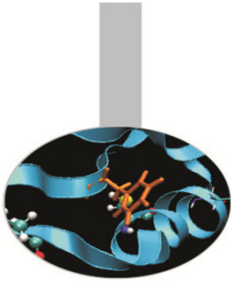
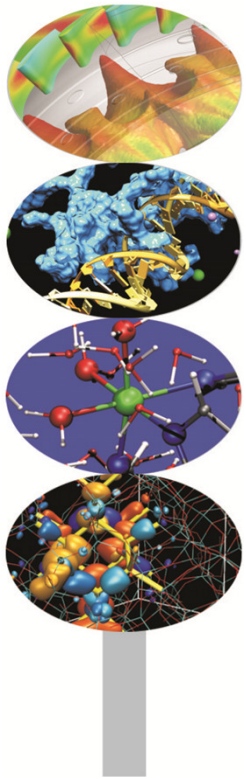
CINECA SCAI
SuperComputing Applications and Innovation

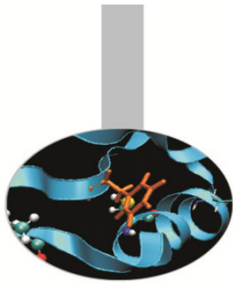


STL



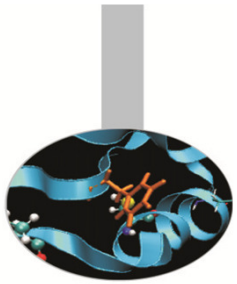
CINECA

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 e priority_queue

La Standard Template Library



STL = strutture dati + algoritmi

using namespace std;

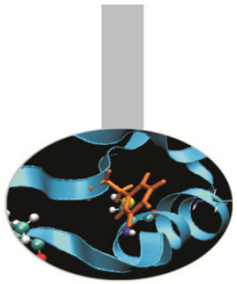
STL: <**container**, **iteratori**, **algoritmi**>

container : *template* di classi - strutture di dati

iteratori : classi (contenute nei container)

algoritmi : funzioni per manipolare i container

STL: <**container**, iteratori, algoritmi>



container: <*container di prima classe*, *adattatori di container*>

container di prima classe: <*container sequenziali*, *container associativi*, *quasi container* >

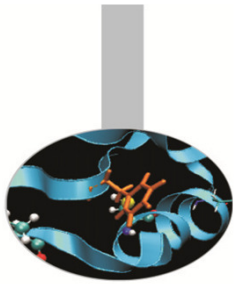
Sintassi:

tipo_container<tipo_dato> nome_container (lista_argomenti);

Esempi:

```
vector<int> v1(6,2);   // container vector v: 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
```

STL: <container, **iteratori**, algoritmi>



iteratori: per accedere agli elementi dei container di prima classe

Sintassi:

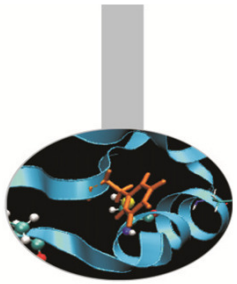
tipo_container<tipo_dato> ***::*** *tipo_iteratore nome_iteratore;*

tipo_iteratore_I/O<tipo_dato> nome_iteratore (lista_argomenti);

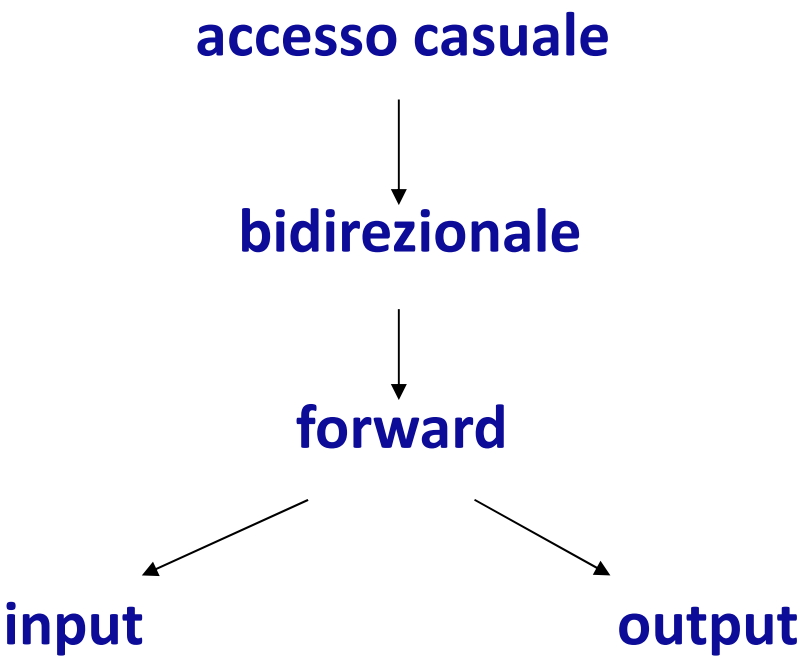
4 *tipi di iteratore* predefiniti:

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

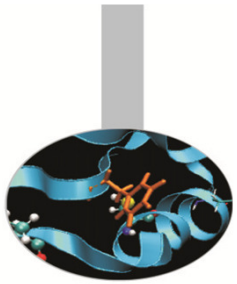
STL: <container, **iteratori**, algoritmi>



