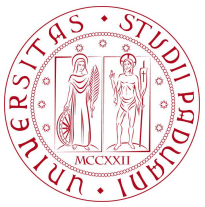
La classe `std::vector` della Standard Template Library del C++

Alberto Garfagnini

Università degli studi di Padova

23 Ottobre 2013



Programmazione generica in C++ : i Template

- I templates sono probabilmente uno degli aspetti più importanti della programmazione in linguaggio C++.
- Permettono di definire delle azioni che si vuole vengano attuate da oggetti di tipo diverso.
- I template si dichiarano usando la parola chiave `template` seguita da una lista di parametri tipo

```
template <parameter list>
...template declaration...
```

Esistono diversi tipi di dichiarazioni template

- di una funzione template;
- di una classe template;
- di membri (metodi) di una classe;
- ...

Esempio di funzione template

- Si voglia scrivere una funzione che ritorna il massimo tra due oggetti generici

```
// funzione che ritorna il piu' grande tra due oggetti
template <typename objT>
objT & get_max( objT & value1, objT & value2 )
{
    if (value1 > value )
        return value1;
    else
        return value2;
}
```

- Consideriamo due esempi di invocazione della funzione con tipi `int` e `double`

```
int n1 = 5, n2 = 7;
int n_max = get_max <int> (n1, n2);
double val1 = 5.3, val2 = -3.4;
double val_max = get_max <double> (val1, val2);
```

Programmazione generica in C++ : i Template

- La Standard Template Library contiene svariati ottimi esempi di template.
- Nella STL sono disponibili una vasta collezione di classi template e funzioni che contengono algoritmi e utility generiche.
- Un esempio tipico sono i contenitori template che implementano array dinamici, liste, code, stack, etc.
- sono disponibili:
  - contenitori per lo storage dell'informazione;
  - iteratori che permettoni di accedere all'informazione;
  - algoritmi per manipolare il contenuto dei contenitori

Contenitori sequenziali: `std::vector`

- la classe `std::vector` permette di creare un array dinamico e di accedere o manipolare un suo elemento data la posizione (indice) usando l'operatore `[]`

1. è un array dinamico
2. è possibile aggiungere o togliere elementi, modificando run-time la dimensione
3. normalmente si aggiungono elementi nuovi alla fine, ma è anche possibile inserirli all'inizio o in una posizione qualsiasi
4. è possibile effettuare delle ricerche all'interno del vector

Gli iteratori nella STL

- l'esempio più semplice di iteratore è un puntatore
- Gli iteratori sono classi template che possono essere pensati come una generalizzazione dei puntatori
- tramite gli iteratori è possibile *navigare* tra gli elementi di un contenitore in modo da accedere agli elementi ed eseguire operazioni
- gli iteratori STL posssono essere classificati in
  - Iteratore di Input : possono essere usati per accedere ad un oggetto che si trova in una collezione.
  - Iteratori di Output : permettono al programatore di scrivere nella collezione.
- inoltre, a seconda di come permettono di muoversi nella collezione sono
  - Iteratori in Avanti
  - Iteratori Bidirezionali
  - Iteratori ad Accesso Casuale

std::vector

La classe `std::vector` fornisce le funzionalità di un array dinamico con le seguenti caratteristiche:

- inserzione di nuovi elementi alla fine dell'array
- rimozione di elementi dalla fine dell'array
- inserzione/rimozione di elementi in testa o in un punto qualsiasi dell'array
- per usare il contenitore bisogna includere l'header file

```
#include <vector>
```

Es 1 - creazione di vettori

```
#include <vector>
int main()
{
  // Un vettore dinamico
  std::vector <double> v_dyna;

  // Un vettore con 10 elementi
  std::vector <int> vint_10_elements(10);

  // Un vettore con 10 elementi, inizializzati a 90
  std::vector <int> vint_10(10, 90);

  // Un vettore inizializzato con il contenuto di un altro
  std::vector<int> vcopy(vint_10_elements);

  // Un vettore con 5 valori presi da un altro
  std::vector<int> vcopy_5(vint_10.begin(),vint_10.begin()+5);
}
```

Es 2 - Aggiungiamo elementi ad un vettore

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

int main()
{
  // Un vettore dinamico
  std::vector <int> v_dyna;

  // Inseriamo dei valori
  v_dyna.push_back(1);
  v_dyna.push_back(22);

  v_dyna.push_back(22+34);
  v_dyna.push_back( rand() );

  std::cout << "Il vettore contiene ";
  std::cout << v_dyna.size() << " elementi\n";

}
```

Es 3 - Utilizzo semantica degli array

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

int main()
{
  // Fissiamo la dimensione dell'array
  std::vector <double> v(4);

  // Inseriamo dei valori
  v[0] = 1.0;
  v[1] = 22.0;

  v[2] = sqrt(3.1415/2);
  v[3] = 37.5;

  std::cout << "Il vettore contiene ";
  std::cout << v.size() << " elementi\n";

}
```

Es 4 - Uso degli iteratori

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

using namespace std;

int main()
{
  vector <double> v(4);
  ...

  vector <double>::iterator walk = v.begin();
  while (walk != v.end())
  {
    cout << * walk << endl;
    // L'iteratore va incrementato
    // per accedere all'elemento successivo
    walk++;
  }

