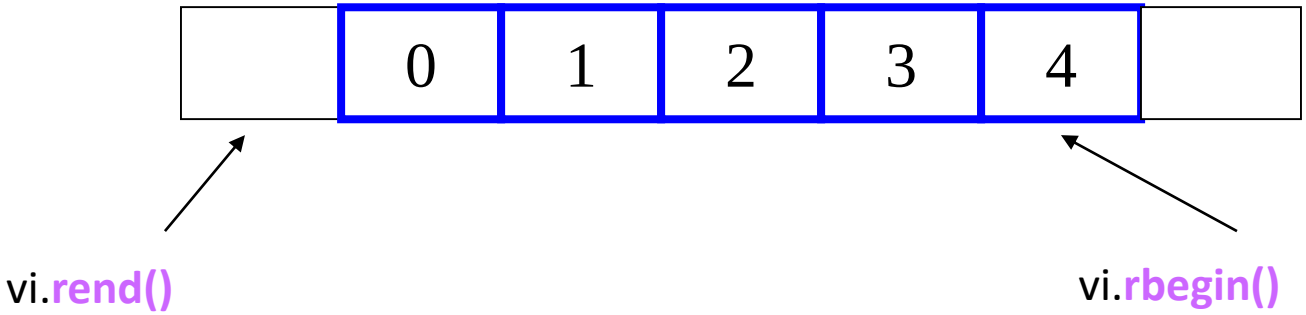
- L'output di quest'ultimo programma è:

```
Vector vi contains: 1 2 3 4 5
Now vector vi contains: 7 8 9 10 11
After inserting arr into vi, we have: 7 8 1 2 3 4 5 9 10
   11
After erasing the first three elements of vi, we have: 2
   3 4 5 9 10 11
```



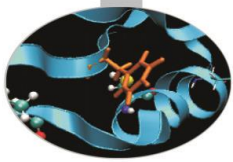
Commento

- ***Alcuni commenti:***
- for(ptr_r = vi.**rbegin()**; ptr_r != vi.**rend()**; ptr_r**++**) : l'iteratore *ptr_r* è *reverse* dunque itera il vector *vi* all'*indietro*, a partire dalla locazione di memoria puntata dall'iteratore restituito da *rbegin()* fino a quella puntata dall'iteratore restituito da *rend()* esclusa; *ptr_r* segue, inoltre, l'aritmetica dei puntatori.



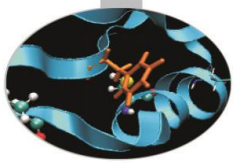
- for(ptr_c = vi.**begin()**; ptr_c != vi.**end()**; ptr_c**++**) : le stesse considerazioni valgono per l'iteratore ptr_c. Essendo stato dichiarato come *const*, esso può essere utilizzato per operazioni di *sola lettura*.

Commento



- vi.insert(vi.begin()+2, arr, arr+dim); con il metodo *insert* è possibile inserire all'interno del vector *vi*, a partire dalla locazione di memoria puntata dall'iteratore restituito da *begin()+2*, il contenuto di *arr*, qui specificato tramite indirizzi.

- vi.erase(vi.begin(), vi.begin()+3); il metodo *erase* permette di cancellare il contenuto delle celle di memoria di *vi* comprese tra gli iteratori restituiti da *begin()* e *begin()+3*, quest'ultima esclusa. Sono cioè cancellati i valori 7, 8 ed 1 da *vi*.



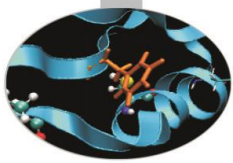
Commento

- I vector o più in generale tutti i container sequenziali della STL sono definiti in modo tale da contenere *un solo tipo* di dato.

- Questo ostacolo può essere aggirato facendo uso del *polimorfismo*.