5 categorie di iteratori:



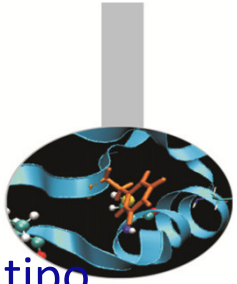
STL: <container, **iteratori**, algoritmi>



Esempi:

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

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 (" ")
```

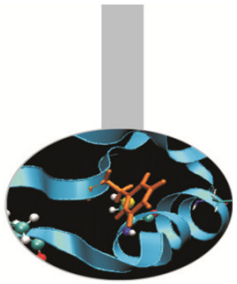
STL: <container, **iteratori**, algoritmi>



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

STL: <container, iteratori, algoritmi>



algoritmi: manipolazione dei container.

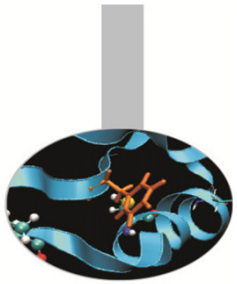
3 famiglie di algoritmi:

- Modifica del contenuto dei container; es.: fill(), replace(), swap()
- Non modifica del contenuto dei container; es.: find(), search()
- Numerici; es.: inner_product(), partial_sum()

Sintassi:

```
#include<algorithm>
nome_algoritmo( lista_argomenti );
```

STL: <container, iteratori, **algoritmi**>



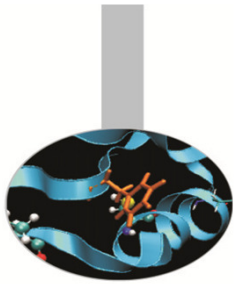
Esempi:

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.

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.

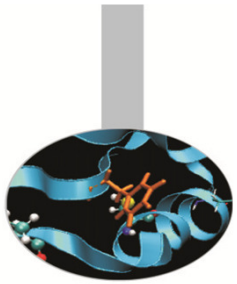
A differenza degli array, inoltre, i vector possono essere *assegnati tra di loro* grazie al costruttore di copia.

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>
```

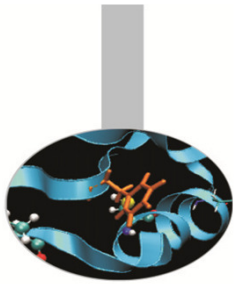
Esempio



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

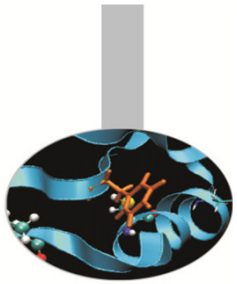
```
int main(){

  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



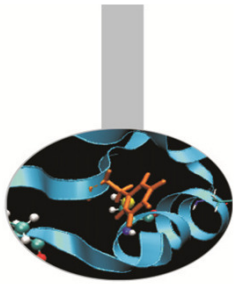
```
ostream_iterator<double> out_dbl (cout, " ");  //iteratore
                         // di tipo ostream per double
cout << "Vector v_dbl contains: ";
copy(v_dbl.begin(), v_dbl.end(), out_dbl);  /* algoritmo
     copy() per la stampa su standard output; fa uso dei
     metodi della classe vector begin() ed end() */
cout << endl;
cout << "Vector vi_two contains: “;
copy(vi_two.begin(), vi_two.end(), out_int);
cout << endl;
```

Esempio



```
  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;
}
```

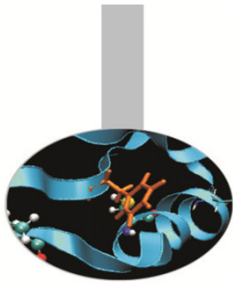
Esempio



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
```

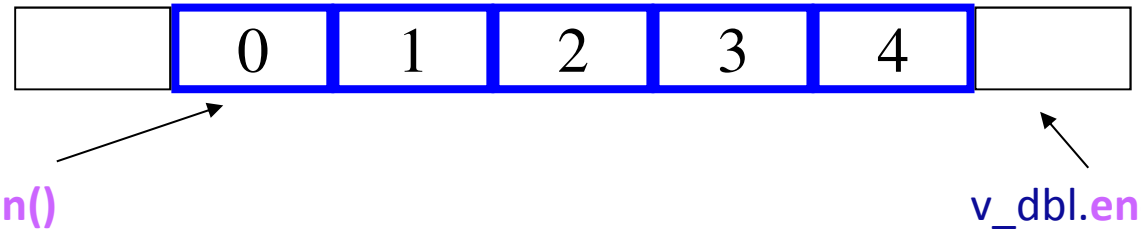
Commenti



Alcuni commenti:

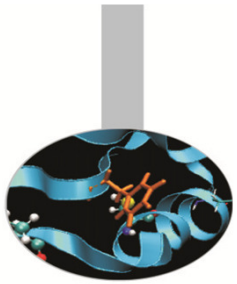
- **vector<**int**>** vi_two(arr_d, arr_d+dim); dice al compilatore di copiare il contenuto delle locazioni di memoria, comprese fra gli indirizzi *arr_d* e *arr_d+dim* escluso, all'interno del vector *vi_two*;
- copy(v_dbl.begin(), v_dbl.end(), out_dbl); dice al compilatore di copiare nella posizione 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:



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.

