Standard Template Library (STL)

- STL inclusa nella libreria standard del C++
- Componenti principali:
  - **Containers**: sono oggetti che rappresentano delle collezioni di altri oggetti, ovvero un contenitore è un oggetto che contiene altri oggetti (vector, list, ...). Ogni container ha funzioni membro associate. Alcune funzionalità sono comuni a tutti i container. Altre funzionalità sono specifiche dei singoli container.
  - **Iterators**: sono oggetti che consentono di attraversare (scorrere) un container qualunque esso sia e operare con gli elementi. Sono intermediari tra containers e algoritmi. Sono assimilabili a puntatori ma si comportano in modo più "intelligente".
  - **Algoritmi**: algoritmi (find, sort, count ...) che per mezzo degli iteratori si applicano ai container. Gli algoritmi sono indipendenti dai contenitori.

Container

- Tipi di containers
  - *Sequential*: contenitori che organizzano i dati in modo lineare (vector, deque, list)
  - *Associative*: gli elementi sono coppie (chiave, valore): set, multiset, map, multimap
  - *Adapters*: stack, queue, priority queue
- File header dei container della STL:

<vector> <list>

<deque>

<queue> contiene queue e priority queue

<stack>

<map> contiene map e multimap

<set> contiene set e multiset

Container

- STL non utilizza generalmente l'ereditarietà e le funzioni virtuali (per motivi di efficienza).
- Quando si inserisce un elemento in un container, **viene effettuata una copia** di tale elemento (se si vuole eseguire l'inserimento in un contenitore di oggetti che contengono memoria allocata dinamicamente è necessario definire un costruttore di copia su tali oggetti per ottenere una copia profonda, altrimenti viene eseguita una copia superficiale).
- il tipo di elemento da memorizzare in un contenitore deve supportare un insieme minimo di funzionalità: costruttore di copia, operatore di assegnamento (se il costruttore di copia di default non è sufficiente).
- E' necessario definire l'operatore minore (<) per gli oggetti contenuti in contenitori dove di vuole utilizzare algoritmi di ricerca e ordinamento.

Container

Funzioni membro comuni a tutti i container della STL	Descrizione
costruttore di default	Costruttore per l'inizializzazione di default del container. Di norma, ogni container ha diversi costruttori che forniscono una varietà di metodi di inizializzazione del container
costruttore di copia	Costruttore che inizializza il container come copia di un container esistente dello stesso tipo
distruttore	Funzione per la distruzione del container
`empty`	Restituisce true se il container non contiene elementi, false altrimenti
`max_size`	Restituisce il numero massimo di elementi per un container
`size`	Restituisce il numero di elementi presenti correntemente in un container
`operator=`	Assegna un container a un altro
`operator<`	Restituisce true se il primo container è minore del secondo, false altrimenti
`operator<=`	Restituisce true se il primo container è minore o uguale al secondo, false altrimenti
`operator>`	Restituisce true se il primo container è maggiore del secondo, false altrimenti
`operator>=`	Restituisce true se il primo container è maggiore o uguale al secondo, false altrimenti
`operator==`	Restituisce true se il primo container è uguale al secondo, false altrimenti
`operator!=`	Restituisce true se il primo container non è uguale al secondo, false altrimenti
`swap`	Scambia gli elementi dei due container

Container

- dati due oggetti contenitori, a e b, si definisce a minore di b (operator<) se a precede b nell'ordinamento "lessicografico", cioè se:
  - tutti gli elementi corrispondenti sono uguali e la dimensione di a è minore della dimensione di b, oppure
  - indipendentemente dalla dimensione di a e di b, il primo elemento di a non uguale al corrispondente elemento di b è minore del corrispondente elemento di b (e quindi è necessario che anche nel tipo degli elementi sia definito operator<)
- due oggetti contenitori sono uguali (operator==) se hanno la stessa dimensione e tutti gli elementi corrispondenti sono uguali (e quindi è necessario che anche nel tipo degli elementi sia definito operator==);
- La dimensione di un contenitore (cioè il numero dei suoi elementi) non è prefissata e immodificabile (come negli array del C).
- I container supportano l'assegnamento/copia (operatore =)

Iteratori

- Uso: accesso agli elementi dei container sequenziali e associativi
- l'operatore (*) applicato ad un iteratore ritorna l'elemento del container a cui punta l'iteratore
- l'operazione ++ su un iteratore restituisce un iteratore all'elemento successivo del container
- l'utilizzo di oggetti di tipo iterator è consentito per riferire elementi di un container che può essere modificato, nel caso di accesso a container non modificabili si può utilizzare un oggetto di tipo const_iterator
- dereferenziare un iteratore posizionato al di fuori del suo container genera un errore logico in fase di esecuzione
- creare un iteratore non const per un container const genera un errore
- iteratori inversi: attraversamento dei container in direzione inversa.