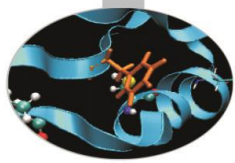
- E' altresì necessario adoperare i *puntatori* per implementare container polimorfici che non producano errori in fase di compilazione.

Esempio



Esempio: costruiamo un vector che contenga valori interi e double

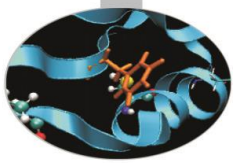
```
// header file cl_v.h
class Numbers{                              // classe base pura
 public:
  virtual void print() =0; };

class NumDbl : public Numbers {      // classe derivata
 private:
  double num;
 public:
  NumDbl(double=0);
  void print(); };

class NumInt : public Numbers {       // classe derivata
 private:
  int num;
 public:
  NumInt(int=0);
  void print(); };
```



Esempio

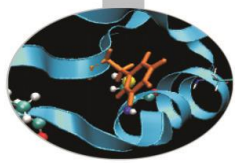
```
// file cl_fun.cc
#include <iostream>
#include "cl_v.h"

NumDbl::NumDbl(double nn){
  num=nn;
}

void NumDbl::print(){
  cout << "double: " << num << endl;
}
NumInt::NumInt(int nn){
  num=nn;
}

void NumInt::print(){
  cout << "int: " << num << endl;
}
```

Esempio



```
// file cl_prg.cc
#include <iostream>
#include "cl_v.h"
#include <vector>
int main() {
  vector<Numbers*> vct;          // vct rappresenta un vector di
   puntatori  a Numbers
  vct.push_back(new NumDbl(2.4));          // new è necessario
   perché vct contiene puntatori a NumDbl o a NumInt
  vct.push_back(new NumInt(20));
  vct.push_back(new NumInt());
  vct.push_back(new NumDbl(19.75));
  vct.push_back(new NumInt(32));

  vector<Numbers*>::const_iterator p;
  for(p=vct.begin(); p!=vct.end(); p++)
    (*p)->print();                    /* p itera un vector di
   puntatori, è come un doppio puntatore */
  return 0; }
```

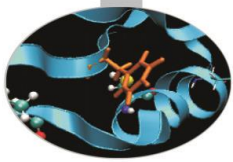
Esempio



Otteniamo come output:

```
double: 2.4
int: 20
int: 0
double: 19.75
int: 32
```

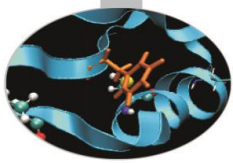
Il container sequenziale deque



- **DEQUE**: **D**ouble **E**nded **Que**ue, rappresenta un container adatto all' inserimento e all'eliminazione di dati su *entrambi* gli estremi: per questa ragione è principalmente usato per trattare strutture di tipo *FIFO*.
- L'accesso agli elementi di un deque è, comunque, *casuale*. Si possono dunque usare, a tal fine, i metodi *at()* e *operator[ ]*.
- Rispetto ad un vector, il container deque non dispone dei metodi *reserve()* e, quindi, *capacity()*, ma ha in più i metodi *push_front()* e *pop_front()* per, rispettivamente, aggiungere e rilevare dati in testa al container.
- Per utilizzare il container deque è necessario includere all'interno del programma l'istruzione:

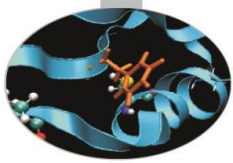
```
#include<deque>
```

Esempio



Esempio: creazione di un semplice deque

```
#include<iostream>
#include<deque>
#include<algorithm>
int main(){
  deque<int> d_int;          // dichiarazione di deque di
   interi chiamato d_int
  ostream_iterator<int> out(cout, " ");

  d_int.push_front(28);  // aggiunge un elemento in testa
  d_int.push_front(1);
  d_int.push_back(20);  // aggiunge un elemento in coda
  d_int.push_back(4);

  cout << "Deque d_int contains: ";
  copy(d_int.begin(), d_int.end(), out);
  cout << endl;

  // continua
```