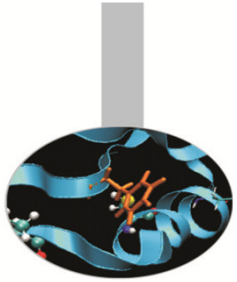
Esempio



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

```
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
```

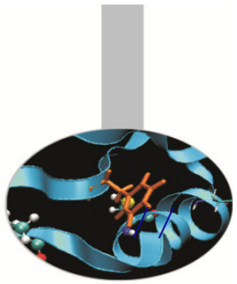
Esempio



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

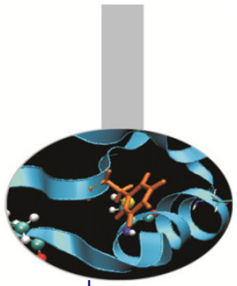
```
int main() {
  vector<int> vi1;
  const int dim=5;
  int arr_i[dim]={1,5,9,14,19};
  vector<int> vi2(arr_i, arr_i+dim);

  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



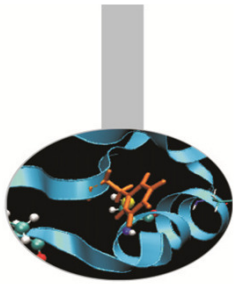
```
cout << "vi2 is: ";
  segue
 copy(vi2.begin(),vi2.end(),out_int);
 cout << endl;

 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



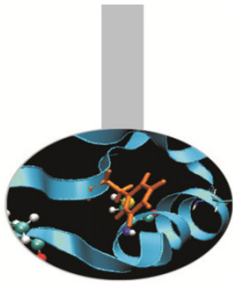
```
// segue
  vi1.push_back(20);
     // metodo push_back: aggiunge un elemento in coda al vector
   vi1
  vi1.push_back(32);
  vi1.at(2)=14;
     //metodo at: accede all'elemento del vector vi1 in
       posizione 2 e gli assegna il valore 2
  vi1[6]=4;       // metodo operator[ ]: funziona come at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



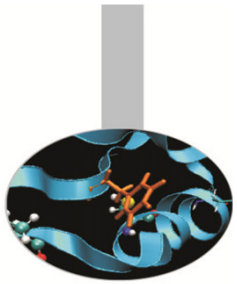
```
// segue
  vi1.pop_back();  // metodo pop_back(): dealloca l'ultimo
   elemento del vector vi1
  vi1.pop_back();
  cout << "The last two elements have been cancelled; vi1 is: ";
  copy(vi1.begin(), vi1.end(), out_int);
  cout << endl;

  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
```

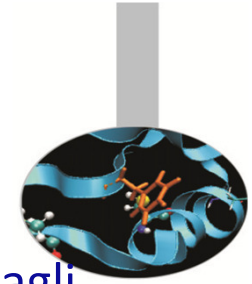
Commenti



Alcuni commenti:

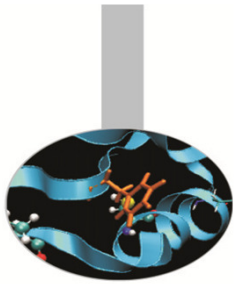
- vi1.**reserve(**10**)**; il metodo *reserve()* alloca 10 locazioni di memoria destinate al vector *vi1*. Può essere utile preallocare memoria perché l'uso dei metodi push_back e pop_back può allungare i tempi di compilazione e rallentare l'esecuzione del programma;
- vi1.**capacity()** : il metodo *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.

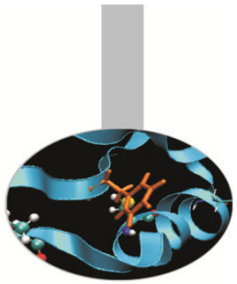
Esempio



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

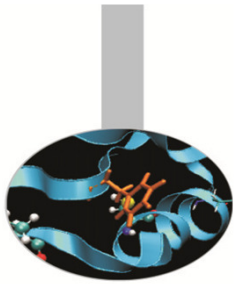
```
#include <iostream>
#include <iterator>
#include <vector>
#include <algorithm>
using namespace std;
  int main(){

