Esercitazione 8

19/11/2010

Es.

Per ogni punto creare un programma differente.

1. Creare un vettore di complex e ordinarlo, utilizzando l'algoritmo della STL sort, in maniera decrescente rispetto al valore assoluto.

2. Creare un vettore di interi avente diverse occorrenze del numero 55. Utilizzare l'algoritmo della STL find per trovare il terzo elemento di valore 55.

3. Creare una lista di **float** e contare il numero di elementi il cui valore assoluto è minore di 4.5. Utilizzare l'algoritmo della STL count_if.

4. Creare un insieme di interi e trovare la somma di tutti gli elementi, ciascuno moltiplicato per 3. Utilizzare l'algoritmo della STL accumulate sia per la somma che per la moltiplicazione.

5. Creare due insiemi di interi e trovare la loro intersezione e la loro unione, utilizzando gli algoritmi della STL set_union e set_intersection .

6. Creare una mappa che prende come chiave un intero e come valore un puntatore a funzione, che rappresenta una funzione da $\mathbb{R}^3 \to \mathbb{R}$. Inserire nella mappa le seguenti due funzioni, di chiave rispettivamente 1 e 5

$$f_1(x, y, z) = x + 2y - z^2\,,$$
$$f_5(x, y, z) = 2y - 6\,.$$

   Estrarre quindi dalla mappa la funzione associata alla chiave 5 e valutarla nel punto $(1, 2, 3)$.

Soluzione

Analizziamo la soluzione punto per punto.

1. Presentiamo qui di seguito il listato contenente la soluzione del primo esercizio

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

typedef complex<int> Complex;

struct sortPredicate
{
  inline bool operator()( const Complex& first, const Complex& second ) const
  {
    return (norm(first) > norm(second));
  }
};

void printVect( const vector<Complex>& );

int main(int argc, char* argv[])
{
  // The number of complex
  unsigned int N = 5;

  // Create the vector with zero length
  vector<Complex> vectCompl(0);

  // Reserve N complex
  vectCompl.reserve( N );

  // Fill the vector
  for ( unsigned int i = 0; i < N; ++i )
    {
      double real = ( rand() % 100 );
      double imag = ( rand() % 100 );
      vectCompl.push_back( Complex( real, imag ) );
    }

  // Print the vector
  printVect( vectCompl );

  // Sort the vector
  sort( vectCompl.begin(), vectCompl.end(), sortPredicate() );

  cout << "Ordered sequence" << endl;

  // Print the sorted vector
  printVect( vectCompl );


  return 0;
}

void printVect( const vector<Complex>& vect )
{
  for ( vector<Complex>::const_iterator it = vect.begin(); it != vect.end(); it++)
    cout << it->real() << " + i * " << it->imag() << endl;
}
```

Per l'utilizzo dell'algoritmo sort della standard template library, è necessario creare un predicato che permette di valutare l'espressione >. Un predicato è una funzione, o un funtore, che riceve in ingresso dei parametri e fornisce in uscita un **bool**. Nella soluzione presentata si è scelto di utilizzare un funtore. Nel caso dell'algoritmo sort il predicato prende in ingresso due valori utili al loro confronto, tale algoritmo funziona su tutti i contenitori della standard template library.

2. Presentiamo il listato contenente la soluzione del secondo esercizio

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

typedef vector<int> Vector;

void printVect ( const Vector& );

int main( int argc, char* argv[] )
{
  Vector values(10,10);

  // Insert four 55
  values[2] = 55;
  values[4] = 55;
  values[6] = 55;
  //  values[9] = 55;

  // Print the vector
  printVect( values );

  // Search the first 55
  Vector::iterator it1 = find( values.begin(), values.end(), 55 );

  // Determine if it is found, then search the second one
  if ( it1 != values.end() )
    {
      Vector::iterator it2 = find( ++it1, values.end(), 55 );

      // Determine if the second 55 is found, then search the third
      if ( it2 != values.end() )
        {
          it1 = find( ++it2, values.end(), 55 );

          // Determine if the third 55 is found
          if( it1 != values.end() )
            {
              cout << "Three values of 55 find!!" << endl;
              cout << "Position of the third " << static_cast<int>(it1 - values.begin())
                   << endl;
            }
          else
            {
              cerr << "Third value of 55 not found" << endl;
              abort();
            }
        }
      else
        {
          cerr << "Second value of 55 not found" << endl;
          abort();
        }
      // Search the second 55
    }
  else
    {
      cerr << "First value of 55 not found" << endl;
      abort();
    }

  return 0;
}

void printVect( const Vector& vect )
{
  for ( unsigned int i = 0; i < vect.size(); ++i )
    cout << vect[i] << endl;
}
```