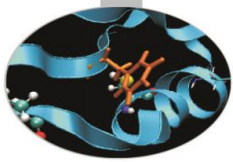


Esempio

```
// segue
  d_int[1]=15;
  d_int.at(2)=9;
  cout << "Now deque d_int contains: ";
  copy(d_int.begin(), d_int.end(), out);
  cout << endl;
  return 0; }
```

OUTPUT

```
Come output abbiamo:
Deque d_int contains: 1 28 20 4
Now deque d_int contains: 1 15 9 4
```

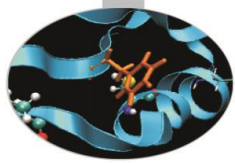


Il container sequenziale list

- Il container **list** rappresenta una sequenza di locazioni di memoria ottimizzata per l'inserimento e l'eliminazione di dati in qualsiasi punto del container.
- Al fine di rendere massima l'efficienza di questo tipo di operazioni, l'uso dei metodi *at()* ed *operator[ ]* non è consentito, così come non sono supportati iteratori casuali, ma soltanto bidirezionali.
- Ciascuna locazione (nodo) di memoria del container list contiene un puntatore sia al nodo *successivo* che a quello *precedente*.
- La classe list presenta fra i suoi metodi: *splice*, per inserire elementi all'interno del container; *sort*, per ordinare in maniera crescente gli elementi della lista; *reverse*, per invertire l'ordine degli elementi della lista; *unique*, per cancellare elementi ripetuti in posizioni consecutive; *remove*, per eliminare elementi che assumono un determinato valore; *merge*, per rimuovere gli elementi da una lista ed inserirli in un'altra in maniera ordinata.
- L'utilizzo del container list richiede l'inclusione all'interno del programma dell'istruzione:

```
#include<list>
```

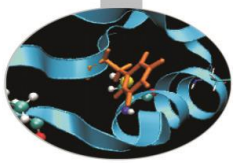
Esempio



Esempio: costruzione di un container list ed uso dei suoi metodi.

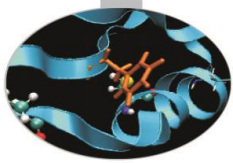
```
#include<iostream>
#include<list>
#include<algorithm>
int main(){
  list<int> l1;
  list<int> l2(3,2);// lista composta da tre elementi uguali a 2
  ostream_iterator<int> out(cout, " ");

  l1.push_front(32);
  l1.push_front(81);
  l1.push_front(75);
  l1.push_back(15);
  l1.push_back(75);
  l1.push_back(2);

  cout << "List l1 is:(start) ";
  copy(l1.begin(), l1.end(), out);
  cout << endl;
  // continua
```

Esempio



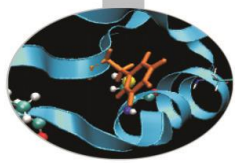
```
// segue
  cout << "List l2 is:(start) ";
  copy(l2.begin(), l2.end(), out);
  cout << endl;

  l1.splice(++++++l1.begin(), l2, l2.begin());
       // l1.begin()+3 sarebbe sbagliato: accesso casuale
       // copia nella 3 cella di l1 un elemento di l2, ovvero
   quello puntato da l2.begin()
  cout << "List l1 is:(splice) ";
  copy(l1.begin(), l1.end(), out);
  cout << endl;

  cout << "List l2 is:(splice) ";
  copy(l2.begin(), l2.end(), out); // l'elemento copiato in l1
   è stato cancellato in l2
  cout << endl;

  l1.merge(l2);
  cout << "List l1 is:(merge) ";
  copy(l1.begin(), l1.end(), out);
  cout << endl;
```