  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



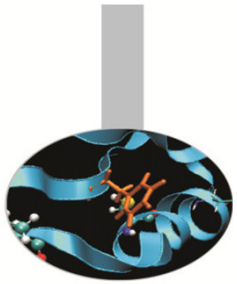
```
// segue
  cout << "Now vector vi contains: ";
  vector<int>::const_iterator ptr_c; // dichiarazione dell'iteratore
   const ptr_c
  for(ptr_c = vi.begin(); ptr_c != vi.end(); ptr_c++)
    cout << *ptr_c << " ";

  cout << endl;

  vi.insert(vi.begin()+2, arr, arr+dim); // metodo insert(), per
                        //      aggiungere nuovi elementi in 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:

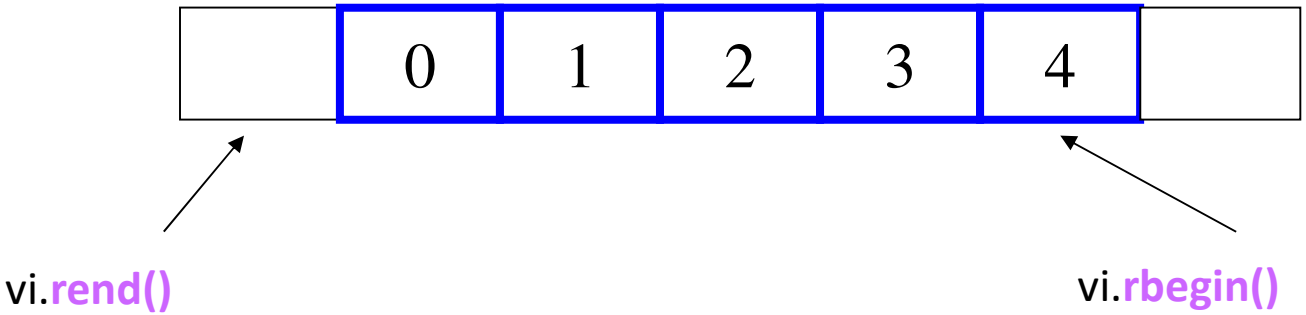
```
Vector vi contains: 1 2 3 4 5
Now vector vi contains: 7 8 9 10 11
After inserting arr into vi, we have: 7 8 1 2 3 4 5 9 10
   11
After erasing the first three elements of vi, we have: 2
   3 4 5 9 10 11
```



Commento

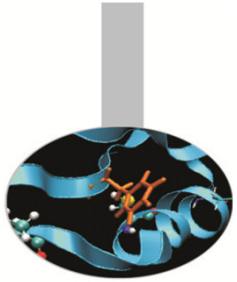
Alcuni commenti:

- 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*.

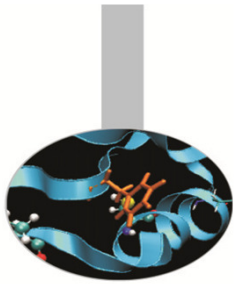
Commento



- 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*.

Esercizi



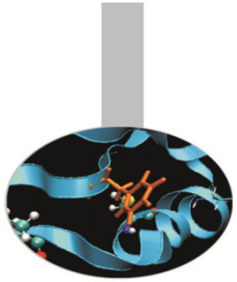
Esercizio array-1: Scrivere un programma che legga da tastiera 10 parole e le stampi in ordine inverso rispetto al loro inserimento. Utilizzare un array

Template: *tmp-array1.cpp*

Esercizio vector-1: Scrivere un programma che legga da tastiera 10 parole e le stampi in ordine inverso rispetto al loro inserimento. Utilizzare un vector

template: usare il codice dell'esercizio precedente

Commento

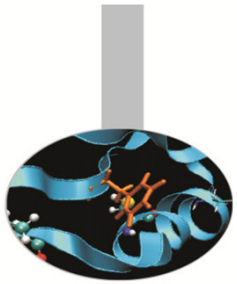


I vector o più in generale tutti i container sequenziali della STL sono definiti in modo tale da contenere *un solo tipo* di dato.

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

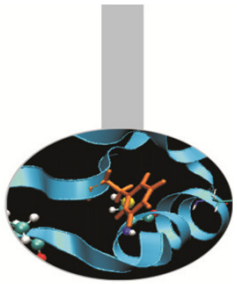


Esempio 4: costruiamo un vector che contenga valori interi e double

// header file emp-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(); };
```

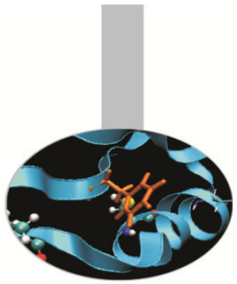
Esempio



// file emp-cl_fun.cpp

```
#include "emp-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;
}
```

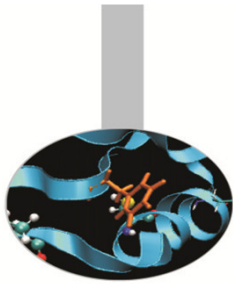
Esempio



// file emp-vector4.cpp

```
#include "cl_v.h"
int main() {
  vector<Numbers*> vct; // vct: 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
  return 0; }
```

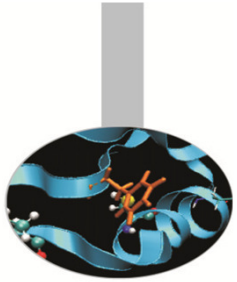
Esempio



Otteniamo come output:

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

Il container sequenziale deque

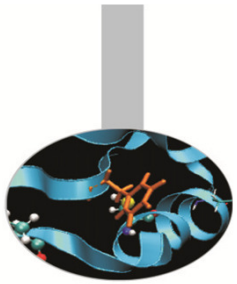


DEQUE: **D**ouble **E**nded **Que**ue, rappresenta un container adatto all'inserimento e all'eliminazione di dati su *entrambi* gli estremi: per questa ragione è principalmente usato per trattare strutture di tipo *FIFO*.

L'accesso agli elementi di un deque è, comunque, *casuale*. Si possono dunque usare, a tal fine, i metodi *at()* e *operator[ ]*.

A differenza di un vector, un deque *non* occupa celle contigue di memoria.

Il container sequenziale deque

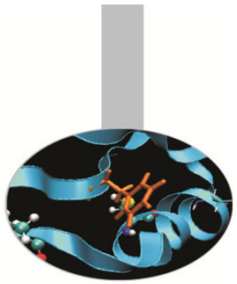


Rispetto ad un vector, il container deque non dispone dei metodi *reserve()* e, quindi, *capacity()*, ma ha in più i metodi *push_front()* e *pop_front()* per, rispettivamente, aggiungere e rilevare dati in testa al container.

