Introduzione alla C++ Standard Library

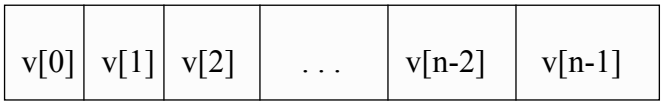
- Contenitori
  - Un insieme di Template di Classe che implementano le collezioni piu' comuni (lista, vettore, insieme, coda, etc.)
- Iteratori
  - Un mezzo generico per accedere agli elementi di un oggetto contenitore
- Algoritmi Generici
  - Un insieme di funzioni per realizzare le operazioni più comuni su oggetti contenitore

C++ Standard Library
Classi Contenitore

- Vector
- List
- Deque
- Stack
- Queue
- Set
- Priority Queues
- Map (dizionari)

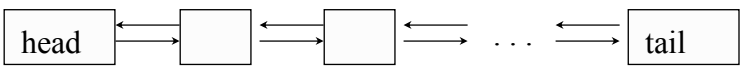
Vector

- Generalizzazione di un array
- Efficiente nell'accesso casuale agli elementi
- Le dimensioni del vettore crescono dinamicamente



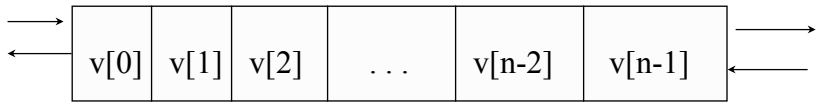
List

- Contenitore di dimensione arbitraria, molto efficiente se la dimensione cambia frequentemente
- Permette solo accesso sequenziale agli elementi (a partire dalla prima o dall'ultima posizione)
- Inserimenti e rimozioni sono efficienti in qualsiasi posizione



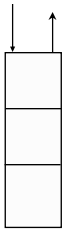
Deque

- Cresce e si riduce dinamicamente
- Inserimenti e rimozioni efficienti sia in testa che in coda
- Permette anche l'accesso casuale agli elementi

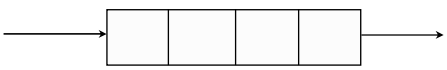


Stack e Queue

- Rappresentano una specializzazione di *deque*
- *Stack* adotta una politica LIFO (Last In, First Out)



- *Queue* adotta una politica FIFO (First In, First Out)



Set

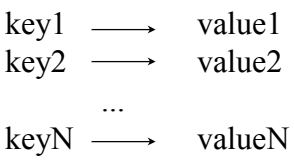
- Collezione Ordinata
- Inserimento, rimozione e test di inclusione sono molto efficienti (logaritmico)
- Efficiente implementazione di operazioni insiemistiche (es: unione, differenza, …)
- *Multiset*: come *Set*, ma permette più di un elemento con lo stesso valore

Priority Queue

- Efficiente (logaritmico) nell'inserimento di nuovi elementi
- Molto efficiente per l'accesso e la rimozione del più piccolo (o del più grande) valore nella collezione (O(c) e O(log n) rispettivamente)

Map (dizionario)

- Collezione di coppie chiave-valore
- Le chiavi possono essere qualsiasi tipo di dato su cui e' possibile definire una relazione d'ordine (ad es. string, int, etc.)
- I valori possono essere di qualsiasi tipo di dato
- Efficiente per inserimento, rimozione, e test di inclusione (sulle chiavi)



Vector

```
#include <vector>
vector<int> vecOne(10,0);
  // crea un vettore con 10 int
  // inizializzati a 0
```

- Altri costruttori. Se la dimensione non viene specificata il vettore inizialmente non contiene nessun elemento e cresce automaticamente quando gli elementi vengono aggiunti.

```
vector<int> vecTwo(5, 3);
vector<int> vecThree;
```

- Il costruttore di copia permette di creare un clone di un vettore da un altro vettore

```
vector<int> vecFour(vecTwo);
```

Vector: inizializzazione

- Un vettore può anche essere inizializzato usando elementi di un'altra collezione (ad esempio di una lista) per mezzo di una coppia di *iteratori*.

```
vector<int> vecFive(aList.begin(),aList.end());
```

Vector: assegnazione

- Per tutti i contenitori è ridefinita l'assegnazione:

```
vecThree = vecFive;
```

- Tuttavia può tuttavia essere più efficiente usare il metodo `assign()`

```
vecFour.assign(3, 7); // 3 copie di 7
vecFive.assign(12); // 12 copie di 0
vecSix.assign(listTen.begin(),
  listTen.end());
```

- I valori di due vettori possono essere scambiati con il metodo `swap()`:

```
vecThree.swap(vecFour);
```

Vector: accesso agli elementi

- Per accedere agli elementi di un oggetto vector si può usare l'operatore di subscript []

```
cout << vecFive[1] << endl;
vecFive[1] = 15;
```

- Il metodo `at()` può essere usato al posto dell'operatore di subscript.
- Il metodo `front()` restituisce il primo elemento nel vettore vector; il metodo `back()` l'ultimo.

```
cout << vecFive.front() << " ... " <<
  vecFive.back() << endl;
```

Vector: dimensione e capacità

- Per ogni vettore abbiamo:
  - size(): numero di elementi nel vettore
  - capacity(): massima dimensione a cui il vettore può essere espanso senza richiedere un nuova allocazione di memoria
  - max_size(): limite superiore della dimensione di qualsiasi vettore.

```
cout<<"size: " << vecFive.size() << endl;
cout<<"capacity: " << vecFive.capacity() <<
  endl;
cout<<"max_size: " << vecFive.max_size() <<
  endl;
```