Esempio



```
// segue
  cout << "List l2 is:(merge) ";
  copy(l2.begin(), l2.end(), out); // ora l2 è vuota
  cout << endl;

  l1.remove(75);
  cout << "List l1 is:(remove 75) ";
  copy(l1.begin(), l1.end(), out);
  cout << endl;

  l1.sort();
  cout << "List l1 is:(sort) ";
  copy(l1.begin(), l1.end(), out);
  cout << endl;

  l1.unique();
  cout << "List l1 is:(unique) ";
  copy(l1.begin(), l1.end(), out);
  cout << endl;
  // continua
```

Esempio



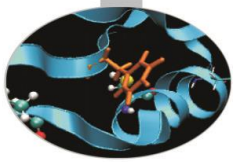
```
// segue
   l1.reverse();
   cout << "List l1 is:(reverse) ";
   copy(l1.begin(),l1.end(),out);
   cout << endl;
return 0; }
```

OUTPUT

L'output del programma è:

```
List l1 is:(start) 75 81 32 15 75 2
List l2 is:(start) 2 2 2
List l1 is:(splice) 75 81 32 2 15 75 2
List l2 is:(splice) 2 2
List l1 is:(merge) 2 2 75 81 32 2 15 75 2
List l2 is:(merge)
List l1 is:(remove 75) 2 2 81 32 2 15 2
List l1 is:(sort) 2 2 2 2 15 32 81
List l1 is:(unique) 2 15 32 81
List l1 is:(reverse) 81 32 15 2
```

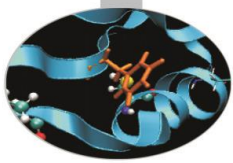
I container associativi: set e multiset



- Un container associativo rappresenta, in generale, un gruppo di valori *ordinati* cui è possibile accedere tramite *chiavi* di ricerca.
- Nel caso dei container **set** e **multiset** le chiavi di ricerca *coincidono* con i valori stessi.
- Il container set (insieme), a differenza del container multiset, non può contenere valori che si ripetono.
- Tutti i container associativi supportano iteratori bidirezionali, ma *non* ad accesso casuale.
- Avendo a che fare con container associativi torna utile fare uso di oggetti della classe *pair*. Essi hanno due membri public, *first* e *second* (spesso sono iteratori). Il primo può essere associato alla chiave ed il secondo al valore.
- Per utilizzare set e multiset è necessario includere nel programma l'istruzione:

```
#include<set>
```

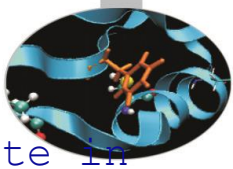
Esempio



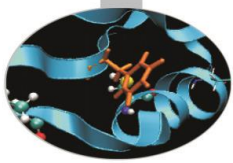
Esempio1: costruzione di un set di interi

```
#include <iostream>
#include <iterator>
#include <set>
#include <algorithm>
using namespace std;
int main(){
  int arr[5]={11,31,15,7,3};
  set<int> s1(arr,arr+5);  // dichiarazione del set s1 il cui
   contenuto è quello di arr
  ostream_iterator<int> out(cout, " ");

  cout << "Set s1 is: ";
  copy(s1.begin(), s1.end(), out);
  cout << endl;

  s1.insert(28);
  s1.insert(31);   // 31 è già presente in s1, questa istruzione viene ignorata
  s1.insert(51);
  cout << "Set s1 is: ";
  copy(s1.begin(), s1.end(), out);
  cout << endl;
  // continua
```

Esempio



```
// segue
cout << "Is the number 11 present?" << endl;
  if(s1.count(11))  // metodo count: restituisce 1 se 11 è presente in
   s1
    cout << "yes" << endl;
  else
    cout << "no" << endl;

  cout << "Is the number 55 present?" << endl;
  set<int>::const_iterator s_it;   // dichiarazione dell'iteratore
   costante s_it
  s_it=s1.find(55);  // metodo find: restituisce un iteratore che punta
   a 55
  if(s_it != s1.end())  // se 55 non è presente in s1 s_it va a puntare
   s1.end()
    cout << "yes" << endl;
  else
    cout << "no" << endl;

  s1.clear();  // cancella l'intero set s1
  cout << "Set s1 is: ";
  copy(s1.begin(), s1.end(), out);
  cout << endl;
  cout << "Set s1 size: " << s1.size() << endl;
  return 0; }
```



