# STL

- **Breve introduzione alla Standard Template Library del C++**
- **Analisi di semplici esempi**

# STL: Standard Template Library

- Sviluppate da Alexander Stepanov e Meng Lee presso la Hewlett-Packard
- Libreria molto estesa  inclusa nello standard ANSI/ISO nel 1995
- Template di classi disponibili in libreria (utilizzabili includendo opportuni file)
- Implementano potenti  strutture dati di base; forniscono circa 70 potenti algoritmi su tali strutture
- Realizzano  "contenitori" che offrono specifici servizi di gestione dei dati in essi contenuti…
- Realizzano tipi speciali come stringhe, numeri complessi, stream, …

# Componenti principali

- **Contenitori** : strutture dati di base
- **Iteratori** : utilizzati per accedere ai singoli dati dei contenitori
- **Algoritmi generici** : applicabili ai dati contenuti nei contenitori

In più, **TIPI** nuovi, come `string` e `complex`, da usare come i tipi di base del C.

Iniziamo dal tipo `string...`



# Una classe speciale: string

- Classe per gestire sequenze di caratteri
- Pensata per essere utilizzata come un tipo base del C++
- Caratteristiche:
  - Facile dichiarazione. Es: `string s = "casa";`
  - Concatenazione e assegnazione. Es: `s = s + " dolce";`
  - Operatori di confronto. Es: `==, !=, <, <=, >, >=`
  - Metodi per la gestione. Es: `length(), insert(), erase(), replace(), find(),...`
- Vedere esempio (`String.cpp`)

# Tipi di contenitori

- Contenitori sequenziali
  - VECTOR (vettori a lung. variabile)
  - LIST (liste)
  - DEQUE (code a doppio ingresso)

**Organizzano gli elementi in modo lineare. L'ordine degli elementi dipende dal tipo di inserimento (all'inizio, alla fine)**

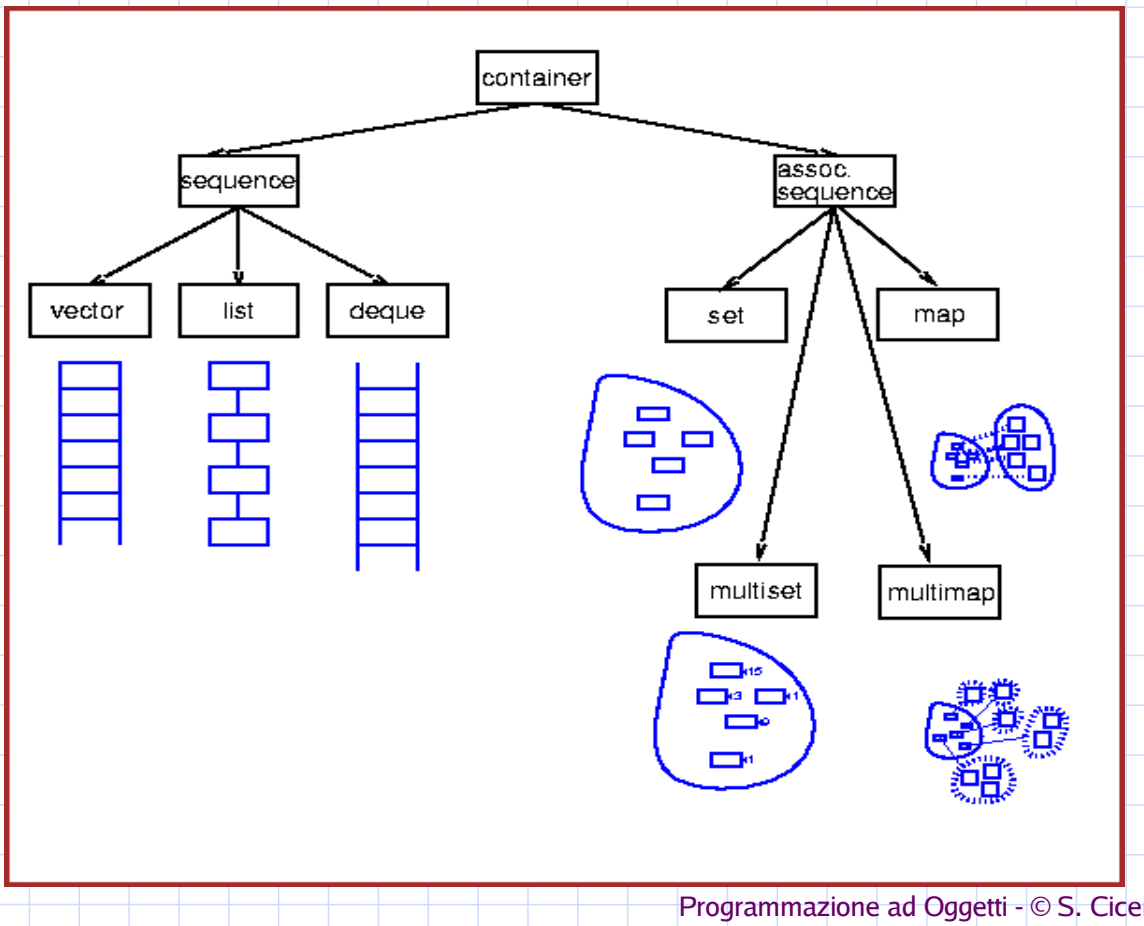
- Contenitori ordinati associativi
  - MAP (funzioni: chiave-valore)
  - MULTIMAP (relazioni: più chiavi-valore)
  - SET (insiemi)
  - MULTISET (multinsiemi)