Per utilizzare il container deque è necessario includere all'interno del programma l'istruzione:

```
#include <deque>
```

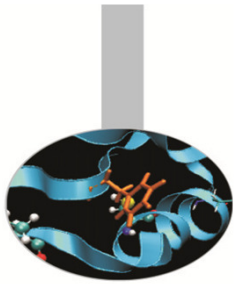
Esempio



Esempio deque1: creazione di un semplice deque

```
#include<iostream.h>
#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

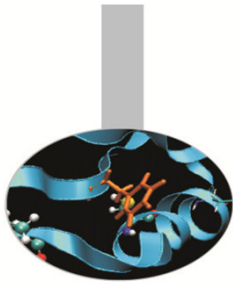


```
// segue
  d_int[1]=15;
  d_int.at(2)=9;
  cout << "Now deque d_int contains: ";
  copy(d_int.begin(), d_int.end(), out);
  cout << endl;
  return 0; }
```

OUTPUT

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

Esercizi



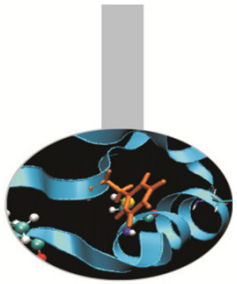
Esercizio deque-1: implementare una semplice coda FIFO e una coda LIFO

template: *tmp-deque1.cpp*:

```
cout << "Valori LIFO..." << endl;
while ( v != 9999 ) {
        cout << "trasmetti un numero intero ";
        cin >> v; . . .
}
cout << "I valori inseriti sono: "; . . .; cout << endl;
d_int.clear();
```

. . .

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.

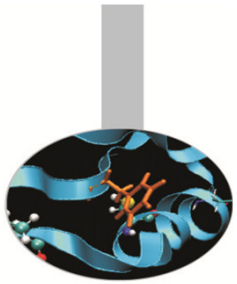
Al fine di rendere massima l'efficienza di questo tipo di operazioni, l'uso dei metodi *at()* ed *operator[ ]* non è consentito, così come non sono supportati iteratori casuali, ma soltanto bidirezionali.

Ciascuna locazione (nodo) di memoria del container list contiene un puntatore sia al nodo *successivo* che a quello *precedente*.

L'utilizzo del container list richiede l'inclusione all'interno del programma dell'istruzione:

```
#include<list>
```

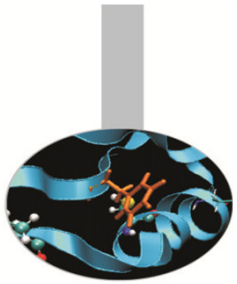
Il container sequenziale list



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.

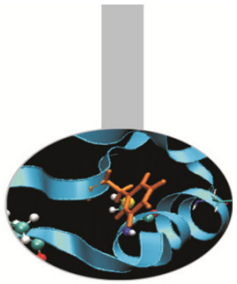
Esempio



Esempio emp-list1: costruzione di un container list ed uso dei suoi metodi.

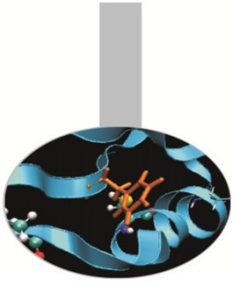
```
#include<list>

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



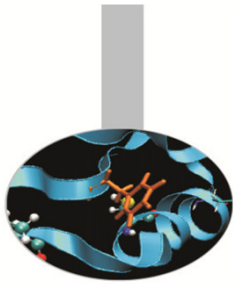
```
// segue
  cout << "List l2 is:(start) ";
  copy(l2.begin(), l2.end(), out);
  cout << endl;

  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



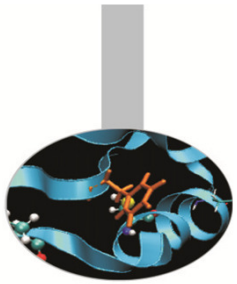
```
// segue
   l1.reverse();
   cout << "List l1 is:(reverse) ";
   copy(l1.begin(),l1.end(),out);
   cout << endl;
return 0; }
```

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
```

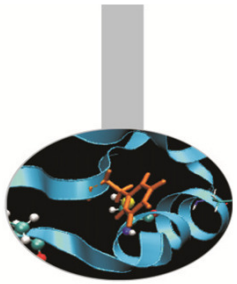
Esercizio



Esercizio list-1: implementare usando una lista un interprete semplice che raccoglie valori interi e infine ne calcola la somma e la media; usare i metodi visti nell'esempio precedente

template: *tmp-list1.cpp*

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.

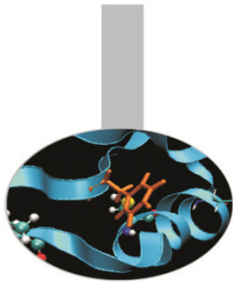
Tutti i container associativi supportano iteratori bidirezionali, ma *non* ad accesso casuale.