Iteratori

- Un oggetto iteratore si ottiene istanziando un tipo iteratore qualificato con il nome della classe di appartenenza. Per esempio, consideriamo il contenitore vector, specializzato con argomento int; l'istruzione:

  vector<int>::iterator it;

  definisce l'oggetto iteratore it, istanza del tipo iterator della classe vector<int>

Iteratori

- Tipi di iteratori:
  - *Bidirezionale:* legge e scrive i valori, supporta incremento e decremento
  - *ad accesso casuale:* legge e scrive i valori con spostamenti arbitrari. Consente di utilizzare l'aritmetica e il confronto dei puntatori.

Container	Tipo di operatore supportato
Container sequenziali	
`vector`	ad accesso casuale
`deque`	ad accesso casuale
`list`	bidirezionale
Container associativi	
`set`	bidirezionale
`multiset`	bidirezionale
`map`	bidirezionale
`multimap`	bidirezionale
Adattatori al container	
`stack`	nessun iteratore supportato
`queue`	nessun iteratore supportato
`priority_queue`	nessun iteratore supportato

Iteratori

Typedef predefinite per i tipi di operatore	Direzione di ++	Funzionalità
iterator	avanti	lettura/scrittura
const_iterator	avanti	lettura
reverse_iterator	indietro	lettura/scrittura
const_reverse_iterator	indietro	lettura

- Le operazioni basilari sugli iteratori sono 3 e precisamente:
  - "accedi all'elemento puntato" (dereferenziazione, operatore *)
  - "punta al prossimo elemento" (incremento, rappresentata dall'operatore ++)
  - "esegui il test di uguaglianza o disuguaglianza" tra operatori (rappresentate dagli operatori == e !=): due iteratori sono uguali quando puntano allo stesso elemento del container

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

Container

- Funzioni dei soli container sequenziali e associativi:
  - **begin()** restituisce un iteratore riferito al primo elemento del container
  - **end()** restituisce un iteratore riferito alla posizione successiva **dopo** la fine del container (iteratore che punta alla fine della sequenza, non esiste un iteratore NULL, come nei normali puntatori)
  - **rbegin()** restituisce un iteratore inverso riferito all'ultimo elemento del container (ovvero un iteratore che punta all'inizio della sequenza inversa)
  - **rend()** restituisce un iteratore inverso che punta alla fine della sequenza inversa
  - **erase()** elimina uno o più elementi del container. Dopo la chiamata iteratori che si riferivano ad elementi eliminati vengono invalidati. Ritorna un iteratore che punta all'elemento successivo.

    iterator erase (iterator position);

    iterator erase (iterator first, iterator last); **elimina da "first" incluso a "last" escluso**
  - **clear()** elimina tutti gli elementi del container
  - **swap(T& a, T& b)** scambia i valori di due contenitori a e b dello stesso tipo

Container sequenziali

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

- Le classi vector e deque si basano su array monodimensionali.
- Un vector può cambiare dimensioni in modo dinamico. A differenza degli array in stile C e C++, i vector possono essere assegnati tra loro.
- La classe list rispetto a vector manca dell'accesso tramite indice e di varie operazioni sugli iteratori, che non sono ad accesso casuale ma bidirezionali; è più efficiente di vector nelle operazioni di inserimento e cancellazione di elementi in posizioni intermedie.