**Contenitori non lineari. Un ordinamento può essere indotto dai valori degli dati per un recupero efficiente dei dati stessi.**

**set e multiset possono essere visti come degenarazioni di map e multimap dove vengono mantenute solo le chiavi**



# Tipi di contenitori: *grafica*



# Vector: esempio d'uso

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

int main () {
  vector<int> seq;          //vettore di interi

  for (int i=1; i<=6; ++i)  //inserisce i valori da 1 a 6
    seq.push_back(i);

  for (int i=0; i<seq.size(); ++i) //stampa gli elementi
    cout  <<  seq[ i ]  << " ";

 cout  <<  endl;
}
```



# List: esempio d'uso

```
#include <iostream>
#include <list>
using namespace std;

int main(){
   list<char> seq;      // Lista di char

   for (char c='a'; c<='z'; ++c) // inserimento elementi
      seq.push_back(c);          // dalla 'a' alla 'z'

   while (! seq.empty()) {       //stampa ed eliminazione degli elementi
      cout << seq.front() << ' ';
      seq.pop_front();
   }
   cout << endl;
}
```

# Proprietà

**container sequenziali**

vector inserimenti/eliminazioni rapidi in coda; accesso diretto a qualsiasi elemento

list lista a doppio concatenamento; inserimenti ed eliminazioni rapidi ovunque

deque inserimenti/eliminazioni rapidi in testa o in coda; accesso diretto a qualsiasi elemento

**container associativi**

set ricerca rapida; non sono consentiti duplicati

multiset ricerca rapida; sono consentiti duplicati

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

multimap mapping uno-a-uno; sono consentiti duplicati; ricerca rapida di una chiave

**adattatori di container**

stack l'ultimo inserito è il primo estratto: last-in-first-out (LIFO)

queue il primo inserito è il primo estratto: first-in-first-out (FIFO)

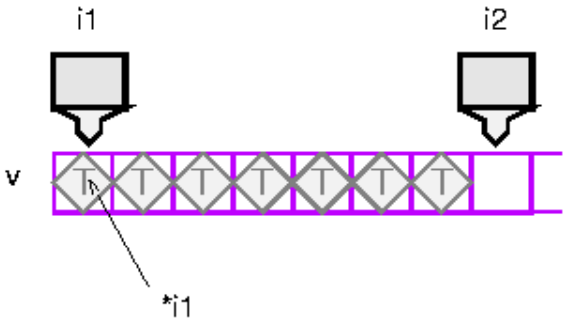
priority_queue l'elemento di priorità più alta è sempre il primo elemento estratto



# Iteratori: oggetti che …

- permettono di accedere agli elementi dei container
- permettono di eseguire algoritmi generici su ogni tipo di container
- sono implementati in modo differente per ciascun tipo di container
- alcune operazioni degli iteratori hanno lo stesso significato su tutti i container.