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>
```

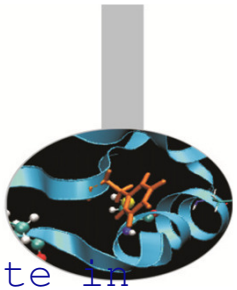
Esempio



Esempio emp-set1: costruzione di un set di interi

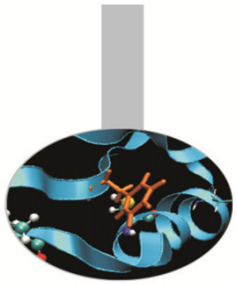
```
#include <set>

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

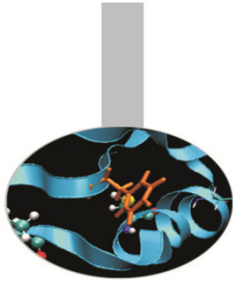


OUTPUT

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
```

Esercizio

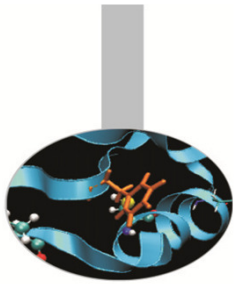


Esercizio set-1: implementare usando un set una collezione semplice di «esemplari avvistati», da interrogare successivamente alla ricerca dell'esemplare di interesse

template: *tmp-set1.cpp*

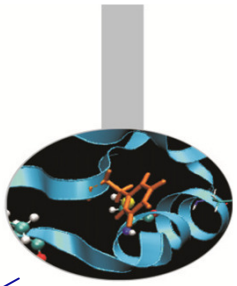
Esempio

Esempio emp-mset1: creazione di un multiset di interi



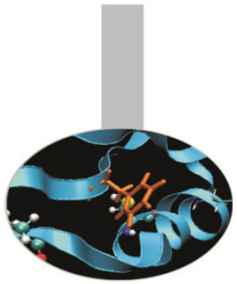
```
#include<set>

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
  pair<multiset<int>::iterator, multiset<int>::iterator> pr;
   /* pr è un oggetto della classe pair, i suoi due membri sono
   iteratori  */
  pr=m1.equal_range(4); /* il metodo equal_range serve per
   calcolare lower_bound ed upper_bound di 4. Qui pr, della
   classe pair, non è associato ad una coppia chiave/valore */
  cout << "Lower_bound of 4: " << *(pr.first) << endl;
  cout << "Upper_bound of 4: " << *(pr.second) << endl;
   // dereferenziazione dei membri di pr
  return 0; }
```

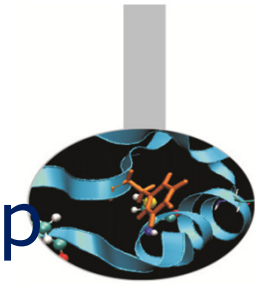
Esempio



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
Lower_bound of 4: 4
Upper_bound of 4: 12
```

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;
```

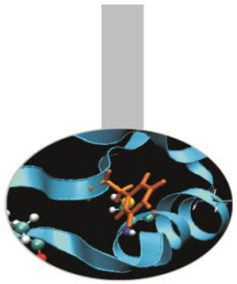
Il container *map* ammette chiavi di ricerca *uniche*, mentre nel container *multimap* la stessa chiave di ricerca può essere *ripetuta*.

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>
```

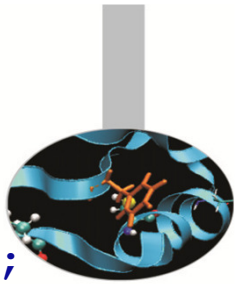
Esempio



Esempio emp-map1: uso di map per illustrare la formazione di una squadra di calcio

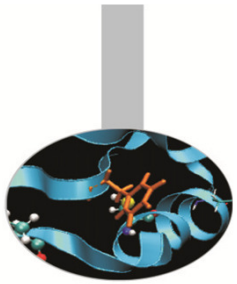
```
#include<map>

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(map<int, string>::value_type(16, "Favalli"));
    // metodo insert per inserire un nuovo elemento nel map
    //l'istruzione map<int, string>::value_type(16,"Favalli") crea
    //un oggetto pair ove first è la chiave e second il valore
   inter.insert(map<int, string>::value_type(23, "Materazzi"));
   inter.insert(map<int, string>::value_type(4, "Zanetti"));
   inter.insert(map<int, string>::value_type(5, "Emre"));
   inter.insert(map<int, string>::value_type(11, "Stankovic"));
   inter.insert(map<int, string>::value_type(7, "Van Der Meyde"));
```

Esempio



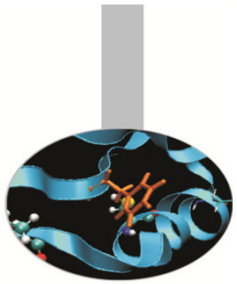
```
// segue
  inter.insert(map<int, string>::value_type(4, "Zanetti"));
       // la chiave 4 è già presente, l'istruzione viene ignorata
  inter.insert(map<int, string>::value_type(20, "Recoba"));
  inter.insert(map<int, string>::value_type(30, "Martins"));
  inter.insert(map<int, string>::value_type(32,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



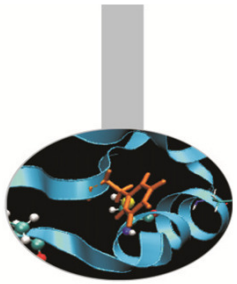
```
// segue