Container sequenziali

- Funzioni comuni: **front()** che restituisce un riferimento al primo elemento del container, **back()** restituisce un riferimento all'ultimo elemento del container
- **push_back(const value_type& val)**

inserisce un nuovo elemento alla fine del container

- **pop_back()** elimina l'ultimo elemento del container
- **insert()** per inserire un elemento prima della posizione specificata da un iteratore position, ritorna un iteratore che punta all'element inserito

```
iterator insert (iterator position, const value_type& val);
```

- **assign()** per assegnare nuovi elementi ad un contenitore presi da un contenitore sorgente in un intervallo definito da due iteratori first e last

```
void assign (InputIterator first, InputIterator last);
```

vector

- memorizza un array monodimensionale
- supporta iteratori ad accesso casuale (tramite l'overloading dell'operatore []) per accedere agli elementi con un indice
- vector cresce automaticamente di dimensione dopo ogni inserimento
- L'inserimento nel mezzo di un vector non è efficiente (e allo stesso modo l'eliminazione) perchè deve essere spostata l'intera porzione del vector che segue l'elemento inserito (o eliminato): gli elementi di un vector, infatti, occupano celle di memoria contigue, come gli array
- per eliminare un elemento in testa: vec.erase(vec.begin());

vector

```
#include <vector>
#include <iostream>
using namespace std;
int main() { vector<int> vNumbers;
 vNumbers.push_back(1); vNumbers.push_back(110);
 vNumbers.push_back(5);vNumbers.push_back(74); vNumbers[2] = 7;
 vNumbers.insert(vNumbers.begin() + 3, 22);
 cout << "vNumbers[0]="<< vNumbers[0] << endl;
 vNumbers.insert(vNumbers.end() - 1, 44);
 vector<int>::iterator it = vNumbers.begin();
 it++; *it = 555;
 for (int i = 0; i<vNumbers.size(); i++) { cout << vNumbers[i] << " "; }
 cout << endl;
 for (it = vNumbers.begin(); it != vNumbers.end(); it++) { cout << *it << " "; }
 cout << endl;
 vector< int >::reverse_iterator p2;
 for (p2 = vNumbers.rbegin(); p2 != vNumbers.rend(); ++p2) cout << *p2 << " ";
 cout << endl;
 it = vNumbers.begin(); it++;
 vNumbers.erase(it);
 for (it = vNumbers.begin(); it != vNumbers.end(); it++) { cout << *it << " "; }
 cout << endl;
 vNumbers.clear();
 cout << "vNumbers.size()=" << vNumbers.size() << endl;
 return 0; }
```

vector

```
// Compute the average and standard deviation of an input set of grades.
#include <fstream>
#include <iostream>
#include <vector> // to access the STL vector class
#include <cmath> // to use standard math library and sqrt
int main(int argc, char* argv[]) {
 if (argc != 2) { std::cerr << "Usage: " << argv[0] << " grades-file\n"; return 1; }
 std::ifstream grades_str(argv[1]);
 if (!grades_str) { std::cerr << "Can not open the grades file " << argv[1] << "\n";
  return 1; }
 std::vector<float> scores; // Vector to hold the input scores; initially empty.
 float x; // Input variable
 while (grades_str >> x) { scores.push_back(x); }
 if (scores.size() == 0) { std::cout << "No scores entered. Please try again!" <<
std::endl; return 1; // program exits with error code = 1 }
 float sum = 0;
 for (unsigned int i = 0; i < scores.size(); ++ i) { sum += scores[i]; }
 double average = double(sum) / scores.size();
 std::cout<<"The average of "<<scores.size()<<" grades is " << average << std::endl;
 double sum_sq_diff = 0.0;
 for (unsigned int i=0; i<scores.size(); ++i) { double diff = scores[i] - average;
  sum_sq_diff += diff*diff; }
 double std_dev = sqrt(sum_sq_diff / (scores.size()-1));
 std::cout << "The standard_deviation of " << scores.size() << " grades is " <<
 std_dev << std::endl; return 0; // everything ok
}
```

list

- contenitore efficiente per operazioni di inserimento ed eliminazione in posizioni intermedie del container. Supporta iteratori bidirezionali.
- la classe list fornisce funzioni specifiche:

push_front(const value_type& val), **pop_front()**

remove(const T& value), rimuove tutte le occorrenze di un elemento

unique(), cancella gruppi di elementi consecutivi uguali tranne il primo elemento di ciascun gruppo

list.merge(list& x), fonde x in list trasferendo tutti gli elementi di x in modo ordinato (**entrambe le liste "list" e "x" devono essere già ordinate in modo crescente**)

