Corso di Programmazione Orientata agli oggetti

Standard Template Library (STL)

a.a. 2020/2021
Francesco Fontanella

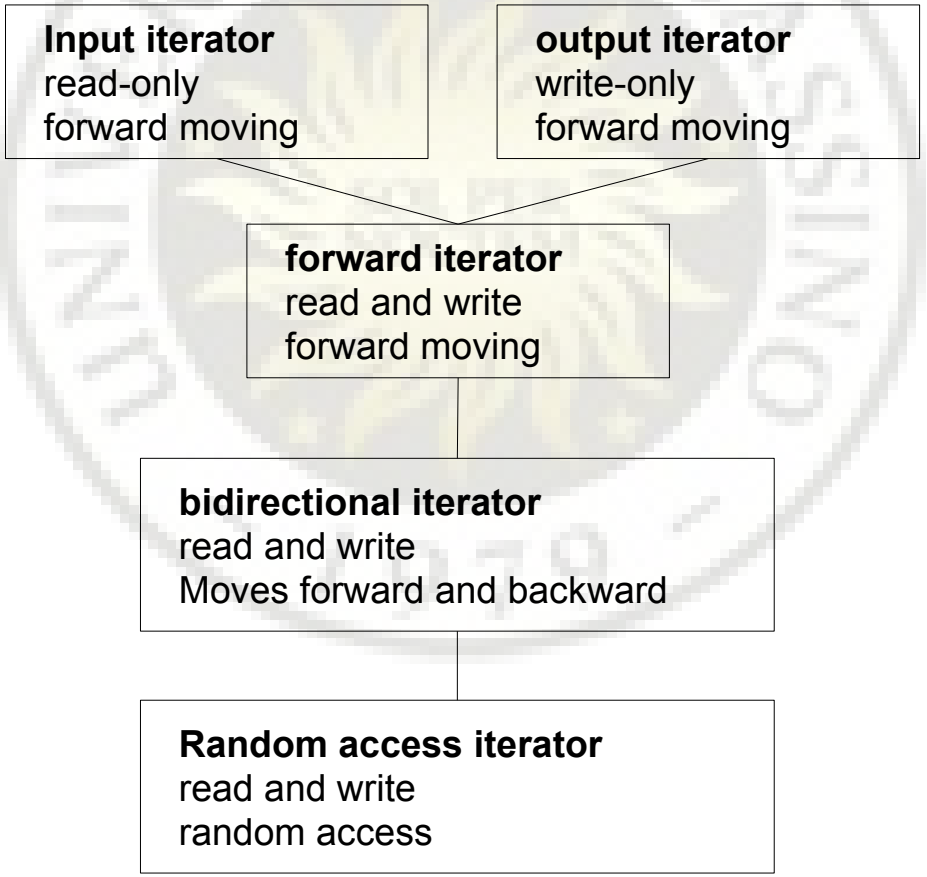
Standard Template Library (STL)

- Libreria di classi di:
  - Contenitori: sono essenzialmente delle ADT
  - Iteratori: puntatori *intelligenti*
  - Algoritmi: find, sort, ecc

- Generica:
  - tutti i suoi componenti sono parametrizzati mediante l'utilizzo dei template

Gli iteratori

- Sono una generalizzazione dei puntatori che consentono di interagire in maniera uniforme con i diversi contenitori della STL
- Gli iteratori hanno la seguente struttura gerarchica:



I contenitori (containers)

- Sono essenzialmente delle ADT che:
  - possono contenere dati di qualsiasi tipo (oggetti)
  - implementano metodi per accedere ai dati.
  - hanno un iteratore associato che permette di muoversi tra gli elementi del contenitore.

- I contenitori gestiscono in maniera automatica lo spazio necessario a memorizzare gli oggetti da contenere

C++ STL Containers

Container class templates

Sequence containers:

array C++11	Array class (class template)
vector	Vector (class template)
deque	Double ended queue (class template)
forward_list C++11	Forward list (class template)
list	List (class template)

Container adaptors:

stack	LIFO stack (class template)
queue	FIFO queue (class template)
priority_queue	Priority queue (class template)

Associative containers:

set	Set (class template)
multiset	Multiple-key set (class template)
map	Map (class template)
multimap	Multiple-key map (class template)