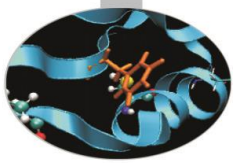
Esempio

OUTPUT

Abbiamo come output:

```
Set s1 is: 3 7 11 15 31
Set s1 is: 3 7 11 15 28 31 51
Is the number 11 present?
yes
Is the number 55 present?
no
Set s1 is:
Set s1 size: 0
```

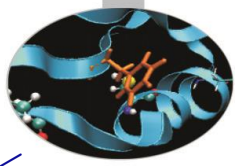
Esempio



esempio2: creazione di un multiset di interi

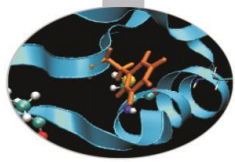
```
#include<iostream.h>
#include<set>
#include<algorithm>

int main(){
  int arr[10]={2,4,4,12,4,4,4,20,32,20};
  multiset<int> m1(arr,arr+10); // dichiarazione del multiset m1
   che contiene arr
  ostream_iterator<int> out(cout, " ");

  cout << "Multiset m1 is: ";
  copy(m1.begin(), m1.end(), out);
  cout << endl;

  cout << "How many 4 are into m1? ";
  cout << m1.count(4) << endl;  // il metodo count conta quante
   volte 4 compare in m1

  cout << "Lower bound of 20: " << *(m1.lower_bound(20)) <<endl;
  // il metodo lower_bound ritorna un iteratore che punta alla
   prima posizione
  // occupata da  20
```

Esempio



```
// segue
   cout << "Upper bound of 20: " << *(m1.upper_bound(20)) <<
    endl;
    // il metodo upper_bound restituisce un iteratore che punta
    alla posizione
    // occupata dal primo numero successivo a 20
return 0; }
```

Esempio

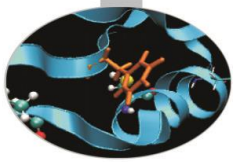


OUTPUT

L'output del programma è:

```
Multiset m1 is: 2 4 4 4 4 4 12 20 20 32
How many 4 are into m1? 5
Lower bound of 20: 20
Upper bound of 20: 32
```

I container associativi: map e multimap



- I container associativi **map** e **multimap** servono per la memorizzazione ed il recupero di *valori* associati a *chiavi* di ricerca.
- La dichiarazione di un oggetto map segue la sintassi:

```
map<tipo_chiave, tipo_valore> nome_oggetto;
```

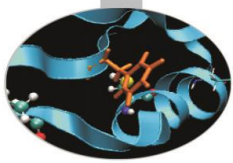
- Analogamente per un oggetto multimap:

```
multimap<tipo_chiave, tipo_valore> nome_oggetto;
```

- Il container *map* ammette chiavi di ricerca *uniche*, mentre nel container *multimap* la stessa chiave di ricerca può essere *ripetuta*.
- Le chiavi di ricerca sono automaticamente ordinate all'interno di entrambi i container.
- Per far uso del container map o multimap è necessario inserire all'interno del programma l'istruzione:

```
#include<map>
```