reverse() inverte una lista

sort() per ordinamento crescente

list

```
#include <iostream>
#include <string>
#include <list>
using namespace std;
class alunno{
  public:
   string getMatricola(){return matr;};
   string getNome(){return nome;};
   void putNuovo(string mt, string nm) {matr=mt;nome=nm;};
  private: string matr; string nome; };

 void main(){
  list<alunno> elenco;
  alunno temp; string m,n;
  // inserisce alunni in elenco
  while(true){
   cout << "Matricola alunno ";
   getline(cin,m);
   if(m=="") break;
   cout << "Nome alunno ";
   getline(cin,n);
   temp.putNuovo(m,n);
   elenco.push_back(temp); }
  for (list<alunno>::iterator it= elenco.begin(); it != elenco.end(); it++) {
    cout << (*it).getNome() << " "; };
 cout << endl;
 }
```

list

```
#include <iostream> #include <list> #include <algorithm>
using namespace std;
template < class T > void printList( list< T > &listRef );
int main() { list< int > values, otherValues;
values.push_front( 1 ); values.push_front( 2 );
values.push_back( 4 ); values.push_back( 3 );
cout << "values contiene: "; printList( values );
values.sort();
cout << "\nvalues dopo sort() contiene: "; printList( values );
otherValues.insert( otherValues.begin(), values.begin(), values.end() );
cout << "\notherValues contiene: "; printList( otherValues );
values.merge( otherValues );
cout << "\nDopo merge():\n values contiene: "; printList( values );
cout << "\n otherValues contiene: "; printList( otherValues );
values.pop_front(); values.pop_back();
cout << "\nDopo pop_front() e pop_back() values contiene:\n";
printList( values );
values.unique();
cout << "\nDopo unique() values contiene: "; printList( values );
values.swap( otherValues );
cout << "\nDopo swap:\n values contiene: "; printList( values );
cout << "\n otherValues contiene: "; printList( otherValues );
values.assign( otherValues.begin(), otherValues.end() );
cout << "\nDopo assign() values contiene: "; printList( values );
```

list

```
values.merge( otherValues );
cout << "\nvalues contiene: "; printList( values );
values.remove( 4 );
cout << "\nDopo remove( 4 ) values contiene: "; printList( values );
cout << endl;

list<int>::iterator it= values.begin();
it++;
it=values.insert(it, 10);
values.insert(it, 6);
cout << "values contiene: ";
printList(values);


return 0; }

template < class T > void printList( list< T > &listRef ) {
  if ( listRef.empty() )
     cout << "La list e' vuota";
  else { for (typename list<T>::iterator it = listRef.begin(); it != listRef.end();
it++) {
       cout << *it << " ";
} } }
```

deque

- Per frequenti operazioni di inserimento ed eliminazione **ad entrambi** i capi di un container
- il termine deque è una forma abbreviata per “double-ended queue”, coda a due estremi o "coda bifronte"
- La classe deque fornisce le stesse operazioni di base della classe vector, e vi aggiunge le funzioni membro push_front() e pop_front() rispettivamente per l’inserimento e l’eliminazione all’inizio del deque
- Consente accesso casuale agli elementi con indice
- Differenza con list: una list può effettuare inserimenti/eliminazioni efficienti nelle posizioni intermedie

deque

```
#include <iostream>
#include <deque>
using namespace std;
int main() {
  deque< double > values;
  values.push_front( 2.2 );
  values.push_front( 3.5 );
  values.push_back( 1.1 );
  cout << "values contiene: ";
  for ( int i = 0; i < values.size(); ++i )
    cout << values[ i ] << ' ';
  values.pop_front();
  cout << "\nDopo pop_front() values contiene: ";
  for (int i = 0; i < values.size(); ++i)
    cout << values[i] << ' ';
  values[ 1 ] = 5.4;
  cout << "\nDopo values[ 1 ] = 5.4, values contiene: ";
  for (int i = 0; i < values.size(); ++i)
    cout << values[i] << ' ';
  cout << endl;
  return 0;
}
```