L'algoritmo find della standard template library permette di trovare elementi all'interno di un

contenitore, è richiesta quindi l'implementazione dell'operatore == per il tipo memorizzato. In uscita, se l'elemento è stato correttamente trovato, l'algoritmo ritorna un iteratore al valore altrimenti ritorna la posizione end() del contenitore. È quindi facile, e caldamente consigliato, controllare sempre che la ricerca abbia dato esito positivo. Per ricercare l'elemento successivo al primo è possibile utilizzare find nella sotto-sequenza formata dall'elemento successivo a quello appena trovato fino alla fine della sequenza originale. Per scorrere di una posizione l'iteratore è possibile utilizzare il pre-incremento ++. La ricerca della terza occorrenza di 55 è analoga alla seconda. Per sapere quale sia l'indice che corrisponde al un iteratore, in un vettore, è possibile utilizzare l'operatore − tra l'iteratore stesso e la posizione begin() del vettore. Si noti l'utilizzo all'interno del codice.

3. Presentiamo il listato contenente la soluzione del terzo esercizio

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

using namespace std;
typedef list<float> List;

void printList( const List& );

struct listPredicate
{
  inline bool operator()( const float& value ) const
  {
    return ( fabs(value) < 4.5 );
  }
};

int main( int argc, char* argv[])
{
  // Decleare the list of 10 elements
  List countList(10);

  // Fill the list
  float val = 1;
  for ( List::iterator it = countList.begin(); it != countList.end(); ++it)
    {
      *it = val;
      val *= -1.3;
    }

  // Print the list
  printList( countList );

  // Count the number of element in the list with absolute value less then 4.5
  unsigned int N = (unsigned int) count_if( countList.begin(),
                                            countList.end(),
                                            listPredicate() );

  cout << "Number of element with absolute value less then 4.5 is " << N << endl;

  return 0;
}

void printList( const List& printList )
{
  for ( List::const_iterator it = printList.begin(); it != printList.end(); ++it)
    cout << *it << endl;
}
```

Per utilizzare l'algoritmo count_if della standard template library è necessario creare un predicato simile a quello creato nel punto 1, con la differenza che ora vi è solamente un parametro di ingresso. In ingresso la funzione count_if prende la lista su cui operare, definita attraverso due iteratori e il predicato per la valutazione dell'**if**. In uscita cout_if restituisce il numero di

occorrenze trovate che soddisfano il predicato. Notiamo infine l'utilizzo degli const_iterator utile alla stampa della lista, è suggerito, e caldamente consigliato, utilizzare i const_iterator quando il valore associato all'iteratore non viene modificato.

4. Presentiamo il listato contenente la soluzione del quarto esercizio

```
#include <iostream>
#include <set>
#include <numeric>
using namespace std;
typedef set<int> Set;

void printSet( const Set& );

struct setFunctor
{
  inline int operator() ( const int& first, const int& second ) const
  {
    return ( first + 3* second );
  }
};

int main(int argc, char* argv[])
{
  // Declare the set
  Set sumSet;

  // Fill the set
  int N = 10;
  for ( int i = 0; i < N ; ++i )
      sumSet.insert( i );

  // Print the set
  printSet( sumSet );

  // Define the starting value
  int startingValue = 0;

  // Accumulate all the values
  int sum = accumulate( sumSet.begin(), sumSet.end(), startingValue, setFunctor() );