```
vector<int> v;
... // 7 volte push_back()
vector<int>::iterator
        i1 = v.begin(),
        i2 = v.end();
cout << *i1;
sort(i1,i2);
```



# Iteratori: operazioni fondamentali

- Operator *

  restituisce l'elemento a cui l'iteratore fa riferimento (es: *i Se *i ha membri, si può usare ->, es: i->a )

- Operator ++

  l'iteratore passa al prossimo elemento, es: i++. Molti iteratori hanno anche l'operatore --, es: i-- .

- Operator == e !=

  testano se due iteratori si riferiscono allo stesso elemento o meno.

- Operator =

  Assegna ad un iteratore la posizione di un elemento.



# Iteratori: esempio d'uso su list

```
#include <iostream>
#include <list>
using namespace std;

int main(){
   list<char> seq;      // Lista di char

   for (char c='a'; c<='z'; ++c) // inserimento elementi
      seq.push_back(c);          // dalla 'a' alla 'z'

   list<char>::iterator i;          //dichiarazione dell'iteratore
   for ( i = seq.begin(); i != seq.end(); i++ )  //stampa
      cout <<  *i << "  ";

   cout << endl;
}
```

# Tipo di iteratore supportato

**container sequenziali**

| | | |
|---|---|---|
| Vector | ad accesso casuale (con “aritmetica dei puntatori”) | |
| Deque | ad accesso casuale | |
| List | bidirezionale | (solo operatori ++ e -- ) |

**container associativi**

| | |
|---|---|
| Set | bidirezionale |
| Multiset | bidirezionale |
| Map | bidirezionale |
| Multimap | bidirezionale |

**adattatori di container**

| | |
|---|---|
| Stack | nessun iteratore supportato |
| Queue | nessun iteratore supportato |
| Priority_queue | nessun iteratore supportato |



# Typedef per gli iteratori su contenitori

| Typedef | Direzione di ++ | Funzionalità |
|---|---|---|
| iterator | avanti | lettura/scrittura |
| const_iterator | avanti | lettura |
| reverse_iterator | indietro | lettura/scrittura |
| const_reverse_iterator | indietro | lettura |

# VECTOR

- Contenitore sequenziale
- Lista di oggetti specificati dall'utente
- Caratteristiche:
  - Oggetti mantenuti in zone di memoria adiacenti
  - Accesso random in lettura
  - Inserimento e cancellazione alla fine in tempo (ammortato) costante
  - Possibili inserimenti e cancellazioni in altre posizioni, ma richiedono lo spostamento a destra o a sinistra di elementi successivi; quindi, tempo lineare *O(n)*
  - Ricerca in tempo lineare *O(n)*
- Vedere esempio (`testVector.cpp`)



# LIST

- Contenitore sequenziale
- Lista di oggetti specificati dall'utente
- Caratteristiche:
  - Gli oggetti non sono mantenuti in zone di memoria adiacenti
  - Inserimento in tempo costante *O(1)* (sia all'inizio, alla fine, che in mezzo)
  - Ricerca in tempo lineare *O(n)*
- Vedere esempio (`testList.cpp`)

# MAP

- Contenitore associativo
- Memorizza oggetti formati da coppie di valori (K,T): K è una chiave, T è l'elemento associato alla chiave K
- Caratteristiche:
  - Oggetti mantenuti ordinati rispetto alla chiave
  - Permette di trovare velocemente l'oggetto T associato ad una chiave K
  - Senza ripetizioni: non possono esserci due oggetti aventi la stessa chiave
  - Accesso in tempo logaritmico *O(log n)*
- Vedere esempio (`testMap.cpp`)



# Algoritmi generici

- La STL fornisce algoritmi che possono essere utilizzati in modo generale su diversi tipi di container.
- Tali algoritmi sono comunemente utilizzati nella manipolazione dei container.
- Gli algoritmi operano sugli elementi dei container solo indirettamente tramite gli iteratori.
- 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.
- Ad esempio, l'algoritmo `find()` trova un elemento e restituisce un iteratore a tale elemento. Se l'elemento cercato non viene trovato, `find()`restituisce l'iteratore `end()`. L'algoritmo `find()`può essere utilizzato con tutti i container della STL.

# Algorimi di modifica

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