Adapters: queue

- Metodo pubblico **size()** per ottenere il numero di elementi presenti
- Nessun iteratore supportato
- Metodo **front()** per ottenere elemento in testa alla coda
- Metodo **pop()** rimuove elemento in testa ma non ritorna una copia

```
#include <iostream>
using namespace std;
#include <queue>
int main() {
 queue<int> myQueue;
 myQueue.push(1);
 myQueue.push(7);
 myQueue.push(4);
 cout << myQueue.size() << endl;
 while (!myQueue.empty()) {
   cout << myQueue.front() << endl;
   myQueue.pop(); }
}
```

Adapters: stack

```
#include <iostream>
using namespace std;
#include <stack>
int main() {
stack<int> s;
s.push(11);
s.push(33);
s.push(26);
s.push(50);
s.push(2);
cout << "s.size(): " << s.size() << endl;
cout << "s.top(): " << s.top() << endl;
s.pop();
cout << "s.size(): " << s.size() << endl;
cout << "s.top(): " << s.top() << endl;
return 0; }
```

Adapters: priority queue

- Metodo pubblico **top()** per accedere all'elemento a priorità maggiore
- Metodo pubblico **size()** per ottenere il numero di elementi presenti

```
#include <iostream>
#include <string>
using namespace std;
#include <queue>
class Task {
 public: Task(int _priority, string _task);
         bool operator<(const Task&) const;
         int priority; string task; };
Task::Task(int _priority, string _task) : priority(_priority), task(_task) {};
bool Task::operator<(const Task& right) const { return priority < right.priority; }

int main() {
 priority_queue<Task> prioQueue;
 prioQueue.push(Task(4, "Go to the cinema"));
 prioQueue.push(Task(9, "Solve the programming assignments"));
 prioQueue.push(Task(3, "Washing up"));
 while (!prioQueue.empty()) {
   cout << prioQueue.top().task << endl;
   prioQueue.pop(); }
}
```

Container associativi

- progettati per fornire un accesso diretto nelle operazioni di memorizzazione e di recupero di elementi tramite chiavi di ricerca.
- quattro container associativi: multiset, set, multimap, map.
- In ogni container le chiavi sono mantenute ordinate.
- L'attraversamento di un container associativo avviene secondo l'ordinamento previsto su tale container (per una map richiede definizione dell'operatore < sulla chiave).
- Le classi multiset e set forniscono operazioni per la manipolazione di insiemi di valori in cui i valori sono le chiavi, ovvero non c'`e separazione tra valori e chiavi associate.
- La differenza tra un multiset e un set è che un multiset permette chiavi duplicate mentre un set no.

Container associativi

- Le classi multimap e map forniscono operazioni per la manipolazione dei valori associati alle chiavi.
- La principale differenza tra un multimap e un map è che un multimap permette chiavi duplicate e un map permette soltanto chiavi univoche.
- Funzione insert per inserire un nuovo elemento:

  insert (const value_type& val);

- Oltre alle funzioni membro comuni a tutti i container, tutti i container associativi supportano anche funzioni membro tra cui

find(), ritorna iteratore al primo elemento trovato, altrimenti ritorna end()

lower_bound(const value_type& val), ritorna iteratore al primo elemento non inferiore a val

upper_bound(const value_type& val), ritorna iteratore al successivo di val

count (const value_type& val), ritorna il numero di elementi uguali a val

set

```
#include <set>
#include <iostream>
using namespace std;
int main() {
 set <int> mySet;
 set<int>::iterator it;
 mySet.insert(4); mySet.insert(2); mySet.insert(4); mySet.insert(1); mySet.insert(4);
 for ( it = mySet.begin(); it != mySet.end(); it++ ) {
   cout << *it << endl; }
 cout << "s.count(5)=" << mySet.count(5) << endl; // number of elements == 5
 std::set<int>::iterator itlow, itup;
 mySet.clear();
 for (int i = 1; i<10; i++) mySet.insert(i * 10); // 10 20 30 40 50 60 70 80 90
 itlow = mySet.lower_bound(30);                   // ^
 itup = mySet.upper_bound(60);                                 // ^
 mySet.erase(itlow, itup); // 10 20 70 80 90
 it = mySet.find(20);
 mySet.erase(it);
 mySet.erase(mySet.find(80));
 std::cout << "mySet contains:";
 for (std::set<int>::iterator it = mySet.begin(); it != mySet.end(); ++it)
  std::cout << ' ' << *it;
 std::cout << '\n';
 return 0;
}
```