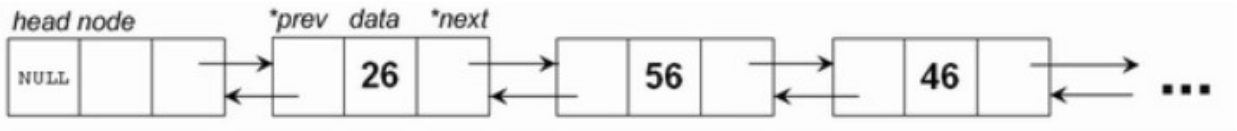
Unordered associative containers:

unordered_set C++11	Unordered Set (class template)
unordered_multiset C++11	Unordered Multiset (class template)
unordered_map C++11	Unordered Map (class template)
unordered_multimap C++11	Unordered Multimap (class template)

http://www.cplusplus.com/reference/stl/

Sequenze

- **Vector**
  - Elementi memorizzati in array (allocazione contigua)
  - Complessità di inserimento/cancellazione:
    - alla fine: **costante**
    - All'interno: **lineare,** con il numero di elementi presenti nel vettore
  - Iteratore: random
- **List**
  - Elementi memorizzati in liste a puntatori doppiamente linkate:



  - Complessità di inserimento/cancellazione **costante** (ovunque)
  - Iteratore: biderezionale

Gli algortimi

- Funzioni globali capaci di agire su contenitori differenti
- La maggior parte di queste funzioni prendono come argomenti intervalli di iteratori e/o oggetti
- **Esempi**
  - sort
  - find
  - fill
  - copy
  - min_element, max_element

La classe vector: dichiarazioni

```
vector<int> v; // crea un vettore v di interi,
               // inizialmente vuoto
vector<double> v(10); // vettore di double
                      // con dimensione iniziale 10
class T;
    .
    .
vector<T> v; // vettore di oggetti della classe T
```

La classe vector: funzioni

```
size_type size() const; // restituisce il #elementi nel vettore
bool empty() const; // restituisce true se il vettore è vuoto
void push_back(const T& x); // aggiunge x alla fine del vettore
void pop_back(); // rimuove l'ultimo elemento del vettore
void clear();  // rimuove tutti gli elementi dal vettore
T& at(size_type i); // restituisce un riferimento  all'elemento
                    // in posizione i
void swap (vector &v); // scambia il contenuto del vettore con quello
                       // del vettore v
void merge(vector &v)  // fonde il contenuto del vettore con quello di v.
                       // i vettori devono essere ORDINATI
void unique() // rimuove tutti gli elementi, tranne il primo,
              // uguali ADIACENTI.
              // È utile quando il vettore è ORDINATO
```

La funzione capacity

- restituisce il numero totale di elementi allocato per il vettore
- Non necessariamente uguale al numero di elementi effettivamente contenuti nel vettore (restituito dalla funzione `size()`)
- Non è la capacità massima del vettore:
  - Quando il vettore si riempie, esso viene automaticamente espanso

Gli iteratori

```
vector<T>::iterator it; // iteratore per l'accesso ad un
                        // vettore di oggetti di tipo T
it++ // avanza di un elemento
it-- // retrocede di un elemento
*it // restituisce l'oggetto puntato da it
iterator begin(); // restituisce un iteratore al primo elemento del
vettore
iterator end();  // restituisce un iteratore che  punta subito DOPO
                 // l'ultimo
                 // elemento del vettore
iterator insert(iterator it, const T& x);
// inserisce x nella posizione precedente a quella
// puntata dall'iteratore it e restituisce un
// iteratore all'elemento inserito
iterator erase(iterator it); // cancella l'elemento puntato
                             // dall'iteratore it e restituisce
                             // l'iteratore all'elemento successivo
                             //  a quello cancellato
```

Esempio

```
#include <vector>
#include <algorithm>
#include <iostream>

using namespace std;
const int SEED = 5;
int main() {
  vector<int> v; // Definizione di un vettori di interi
  int val, i;

  srand(SEED);
  for (i=0; i<10; i++) {
     val = rand() % RAND_MAX;
     v.push_back(val);
  }
  vector<int>::iterator iter;
  for ( iter=v.begin(); iter!=v.end(); iter++)
     cout << "vector : " << *iter << endl;

  iter = find(v.begin(), v.end(), 3); // ricerca del valore 3
  sort (v.begin(), v.end());  // ordinamento del vettore

  for (iter=v.begin(); iter!=v.end(); iter++)
     cout<< "vector : " << *iter << endl;

  iter = min_element( v.begin(), v.end()); // ricerca del minimo

  return 0;
}
```

Iteratori costanti

```
#include <vector>
#include <iostream>

using namespace std;

void show(const vector<int> &v)
{
  vector<int>::const_iterator c_it; // OK
  vector<int>::iterator it; // ERRORE!

  c_it  = v.begin();
  while (c_it != v.end()) {
    cout << *c_it++ << " ";
  }
  cout << endl;

  return;
}
```