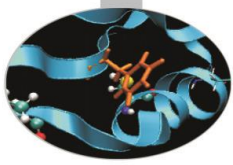
Esempio



esempio1: uso di map per illustrare la formazione di una squadra di calcio

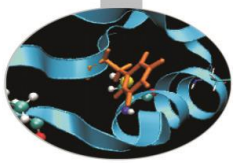
```
#include<iostream.h>
#include<map>
#include<string>

int main(){
  map<int, string> inter;
   // dichiarazione del map inter con chiavi di tipo int e valori
   di tipo string
  cout << "The size of map inter is: " << inter.size() << endl;
  inter[1]="Toldo";  // metodo operator[ ] per inserire un
   elemento nel map
  inter[2]="Cordoba";
  inter.insert(pair<int, string>(16, "Favalli"));
   // metodo insert per inserire un nuovo elemento nel map
  inter.insert(pair<int, string> (23, "Materazzi"));
  inter.insert(pair<int, string>(4, "Zanetti"));
  inter.insert(pair<int, string>5, "Emre"));
  inter.insert(pair<int, string>(11, "Stankovic"));
  inter.insert(pair<int, string>(7, "Van Der Meyde"));
```

Esempio



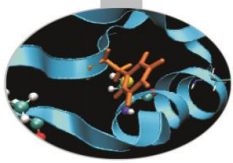
```
// segue
  inter.insert(pair<int, string>(4, "Zanetti"));
       // la chiave 4 è già presente, l'istruzione viene ignorata
  inter.insert(pair<int, string>(20, "Recoba"));
  inter.insert(pair<int, string>(30, "Martins"));
  inter.insert(pair<int, string>(32, "Vieri"));

  cout << "Is map inter empty? ";
  if(inter.empty())
    cout << "No" << endl;
  else
    cout << "Yes" << endl;

  map<int, string>::iterator pl;  // dichiarazione di un
                                  iteratore ti tipo map: punta sia
                                  alla
                                   // chiave (first) che al
                                  corrispondente valore (second)
  for(pl = inter.begin(); pl! = inter.end(); pl++)
    cout << pl->first << " " << pl->second << endl;

  cout << "The size of map inter is now: " << inter.size() <<
   endl;
```

Esempio



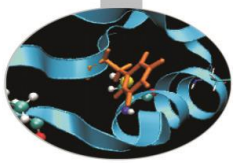
```
// segue

  cout << "Use of insert: ";
  inter.insert(pair<int,string>(30,"Adriano"));
  // la chiave 30 esiste già, l'istruzione insert viene
   ignorata
  map<int, string>::iterator pos;
  pos=inter.find(30);
  if(pos != inter.end())
    cout << (*pos).first << " " << (*pos).second << endl;
  else
    cout << "Number 30 is not in the map" << endl;

  cout << "Use of operator[]: ";  // il metodo operator[ ]
   forza il cambio di valore associato
                // alla chiave 30
  inter[30]="Adriano";
  pos=inter.find(30);
  cout << pos->first << " " << pos->second << endl;


// continua
```

Esempio



```
// segue
  cout << "Use of erase: ";
  inter.erase(30);
  inter.insert(pair<int, string>(10,"Adriano"));
  pos=inter.find(10);
  cout << pos->first << " " << pos->second << endl;
```

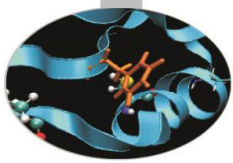
Esempio



Il programma dà il seguente output:

```
The size of map inter is: 0
Is map inter empty? Yes
1 Toldo
2 Cordoba
4 Zanetti
5 Emre
7 Van Der Meyde
11 Stankovic
16 Favalli
20 Recoba
23 Materazzi
30 Martins
32 Vieri
The size of map inter is now: 11
Use of insert: 30 Martins
Use of operator[]: 30 Adriano
Use of erase: 10 Adriano
```