Vector: inserimenti

- Un nuovo elemento può essere aggiunto alla coda di un vettore mediante il metodo push_back(). Se c'è spazio nella corrente allocazione, questa operazione è molto efficiente.

```
vecFive.push_back(21);
// aggiunge 21 alla fine del vettore
```

- Il corrispondente metodo di rimozione è pop_back()

```
vecFive.pop_back();
```

Vector: Inserimenti

- Operazioni di inserimento possono essere fatte anche con il metodo insert(). La posizione di un inserimento è specificata mediante un iteratore. L'inserimento avviene nella posizione che immediatamente precede la locazione specificata.

```
  // find the location of the 7
vector<int>::iterator where =
find(vecFive.begin(), vecFive.end(),7);
  // then insert the 12 before the 7
vecFive.insert(where, 12);
vecFive.insert(where, 6, 14);
  // insert six copies of 14
```

Vector: Inserimenti

- La forma più generale del metodo insert() ha come argomenti una posizione e una coppia di iteratori che denotano una sotto-sequenza da un altro contenitore. L'intervallo di valori descritto dalla sequenza è inserito nel vettore.

```
vecFive.insert (where,
  vecThree.begin(), vecThree.end());
```

- Per la rimozione si usa il metodo erase(), che funziona in maniera del tutto analoga a insert():

```
vecFive.erase(where);
vecFive.erase(where, vecFive.end());
```

Vector: test di inclusione

- Gli algoritmi generici possono essere usati per operazioni utili
- Esempio: la seguente istruzione permette di verificare se un vettore di interi contiene il valore 5:

```
count(vecFive.begin(),vecFive.end(),5);
```

Vector: ordinare

- Per ordinare gli elementi di un vettore si può usare l'algoritmo generico sort(). Nella sua forma più semplice sort() usa l'operatore less-than:

```
sort(aVec.begin(), aVec.end());
```

- E' anche possibile specificare quale operatore deve essere usato

```
sort(aVec.begin(),aVec.end(),greater<int>());
```

- Un altro esempio di utilizzo degli algoritmi generici:

```
vector<int>::iterator where;
where = max_element(vec_five.begin(),
  vec_five.end());
cout << "Il Massimo vale: " << *where << endl;
```

Iteratori

- **Problema:** come permettere l'accesso agli elementi di una collezione senza sapere come la collezione e' organizzata?
- **Soluzione:** definire un nuovo tipo di dato specificatamente per accedere agli elementi di un contenitore
- La C++ Standard Library fornisce un gran numero di algoritmi: tutti quanti possono operare sui contenitori, accedendo agli elementi attraverso iteratori

Iteratori

- Un iteratore rappresenta un puntatore ad un elemento del contenitore.
- Ciascun contenitore ha associata la propria classe iteratore.

```
Vector<int>::iterator where;
```

- Tutti i contenitori hanno i metodi **begin()** e **end()** che ritornano iteratori al primo e all'ultimo elemento della collezione rispettivamente.

```
random_shuffle (aVector.begin(),
                aVector.end());
```

Iteratori

- Un iteratore può essere incrementato usando l'operatore ++: in questo modo punta al prossimo elemento nella sequenza.
- Un iteratore può essere comparato con un altro iteratore. Risultano uguali quando puntano allo stesso elemento.
- Un iteratore può essere deferenziato usando l'operatore *, permettendo cosi di ottenere il valore denotato dall'iteratore.

Iteratori

```
vector<int> aVec(10);
vector<int>::iterator scan;

for(scan=aVec.begin();
     scan !=aVec.end() ; scan++ )
   cout << *scan;
```

Iteratori e Algoritmi Generici

```
#include <algorithm>
```

Trovare valori in un array,

```
int data[100];
int* where;
where = find(data, data+100, 7);
```

o in una lista (o in qualsiasi altro contenitore):

```
list<int>::iterator where;
where=find(aList.begin(),aList.end(),7);
```

Scegliere un Contenitore

In che modo si accederà ai valori?

- random ⇒ vector o deque
- ordinato ⇒ set o map
- sequenziale ⇒ list

L'ordinamento dei valori nella collezione è rilevante?

- SI ⇒ set
- Può essere ordinata ⇒ vector or deque
- Dipende dall'inserimento ⇒ stack o queque

Scegliere un Contenitore

La dimensione del contenitore varierà significativamente ?

- SI ⇒ *list* o *set*
- NO ⇒ *vector* o *deque*

E' possibile stimare la dimensione della collezione?

- SI ⇒ *vector*

Scegliere un contenitore

Il test di inclusione è un'operazione frequente?

- SI ⇨ *set*

La collezione è indicizzata? Ovvero, può essere vista come una collezioni di coppie chiave/valore?

- Se gli indici sono interi, può convenire usare *vector* o *deque*
- Altrimenti *map*

Scegliere un contenitore

Quali sono le posizioni in cui gli elementi verranno inseriti (o da cui verranno rimossi) più frequentemente?

- Inserimenti in posizioni casuali sono efficienti con *list*, mentre non lo sono con *vector*
- *Stack* e *Queue* permettono solo inserimenti in coda

Scegliere un Contenitore

Se la fusione di due o più collezioni in una è un operazione frequente:

- Se la collezione è ordinata conviene usare *set*
- Altrimenti *list* (e concatenare)

Se cercare e rimuovere il più grande (piccolo) valore della collezione è un'operazione frequente

- Può convenire l'uso di una *priority queue* (che risulta leggermente più efficiente di *set*)