map

- Funzione find (const key_type& k) per ricercare un elemento
- Uso della classe std::pair per inserire coppie <chiave,valore>
- std::pair è un contenitore di una coppia di oggetti: il primo elemento è chiamato ‘’first’’, il secondo elemento è chiamato ‘’second’

```
#include <iostream>
using namespace std;
#include <map>
int main() {
 map <int, double> myMap; map<int, double>::iterator it;
 typedef pair<int,double> MyPair;
 myMap.insert(MyPair(1, 4.7));
 myMap.insert(MyPair(3, 8.5));
 myMap.insert(MyPair(4, 9.0));
 myMap[6] = 2.5;
 cout << "myMap[4] = " << myMap[4] << endl;
 for ( it = myMap.begin(); it != myMap.end(); it++) {
   cout << (*it).first << " " << (*it).second << endl; }
 it = myMap.find(4);
 if (it != myMap.end())
   cout << "element with key 4 : "<< (*it).second << endl;
 else cout << "Key not found." << endl; }
```

algoritmi

- STL fornisce algoritmi che possono essere utilizzati in modo generale su diversi tipi di container
- Gli algoritmi di STL riguardano: inserimento, eliminazione, ricerca, ordinamento e altro
- STL include all'incirca 70 algoritmi standard
- Molti algoritmi operano su sequenze di elementi definite da coppie di iteratori, in cui il primo iteratore punta al primo elemento della sequenza e il secondo punta all'elemento successivo all'ultimo della sequenza.
- La funzione membro begin() restituisce un iteratore al primo elemento di un container; end() restituisce un iteratore alla prima posizione dopo l'ultimo elemento del container.
- Gli algoritmi restituiscono spesso iteratori.

algoritmi

- Un algoritmo come find(), per esempio, trova un elemento e restituisce un iteratore a tale elemento.
- Se l'elemento cercato non viene trovato, find() restituisce l'iteratore end()
- verificare se il risultato di find() è uguale a end() è un tipico modo per verificare se la ricerca ha avuto esito positivo.
- L'algoritmo find() può essere utilizzato con tutti i container della STL.
- Gli algoritmi operano sugli elementi dei container in modo indiretto tramite gli iteratori (gli algoritmi non sono funzioni membro dei container).

algoritmi

Algoritmi di modifica del contenuto di container		
copy()	remove()	reverse_copy()
copy_backward()	remove_copy()	rotate()
fill()	remove_copy_if()	rotate_copy()
fill_in()	remove_if()	stable_partition()
generate()	replace()	swap()
generate_n()	replace_copy ()	swap_ranges()
iter_swap()	replace_copy_if()	transform()
partition()	replace_if()	unique()
random_shuffle()	reverse()	unique_copy()

Algoritmi di modifica del contenuto di container

Algoritmi che non modificano il contenuto dei container		
adjacent_find()	equal()	mismatch()
count()	find()	search()
count_if()	for_each()	search_n()

Algoritmi che non modificano il contenuto dei container

Algoritmi: esempi

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
int main (int, char * []) {
  vector<int> v;
  v.push_back(1);
  v.push_back(2);
  v.push_back(3);
  v.push_back(2);
  int i = 7;
  cout << i << " appears " << count(v.begin(), v.end(), i) << " times in v"
<< endl;
  i = 2;
cout << i << " appears " << count(v.begin(), v.end(), i) << " times in v"
<< endl; return 0;
}
```

Algoritmi: esempi

```
// count_if example
#include <iostream> // std::cout
#include <algorithm> // std::count_if
#include <vector> // std::vector
bool IsOdd (int i) { return ((i%2)==1); }

int main () {
 std::vector<int> myvector;
 for (int i=1; i<10; i++) myvector.push_back(i);
 // myvector: 1 2 3 4 5 6 7 8 9
 int mycount = count_if (myvector.begin(), myvector.end(), IsOdd);
 std::cout << "myvector contains " << mycount << " odd values.\n";
 return 0;
}
```

Algoritmi: esempi

- algoritmo sort() per contenitori con iteratori ad accesso casuale (vector)
- sort() non può essere usato su contenitori list, che hanno un loro metodo specifico con lo stesso nome (sort)

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
int main () {
vector<int> v;
v.push_back(1);
v.push_back(22);
v.push_back(33);
v.push_back(12);
sort (v.begin(), v.end()); // comparazione tramite operatore <
for (vector<int>::iterator it = v.begin(); it != v.end(); it++)
 {
   cout << *it << " ";
 }
return 0;
}
```