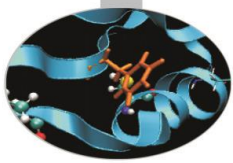
Esempio



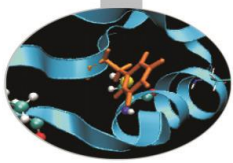
Esempio2: uso di multimap per ricercare i calciatori nati in un determinato anno

```
#include<iostream.h>
#include<map>
#include<string>
int main(){
  multimap<int, string> inter;
   // dichiarazione del multimap inter con chiavi di tipo int e
   valori di tipo string
   inter.insert(multimap<int,string>::value_type(1971,"Toldo"));
   // funziona come per gli oggetti della classe map
   inter.insert(multimap<int, string>::value_type(1971,
   "Gamarra"));
   inter.insert(multimap<int, string>::value_type(1973,
   "Materazzi"));
   inter.insert(multimap<int, string>::value_type(1973,
   "Zanetti"));
   inter.insert(multimap<int, string>::value_type(1974,
   "Gonzalez"));
// continua
```

Esempio



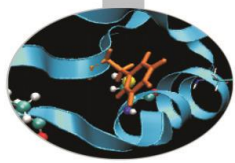
```
inter.insert(multimap<int,string>::value_type(1980, "Emre"));
inter.insert(multimap<int,string>::value_type(1974, "Cruz"));
inter.insert(multimap<int,string>::value_type(1976, "Recoba"));
inter.insert(multimap<int,string>::value_type(1973, "Vieri"));

int year;
cout << "Insert a year (0 to finish)";
cin >> year;

while(year != 0){

  cout << "The first we have found of " << year << " is: "
       << inter.find(year)->second << endl;
 // il metodo find ritorna un iteratore di tipo multimap al primo
 elemento la
 // cui chiave è uguale a year
cout << "There are " << inter.count(year) << " players born in"
             << year << endl;
 // il metodo count restituisce il numero di elementi associati alla
 chiave year
```



Esempio

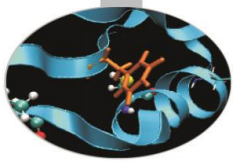
```
// segue
cout << "They are:" << endl;
multimap<int,string>::iterator i_it;
   // dichiarazione di un iteratore di tipo multimap
  for(i_it = inter.begin(); i_it! = inter.end(); i_it++) {
      if(i_it->first == year)
        cout << i_it->second << endl; }

  cout << endl;
  cout << "Insert a year (0 to finish)";
  cin >> year;
  }    // fine del ciclo while
  return 0; }
```

Esempio



Un possibile run del programma darebbe come output:

```
Insert a year (0 to finish)1971
The first we have found of 1971 is: Toldo
There are 2 players born in 1971
They are:
Toldo
Gamarra

Insert a year (0 to finish)1973
The first we have found of 1973 is: Materazzi
There are 3 players born in 1973
They are:
Materazzi
Zanetti
Vieri
Insert a year (0 to finish)0
```

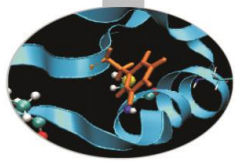


Gli adattatori di container

- Gli adattatori di container *non* forniscono un'implementazione vera e propria di una struttura dati in cui memorizzare elementi perché, come abbiamo visto, *non* supportano gli iteratori.

- Un adattatore di container viene infatti *costruito* su un container sequenziale scelto opportunamente.

- Funzioni membro *comuni* a tutti gli adattatori di container sono *push* e *pop*, che svolgono, rispettivamente, le operazioni di inserimento ed eliminazione di un elemento all'interno della struttura dati sulla quale è stato costruito l'adattatore. Tutte le funzioni di un adattatore di container sono implementate chiamando un metodo del container di base, per es. push richiama push_back e pop invoca pop_back in un adattatore stack



L'adattatore di container stack

- L'adattatore **stack** (*pila*) consente di creare una struttura per l'inserimento e l'eliminazione dei dati ad un solo estremo (*LIFO*). Per default uno stack è implementato su un container *deque*.