  cout << "Use of insert: ";
  inter.insert(map<int,string>::value_type(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(map<int, string>::value_type(10,"Adriano"));
  pos=inter.find(10);
  cout << pos->first << " " << pos->second << endl;
  cout << "Use of pair: ";
  pair< map<int,string>::iterator, bool > player;
  player = inter.insert(map<int,string>::value_type(2,
   "Cordoba"));
   // se la key esiste già, non viene effettuato alcun
   l'inserimento nel map e
   // il valore bool ritorna il valore false
  if(player.second)
    cout << player.first->first << " " << player.first->second
         << " " << "joins the map" << endl;
  else
    cout << player.first->first << " " << player.first->second
         << " " << "in the map already" << endl;
  return 0;}
```

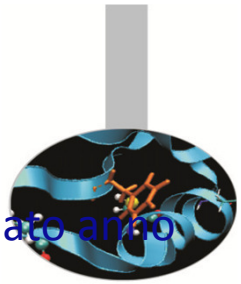
Esempio



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
Use of pair: 2 Cordoba in the map already
```

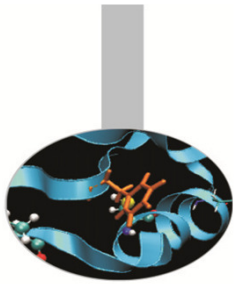
Esempio



Esempio emp-mmap1: uso di multimap per ricercare i calciatori nati in un determinato anno

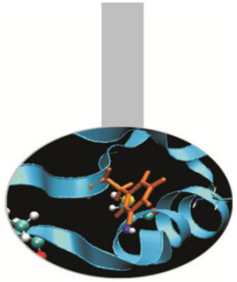
```
#include<map>


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



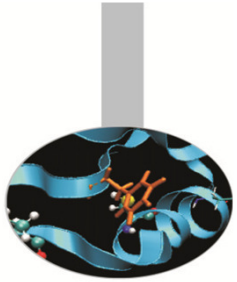
```
inter.insert(multimap<int,string>::value_type(1980, "Emre"));
inter.insert(multimap<int,string>::value_type(1974, "Cruz"));
inter.insert(multimap<int,string>::value_type(1976, "Recoba"));
inter.insert(multimap<int,string>::value_type(1973, "Vieri"));

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; }
```

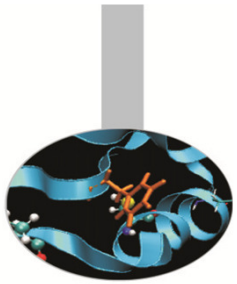
Esempio



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
```

Esercizio

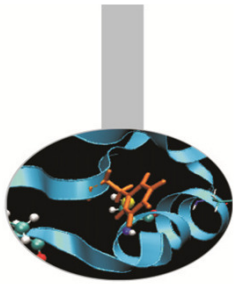


Esercizio mmap-1: generare una pagina HTML semplice utilizzando multimap. Deve contenere:

1. **Titolo della pagina**
2. **Titolo del Capitolo**
3. **Testo**

template: *tmp-mmap1.cpp*

Gli adattatori di container



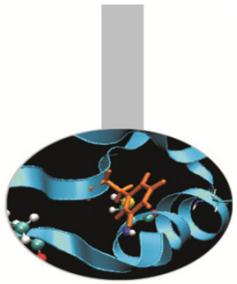
Gli adattatori di container *non* forniscono un'implementazione vera e propria di una struttura dati in cui memorizzare elementi perché *non* supportano gli iteratori.

Un adattatore di container viene *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



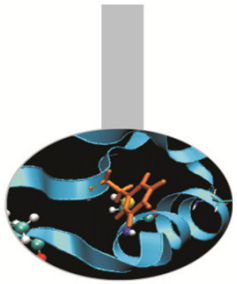
L'adattatore **stack** (*pila*) consente di creare una struttura per l'inserimento e l'eliminazione dei dati ad un solo estremo (*LIFO*). Per default uno stack è implementato su un container *deque* e supporta vector e list.

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.

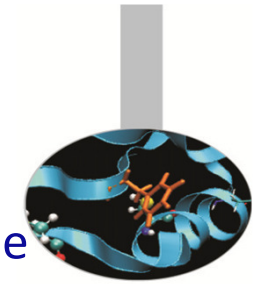
L'adattatore di container stack



I *metodi* di uno stack sono:

- ***push***, per inserire dati in cima alla pila
- ***pop***, per 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);
- ***size***, per conoscere il numero degli elementi presenti nella pila (utilizza la funzione *size* del container di base).

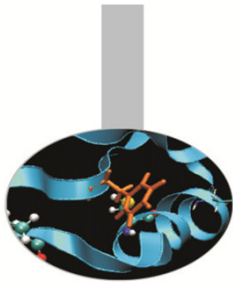
Esempio



Esempio emp-stack1: costruzione di uno stack di interi basata su un deque e di uno stack di double basata su un vector

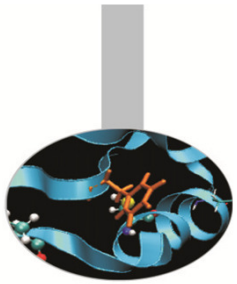
```
#include<stack>
#include<vector>

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

  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;}
```

L'adattatore di container queue



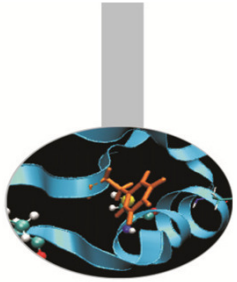
L'adattatore **queue** (coda) consente di costruire strutture di dati che supportino l'*inserimento* di elementi *in coda* e *l'eliminazione in testa* (*FIFO*).