Algoritmi: esempi

```
#include <vector>
#include <algorithm>
using namespace std;
#include <iostream>
#include <string>
int main (int argc, char *argv[]) {
  std::vector <std::string> projects;
  for (int i = 1; i < argc; ++i)
   projects.push_back (std::string (argv [i]));
  std::vector<std::string>::iterator j = std::find (projects.begin (),
  projects.end (), std::string ("Lab8"));
  if (j == projects.end ()) cout<< "Lab8 not found" << endl;
  else cout << "Lab8 found" << endl;
  return 0;
}
```

Algoritmi: esempi

- Le funzioni fill() e fill_n() impostano un intervallo di elementi di un container a un valore specifico.

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
int main() {
vector< char > chars( 10 );
fill( chars.begin(), chars.end(),'5' );
cout << "Il vettore chars dopo fill() con 5:\n";
for (vector< char >::iterator it = chars.begin(); it != chars.end();
it++) { cout << *it << " ";} cout << endl;
fill_n( chars.begin(), 5, 'A' );
cout << "Il vettore chars dopo fill_n() di cinque elementi uguali ad
A:\n";
for (vector< char >::iterator it = chars.begin(); it != chars.end();
it++) { cout << *it << " "; }
cout << endl;
return 0; }
```

Algoritmi: esempi

```
#include <iostream>
#include <string>
#include <algorithm>
#include <vector>
using namespace std;
class clibro{
  public: clibro(string t, string a, float p);
  string getTitolo() const {return titolo;};
  string getAutore(){return autore;};
  float getPrezzo(){return prezzo;};
  friend ostream& operator<<(ostream& output, const clibro& l);
  bool operator==(const clibro &t) const;
private:
  string titolo;
  string autore;
  float prezzo; };

clibro::clibro(string t, string a, float p){
  titolo = t; autore = a; prezzo = p; }

ostream& operator<<(ostream& output, const clibro& l){
  output << l.titolo << ' ' << l.autore << ' '<< l.prezzo << endl; return output; };

bool clibro::operator==(const clibro &t) const{ return (titolo== t.getTitolo()); };
```

Algoritmi: esempi

```
bool ordT(clibro l1,clibro l2){return l1.getTitolo()<l2.getTitolo();};
bool ordP(clibro l1,clibro l2){return l1.getPrezzo()<l2.getPrezzo();};

void ordina(vector<clibro>& vl){
  int scelta;
  cout << "Ordina per titolo(1) o prezzo (2)" << endl;
  cin >> scelta;
  if(scelta==1)
     sort(vl.begin(),vl.end(),ordT); // comparazione tramite funzione booleana
  if(scelta==2)
     sort(vl.begin(),vl.end(),ordP); };

void stampa(const vector<clibro>& vl) {
    vector<clibro>::const_iterator i;
    for (i = vl.begin(); i != vl.end(); i++) cout << *i; };

int main() {
 vector<clibro> vLibri;
 clibro libro1("autore1","titolo1",15.10); clibro libro2("autore2", "titolo2", 10.0);
 clibro libro3("autore3", "titolo3", 11.0); clibro libro4("autore4", "titolo4", 21.0);
 vLibri.push_back(libro1); vLibri.push_back(libro2); vLibri.push_back(libro3);
 vLibri.push_back(libro1);
 ordina(vLibri); stampa(vLibri);
 replace(vLibri.begin(), vLibri.end(), libro1, libro4);
 cout << "vLibri after replace:" << endl; stampa(vLibri); return 0; }
```

Regola per uso libreria STL in esami

- Nello svolgimento degli esami è consentito utilizzare le classi contenitore della libreria STL (ed eventualmente i relativi algoritmi) **SOLAMENTE** quando espressamente indicato nel testo dell'esame. Quando non viene indicato esplicitamente che è possibile utilizzare le classi della libreria standard STL occorre SEMPRE utilizzare oggetti delle classi base che abbiamo decritto nel corso (es: LList, BST, PQ …).