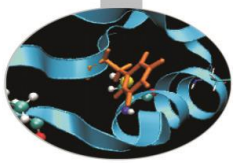
- I *metodi* di uno stack sono (oltre a *push* e *pop* che eseguono, rispettivamente, l'inserimento e la rimozione di un dato in cima alla pila): *top*, per ottenere il valore presente in cima alla pila (è implementato sulla funzione *back* del container di base); *empty*, per determinare se la pila è vuota (chiama l'omonima funzione del container di base) e *size*, per conoscere il numero degli elementi presenti nella pila (utilizza la funzione *size* del container di base).

- Per far uso di uno stack è necessario includere all'interno del programma l'istruzione:

```
#include<stack>
```

- oltre a quella corrispondente al container di base, se diverso da quello di default.

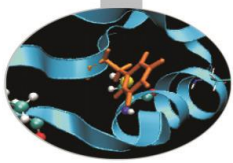
Esempio



Esempio: costruzione di una stack di interi

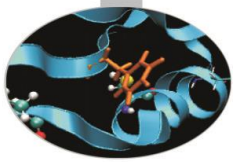
```
#include<iostream>
#include<stack>
#include<vector>
using namespace std;

int main(){
stack<double> s_vec;  // dichiarazione della stack s_vec basata
                                  //  un vector
cout << "Filling in s_vec" << endl;
  for(int j=0; j<5; j++)
    s_vec.push((j+1)*2);

  cout << "Is the stac s_vec empty? ";
  if(s_vec.empty())
    cout << "Yes" << endl;
  else
    cout << "No" << endl;

  cout << "The size of s_vec is: " << s_vec.size() << endl;
  cout << "On the top of s_vec there is: " << s_vec.top() << endl;
```

Esempio



```
cout << "Popping from s_vec: ";
while(s_vec.empty()!=1){
  cout << s_vec.top() << " ";
  s_vec.pop();
}
cout << endl;
return 0;}
```

Filling in s_vec
Is the stac s_vec empty? No
The size of s_vec is: 5
On the top of s_vec there is: 10
Popping from s_vec: 10 8 6 4 2



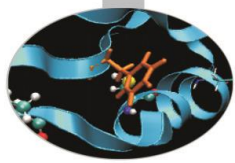
L'adattatore di container queue

- L'adattatore **queue** (coda) consente di costruire strutture di dati che supportino l'*inserimento* di elementi *in coda* e *l'eliminazione in testa* (*FIFO*).
- Un queue può avere come container di base *deque* (di default) o *list*.
- Le funzioni di un queue sono: *push*, per l'inserimento in coda (implementata su push_back); *pop*, per la rimozione in testa (implementata su pop_front); *front*, per conoscere il valore dell'elemento in testa al queue; *back*, per conoscere, invece, il valore dell'elemento in coda al queue; *empty*, che determina se il queue è vuoto e *size* che restituisce la dimensione del queue. Le ultime quattro funzioni sono realizzate facendo uso degli omonimi metodi del container di base.
- Naturalmente, per poter far uso dell'adattatore queue è necessario includere nel programma l'istruzione:

```
#include<queue>
```

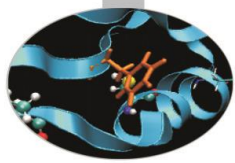
- oltre a quella corrispondente al container di base, se diverso da quello di default.

Esempio



Esempio: costruzione di un queue di interi

```
#include<iostream>
#include<queue>
using namespace std;
int main(){
  queue<int> tail; // dichiarazione di un queue di interi; il
   container di base è deque
  tail.push(2);
  tail.push(41);
  tail.push(97);
  tail.push(11);
  cout << "The element on the top of tail is: " << tail.front() <<
   endl;
  cout << "The element on the bottom of tail is: " << tail.back() <<
   endl;
  cout << "The size of tail is: " << tail.size() << endl;
  cout << "Popping from tail: ";
  while(tail.empty()!=1){
      cout << tail.front() << " ";
      tail.pop();} // il queue viene svuotato per
                   visualizzarne il contenuto
  }
```



Esempio

The element on the top of tail is: 2

The element on the bottom of tail is: 11

The size of tail is: 4