Vettori di oggetti

```
class Esame {
public:
  string nome;
  int voto;
  int crediti;
  Esame(string n, int v, int c) {
    nome = n;
    voto = v;
    crediti = c;
  }
};
```

```
float media(vector<Esame> v)
{
  float m = 0;
  int c, i, N;

  N = v.size();

  for (i=0, m=0.0, c=0; i < N; ++i) {
    m += v[i].voto * v[i].crediti;
    c += v[i].crediti;
  }

  return m/c;
}
```

```
float media(const vector<Esame> &v)
{
  float m = 0;
  int c;
  vector<Esame>::const_iterator it;

  it = v.begin();
  c = 0;
  m = 0.0;
  while (it != v.end()) {
    m += (*it).voto *  (*it).crediti;
    c += (*it++).crediti;
  }

  return m/c;
}
```

Accesso con iteratore

Accesso “classico”

```cpp
#include <iostream>
#include <vector>
#include <string>

using namespace std;

int main()
{
  vector<Esame> v;
  Esame e1("Programmazione a Oggetti",30,6);
  Esame e2("Calcolo 2",25,4);
  Esame e3("Fisica 1",28,6);
  float m;

  v.push_back(e1);
  v.push_back(e2);
  v.push_back(e3);

  m = media(v);
  cout << "media = " << m << endl;

  return 0;
}
```

La classe List: dichiarazioni

```
#include <list> // inclusione libreria list di stl

list<int> l; // crea una lista vuota di interi
list<Esame> l; // crea una lista vuota di oggetti
               //   della classe esame
list<T> l; // lista di oggetti del tipo T
```

Classe list: gli iteratori

```
vector<T>::iterator it; // iteratore per l'accesso ad un
                        // vettore di oggetti di tipo T
it++ // avanza di un elemento
it-- // retrocede di un elemento
*it // restituisce l'oggetto puntato da it
iterator begin(); // restituisce un iteratore al primo elemento del
vettore
iterator end();  // restituisce un iteratore che  punta subito DOPO
l'ultimo
                 // elemento del vettore
iterator insert(iterator it, const T& x);
// inserisce x nella posizione precedente a quella
// puntata dall'iteratore it e restituisce un
// iteratore all'elemento inserito
iterator erase(iterator it); // cancella l'elemento puntato
                             // dall'iteratore it e restituisce
                             // l'iteratore all'elemento successivo
                             //  a quello cancellato
```

La classe List: funzioni principali

```
size_type size() const; // restituisce #elementi nella lista
bool empty() const;     // restituisce true se la lista è vuota
void push_back(const T& x);  // appende l'oggetto x alla fine della lista
void push_front(const T& x); // inserisce l'oggetto x in testa alla lista
void pop_back(); // rimuove l'ultimo elemento della lista
void pop_front(); // rimuove il primo elemento della lista
void clear(); // rimuove tutti gli elementi  dalla lista
void reverse(); // inverte la lista
void sort(); // ordina gli elementi della lista in ordine crescente
void remove(const T& x); // rimuove tutti gli elementi uguali a x
void swap (list &l); // scambia il contenuto della lista con quello della
                     // lista l
void merge(list &l)  // fonde il contenuto della lista con quello di l.
                     // le liste devono essere ordinate
void unique() // rimuove tutti gli elementi, tranne il primo,
              // uguali ADIACENTI.
              // È utile quando la lista è ORDINATA
```

Esempi

passaggio per riferimento costante

passaggio per riferimento

```
void eliminaPari(list<int> &l)
{
  list<int>::iterator it;

  it = l.begin();
  while (it != l.end())
    if (*it % 2 == 0)
      it = l.erase(it);
    else it++;

  return;
}
```

```
void show(const list<int> &l)
{
  list<int>::const_iterator it;

  for(it = l.begin(); it != l.end(); it++)
     cout << *it << " ";

cout << endl;

  return;
}
```

```
#include <iostream>
#include <list>

using namespace std;

int main()
{
  list<int> l;
  int i;

  for (i = 1; i <= 10; i++)
     l.push_back(i);

  eliminaPari(l);
  show(l) ;

  return 0;
}
```

OUTPUT

```
1 3 5 7 9
```

La classe map

- Implementa un contenitore “associativo”:
  - Realizza un'associazione tra chiavi “univoche” e valori
- Una chiave è un identificatore (unico) assegnato ad un valore
- Successivamente, si può accedere (efficientemente) al valore per mezzo della relativa chiave

Dichiarazioni

```
map<string,int> m; // crea una mappa indicizzata
                   // da chiavi di tipo string,
                   // contenente valori di tipo int

map<int, Studente> m; // crea una mappa indicizzata
                      // da chiavi di tipo intero,
                      // contenenti  oggetti
                      // della classe Studente
map<int,T> m;         // mappa con chiavi di tipo int
                      // e valori di tipo T
```

Il tipo utilizzato per le operazioni di inserimento è definito tramite un **typedef**.