  // Print the sum
  cout << "Sum: " << sum << endl;

  return 0;
}

void printSet( const Set& setPrint)
{
  for( Set::const_iterator it = setPrint.begin(); it != setPrint.end(); it++ )
    cout << *it << endl;
}
```

Per risolvere l'esercizio è necessario creare un funtore, o una funzione, che permette di accumulare i valori, ciascuno moltiplicato per 3. L'utilizzo di default della funzione accumulate permette di sommare tutti i valori all'interno del contenitore, tuttavia dovendo moltiplicare ciascun elemento per 3 è necessaria la creazione di un opportuno funtore. Tale funtore prende in ingresso due valori, in cui il primo è la somma parziale mentre il secondo è il nuovo valore da aggiungere. accumulate permette di definire un valore di partenza e, grazie ad esso, capisce il tipo per il valore di ritorno.

5. Presentiamo il listato contenente la soluzione del quinto esercizio

```
#include <iostream>
#include <set>
#include <algorithm>

using namespace std;
typedef set<int> Set;
```

```
void fillSet( Set& , const int&, const unsigned int& );

void printSet( const Set& );

int main( int argc, char* argv[] )
{
  // Declare two sets
  Set first;
  Set second;

  Set unionSet;
  Set intersectionSet;

  // Fill the first set
  fillSet( first, 1, 10 );

  // Fill the second set
  fillSet( second, 8, 7 );

  // Print the first set
  printSet( first );

  // Print the second set
  printSet( second );

  // Union of the sets
  set_union( first.begin(), first.end(),
             second.begin(), second.end(),
             inserter(unionSet, unionSet.begin() ) );

  printSet( unionSet );

  // Intersection of the sets
  set_intersection( first.begin(), first.end(),
                    second.begin(), second.end(),
                    inserter(intersectionSet, intersectionSet.begin() ) );

  printSet( intersectionSet );

  return 0;
}

void fillSet( Set& fill, const int& start, const unsigned int& N )
{
  for( unsigned int i = 0; i<N; ++i)
      fill.insert( start + i);
}

void printSet( const Set& setPrint )
{
  for( Set::const_iterator it = setPrint.begin(); it != setPrint.end(); it++ )
    cout << *it << endl;
  cout << endl;
}
```

Per risolvere l'esercizio sono stati creati quattro insiemi vuoti, due che conterranno i valori e altri due che conterranno l'intersezione e l'unione dei primi. L'utilizzo di set_union e set_intersection richiede due contenitori in ingresso e un iteratore per l'uscita. Tuttavia se il contenitore associato all'iteratore di uscita è vuoto allora è necessario utilizzare gli insert_iterator , come presentato nella soluzione.

6. Presentiamo il listato contenente la soluzione del sesto esercizio

```
#include <iostream>
#include <map>
#include <cstdlib>
using namespace std;

typedef double (*FunPoint)( const double&,
                            const double&,
```

```
                              const double& );

// Boundary function
double dirichlet_1 ( const double& x, const double& y, const double& z )
{
  return x+2.*y-z*z;
}

double dirichlet_2 ( const double& /*x*/, const double& y, const double& /*z*/ )
{
  return 2.*y-6.;
}

int main()
{
  typedef map<unsigned int, FunPoint> bcMap;

  // Map to store all the boundary conditions
  bcMap boundaryMap;

  // Single boundary condition pair, usefull for insertion
  bcMap::value_type pairBC;

  // Insert the first pair
  boundaryMap[1] = *dirichlet_1;

  // Insert the second pair
  boundaryMap[5] = *dirichlet_2;

  // Extract the iterator of the function with label 5
  bcMap::iterator iteratorBC = boundaryMap.find( 5 );

  // Check if the function is present
  if ( iteratorBC == boundaryMap.end() )
    {
      cerr << " Boundary not found " << endl;
      abort();
    }

  // Evaluate the function
  cout << "Value " << iteratorBC->second(1, 2, 3) << endl;

  return 0;
}
```

L'esercizio presenta un modo elementare ma molto efficace per trattare le condizioni al bordo. Infatti è possibile creare una map la cui chiave è il flag associata ad un bordo, mentre il valore è un puntatore a funzione. In questo modo, tramite l'operatore di accesso [], possiamo accedere alla funzione associata ad una certa chiave. Per riempire la mappa è possibile utilizzare l'operatore di accesso [], mentre per ricercare un elemento con una certa chiave è possibile utilizzare il metodo find delle map; viene restituito un iteratore al valore trovato altrimenti alla posizione end() della mappa. Per accedere al valore associato a tale iteratore è possibile utilizzare −>second.