}
```

Es 4bis - Uso della sintassi degli array

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

using namespace std;

int main()
{
  vector <double> v(4);
  ...
  int index = 0;
  while (index < v.size())
  {
    cout << v[index] << endl;
    // Il contatore va incrementato
    // per accedere all'elemento successivo
    index++;
  }

}
```

Es 5 - Ricerca del numero 2991 nel vettore

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

int main()
{
  vector <int> vi;
  ...
  vector <int>::iterator element;
  element = find(vi.begin(), vi.end(), 2991)

  // Controlliamo se l'elemento e' stato trovato
  if (element != vi.end())
  {
      int index = distance(vi.begin(), element);
      cout << "Valore " << *element;
      cout << " trovato alla posizione " << index << endl;
  }

}
```

Es 6 - Rimozione di un elemento alla fine del vettore

```
#include <iostream>
#include <vector>
int main()
{
  // Un vettore dinamico
  std::vector <int> v_dyna;

  // Inseriamo dei valori
  v_dyna.push_back(23);
  v_dyna.push_back(76);

  std::cout << "Il vettore contiene ";
  std::cout << v_dyna.size() << " elementi\n";

  // Estraiamo un valore
  v_dyna.pop_back();

  std::cout << "Il vettore contiene ";
  std::cout << v_dyna.size() << " elementi\n";
}
```

Es 7 - Vettore a 2-dim

- una matrice n $\times$ m può essere pensata come un vettore di vettori

```
// matrice bidimensionale 3 x 2
#include <vector>
using namespace std;

int main()
{
  vector <vector <double> > A(3, vector <double> (2,0);

  A[0][0] = 0;   A[0][1] = 1;
  A[1][0] = 10;  A[1][1] = 11;
  A[2][0] = 20;  A[2][1] = 21;

  // Loop con indici
  for (int i=0; i<3; i++)
    for (int j=0; j<2; i++)
        std::cout << A[i][j] << endl;
}
```

Es 7 - Vettore a 2-dim con iteratori

```
// matrice bidimensionale 3 x 2
#include <vector>
using namespace std;
int main()
{
  vector <vector <double> > Matrix;
  vector <double> > A, B;
  vector <vector <double> >::iterator iti;
  vector <double>::iterator itj;

  A.push_back(100);
  ...
  B.push_back(20);
  ...
  Matrix.push_back(A);
  Matrix.push_back(B);

  // Loop con iteratori
  for (iti=Matrix.begin(); iti!=Matrix.end(); iti++)
    for (itj=(*iti).begin(); itj!=(*iti.end()); itj++)
        std::cout << *itj << endl;
}
```

std::vector - costruttori

`vector <type> v`

- crea un vettore di tipo `type`

`vector <type> v(n)`

- crea un vettore di tipo `type` e dimensione `n`

`vector <type> v(n, const T & t)`

- crea un vettore di tipo `type` con dimensione `n` e lo inizializza ai valori costanti `t`

`vector <type> v(begin_iterator,end_iterator)`

- crea un vettore di tipo `type` e lo inizializza copiando gli elementi da un altro vettore da `begin_iterator` a `end_iterator`

std::vector - metodi

`v.begin()`

- (iteratore) : ritorna un iteratore che punta al primo elemento del vettore

`v.end()`

- (iteratore) : ritorna un iteratore che punta all'elemento successivo all'ultimo

`v.size()`

- (int) : ritorna il numero di elementi del vettore (è uguale a `v.end() - v.begin()`)

`v.capacity()`

- (int) : ritorna il numero di elementi inseribili senza necessità di riallocare la memoria

`a == b`

- (bool) : ritorna `true` se `a` e `b` hanno la stessa dimensione e ogni elemento di `a` è equivalente (`==`) al corrispondente elemento di `b`

std::vector - metodi

`v.push_back(t)`

- (void) : inserisce un elemento `t` alla fine del vettore

`v.pop_back()`

- (void) : elimina l'ultimo elemento inserito nel vettore

`v.front()`

- (T &) : ritorna il primo elemento

`v.back()`

- (T &) : ritorna l'ultimo elemento

`v.[i]`

- (T &) : accede all'elemento `i`

`v.at(i)`

- (T &) : accede all'elemento `i` (e contolla che i non superi i limiti degli indici dell'array - bound checking)

Array dinamico

```
// definiamo il puntatore all'array
double ** matrix = NULL;

// Allochiamo memoria per le righe
matrix = new double *[ROWS];

// e per le colonne
for (int i=0; i< ROWS; i++)
    matrix[i] = new double[COLS];

// Una volta terminato, liberiamo la memoria
for (int i=0; i< ROWS; i++) {
    delete [] matrix[i];
}

delete [] matrix;
```

Array dinamico generico con i template

```
template <type T>
T ** creaArray(int righe, int colonne){
  T ** matrix;
  matrix = new T *[righe];
  for (int i=0; i< righe; i++)
      matrix[i] = new T[colonne];
  return matrix;
}

template <type T>
void freeArray(T ** matrix){
  delete [] * matrix;
  delete [] matrix;
}

int main() {
 double ** A = creaArray<double>(4,4);
 A[0][0] = 3.1415;
 A[1][1] = 3.1415/2.0;
 cout << A[0][0] << A[1][1] << endl;
 freeArray<double>(A);
}
```