ESEMPIO

```
typedef map<string,int>::value_type valType;
```

Funzioni

```
size_type size() const; // restituisce il numero di elementi nella mappa

bool empty() const; // restituisce true se la mappa m è vuota

pair<iterator,bool> insert(valType("PO", 27)); // inserisce nella mappa
                           // il valore 27 con chiave "PO";
                           // l'inserimento avviene soltanto
                           // se nella mappa non esiste già
                           // un'istanza con la stessa chiave

iterator find("PO") bool; // restituisce un iteratore all'elemento
                          // la cui chiave è "PO"
                          // se l'istanza non è presente restituisce end()
size_type count("PO") bool; // restituisce #elementi nella mappa
                            // la cui chiave è "PO" (0 o 1)
void swap (map &m);  // scambia il contenuto della mappa
                     // con quello della mappa m
```

Iteratori

```
map<string,int>::iterator it; // iteratore ad una
                              // mappa <string,int>
it++ // avanza di un elemento
it-- // retrocede di un elemento
(*it).first // restituisce la chiave dell'oggetto puntato da it
(*it).second // restituisce il valore dell'oggetto puntato da it


iterator begin(); // restituisce un iteratore al primo
                  // elemento della mappa
iterator end();   // restituisce un iteratore che punta subito
                  // dopo l'ultimo elemento della mappa
iterator erase(iterator it); // cancella l'elemento puntato
                             // dall'iteratore; restituisce un
                             // iteratore all'elemento
                             // successivo a quello cancellato
```

Iteratori

```
#include <map>
#include <iostream>

using namespace std;

int main() {
  map<string,string> m;
  typedef map<string,string>::value_type val;

  m.insert(val("Matteo", "0984-1234"));
  m.insert(val("Francesca", "0963-2468"));
  m.insert(val("Giovanni", "+393474567"));

  map<string,string>::iterator it;
  for(it = m.begin(); it != m.end(); it++)
     cout << it->first << ": " << it->second << endl;

  return 0;
}
```

inserimento

OUTPUT

```
Francesca: 0963-2468
Giovanni: +393474567
Matteo: 0984-1234
```

```
#include <map>
#include <iostream>

using namespace std;

int main() {
  map<string,string> m;

  m["Matteo"] = "0984-1234";
  m["Francesca"] = "0963-2468";
  m["Giovanni"] = "+393474567";


  map<string,string>::iterator it;
  for(it = m.begin(); it != m.end(); it++)
     cout << it->first << ": " << it->second << endl;

  return 0;
}
```

OUTPUT

```
Francesca: 0963-2468
Giovanni: +393474567
Matteo: 0984-1234
```

Container: Operatori

- I contenitori possono essere confrontati utilizzando gli operatori:

  ==, !=, <=, >=, <, >,

  questi operatori usano l'ordinamento lessicografico:

  http://it.wikipedia.org/wiki/Ordine_lessicografico

- I container possono inoltre essere assegnati utilizzando l'operatore =.

Algoritmi

```
iterator find(iterator first, iterator last, const T& value);
/** Cerca value nell'intervallo [first,last) (last escluso)
  * restituisce l'iteratore al primo elemento trovato,
  * last altrimenti.
  */

void replace(iterator first, iterator last, const T& oldVal,
const T& newVal);
/** Sostituisce tutte le istanze di oldVal con newVal
  * nell'intervallo [first,last) (last escluso)
  */

void sort(iterator first, iterator last);
/** Ordina gli elementi nell'intervallo [first,last) in ordine
  * crescente
  * NOTA
  * la classe list ha un suo metodo sort(), ottimizzato per le
  * liste
  */
```

```
iterator set_difference(iterator first1, iterator last1,
iterator first2, iterator last2, iterator result);
/** Costruisce la sequenza ordinata degli elementinella prima
  * sequenza ([first1,last1)) ma non contenuti nella seconda
  * ([first2,last2)). L'iteratore result specifica la posizione
  * del contenitore in cui deve essere inserito il risultato
  * L'iteratore restituito punta a una posizione oltre l'ultimo
  * elemento posto nel contenitore individuato da result.
  * E' necessario che le due sequenze in input siano ordinate
  */
```

```
iterator set_intersection (iterator first1, iterator last1,
iterator first2, iterator last2, iterator result);
/** Come il precedente, ma calcola l'intersezione delle
  * due sequenze.
  */
```

```
iterator set_union (iterator first1, iterator last1, iterator
first2, iterator last2, iterator result);
/** Come il precedente, ma calcola l'unione delle
  * due sequenze.
  */
```

Find: esempio

```
#include <list>
#include <algorithm>
#include <iostream>

using namespace std;

int main() {
  list<int> lista;
  int i, x;

  for (i = 1; i <= 10; i++)
     lista.push_back(i);

  cout <<"inserisci numero da cercare: ";
  cin >> x;

  list<int>::iterator it;
  it = find(lista.begin(), lista.end(), x);

  if (it != lista.end())
    cout << "Numero presente\n";
  else cout << "Numero non presente\n";

  return 0;
}
```