Un queue può avere come container di base *deque* (di default) o *list*.

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.



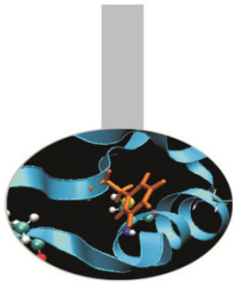
L'adattatore di container queue

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;
- ***size***, che restituisce la dimensione del queue.

Le ultime quattro funzioni sono realizzate facendo uso degli omonimi metodi del container di base.

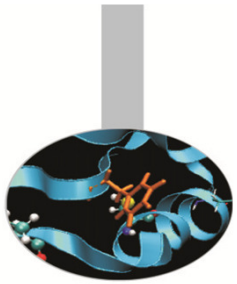
Esempio



ESempio emp-queue1: costruzione di un queue di interi basato su un deque

```
#include<queue>
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

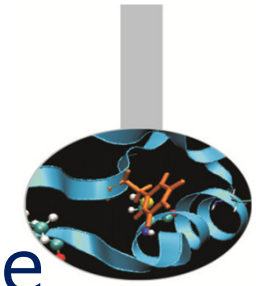


```
// segue
cout << endl;
  if(tail.empty())
    cout << "Now tail is empty" << " and its size is " <<
   tail.size() << endl;
  return 0; }
```

OUTPUT

Abbiamo come output:

```
The element on the top of tail is: 2
The element on the bottom of tail is: 11
The size of tail is: 4
Popping from tail: 2 41 97 11
Now tail is empty and its size is 0
```



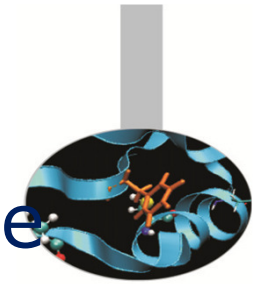
L'adattatore di container priority_queue

L'adattatore **priority_queue** serve per realizzare strutture di dati per l'inserimento *ordinato* di elementi e la loro eliminazione dalla *testa*.

I container di base di un priority_queue sono *vector* (di default) e *deque*.

Quando viene aggiunto un nuovo dato al priority_queue, gli elementi vengono ordinati in modo tale che *il più grande occupi la posizione di testa*, ovvero sia il primo ad essere rimosso dalla struttura. Ciò è ottenuto facendo uso di un algoritmo chiamato *heapsort*.

L'adattatore di container priority_queue

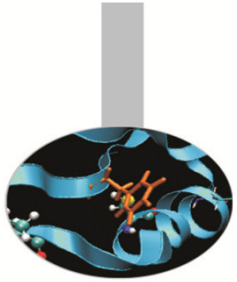


L'uso di un priority_queue richiede di inserire all'interno del programma l'istruzione

```
#include<queue>
```

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

L'adattatore di container priority_queue



Le funzioni di cui dispone un priority_queue sono:

push, per inserire un elemento nella posizione appropriata in base al suo valore (implementata tramite il metodo push_back del container di base e l'algoritmo heapsort);

pop, per eliminare l'elemento in cima all'adattatore (si fa uso della funzione pop_back del container di base);

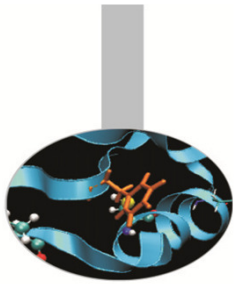
top, per ottenere l'elemento in cima al priority_queue (attraverso la funzione *front* del container di base);

empty, per determinare se il priority_queue è vuoto;

size, per conoscere il numero di elementi.

Le ultime due funzioni chiamano gli omonimi metodi del container di base.

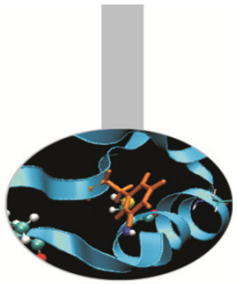
Esempio



Esempio emp-pqueue1: realizzazione di un priority_queue di interi

```
#include <queue>

int main(){
  priority_queue<int> ptail; // dichiarazione del priority_queue di
                             // interi ptail

  ptail.push(12);
  ptail.push(4);
  cout << "The element on the top of ptail is:" << ptail.top() << endl;
  ptail.push(45);
  ptail.push(5);
  cout << "The element on the top of ptail is:" << ptail.top() << endl;
  ptail.push(27);
  cout << "The element on the top of ptail is:" << ptail.top() << endl;

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

  // continua
```

Esempio



```
// segue
cout << "Popping from ptail:";
  while(ptail.empty()!=1)
    {
      cout << ptail.top() << " ";
      ptail.pop();  // ptail viene svuotata per leggerne
                    // il contenuto
    }
  cout << endl;
  cout << "The size of ptail is now: " << ptail.size() << endl;
  return 0;}
```

OUTPUT

Otteniamo come output:

```
The element on the top of ptail is:12
The element on the top of ptail is:45
The element on the top of ptail is:45
The size of ptail is:5
Popping from ptail:45 27 12 5 4
The size of ptail is now: 0
```