Laborator STL

Teme:

1. Verificati functionalitatea containerelor de tip *string* si *list* pornind de la exemplul 1 si tutorialele suplimentare
2. Generati un *vector* de numere aleatoare in plaja 0-100. Afisati toate numerele divizibile cu 3 din container si pozitia lor in container. (Vezi ex. 2)
3. Implementati diversi algoritmi STL. Analizati eficienta lor (Vezi ex. 3, 4, 7)
4. Considerand obiecte functor, algoritmi de tip generator, generati numerele lui Fibonacii intr-un mod cat mai eficient si afisati numarul de aur si sectiunea de aur, pornind de la raportul a doua numere succesive. (vezi ex. 5)
5. Să se creeze variante pentru exemplul 3 ce utilizează alte tipuri de containere.
6. Să se scrie un program ce generează suma salariilor angajaţilor folosind algoritmul *accumulate( )*.
7. Pornind de la exemplele legate de algoritmi STL, verificati pe grupe functionalitatea lor folosind tutoriale suplimentare.
8. Considerand solutiile prezentate la curs legate de clasele *Student, Factura* si *Person*, analizati facilitatile de sortare posibile a fi folosite cu containere STL (*qsort()*, alg. de sortare, *sort()*). Alegeti o metoda de sortare dorita si folosind o clasa la alegere care sa aiba atribute de tip *string* si numerice defínti o colectie de obiecte care sa o sortati dupa mai multe criterii.

Exemplul 1

```
// utlizarea containerelor de tip string si list

#include <list>
#include <iostream>
#include <string>

using namespace std;

class Person
{
private:
       string name;
       int    age;
       string cnp;
public:
       Person(string inName, int inAge, string inCNP)
       {
              name = inName;
              age = inAge;
              cnp = inCNP;
       }

       string& getName(void)
       { return name; }
       int getAge(void)
       { return age; }
       string& getCNP(void)
       { return cnp; }
       bool operator == (const Person& p)
       { return (name == p.name);  }
       bool operator < (const Person& p)
       { return (age < p.age); }
};//class
```

```
void populatePeople(list<Person>& peopleList);
void printAndDeleteList(list<Person>& peopleList);

int main(){
list<Person> peopleList;
       populatePeople(peopleList);
       peopleList.sort();
       printAndDeleteList(peopleList);
       }//main

void populatePeople(list<Person>& peopleList)
{
char continueFlag = 'd';
string name;
int age;
string cnp;

       while (continueFlag == 'd')        {
              cout << "Nume: ";
              cin >> name;
              cout << "Varsta: ";
              cin >> age;
              cout << "CNP: ";
              cin >> cnp;

              Person p(name, age, cnp);

              peopleList.push_back(p);

              cout << "Tastati d pentru a continua sau alta tasta pentru a iesi:";
              cin.ignore();
              continueFlag = cin.get();
       }
}

void printAndDeleteList(list<Person>& peopleList)
{
       while (peopleList.size() > 0) {
              Person p = peopleList.front();
              peopleList.pop_front();
              cout << p.getName() << ":" << p.getAge() << ":" << p.getCNP() <<
endl;
       }
}
```

Exemplul 2

```
// utilizarea unui algoritm de cautare folosind un container de tip  vector
//afisarea tuturor valorilor divizibile cu 3
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

bool div_3 (int n);

 int main (){
```

```
typedef vector <int> IntVec;
IntVec v (10);

       for (int i = 0; i < v.size(); i++)
       {v[i] = (i + 1) * (i + 1);
       cout<<" "<<v[i]<<" ";
       }

       IntVec::iterator iter=v.begin();
       while (iter != v.end()){
       iter = find_if (iter, v.end(), div_3);
       if (iter != v.end()){
    cout<< "\nValoarea "<< *iter<< " din pozitia "<< (iter - v.begin() + 1)<< "
este divizibila cu 3"<< endl;
       iter++;}
       }
}//main

bool div_3 (int n) {
        return n % 3 ? 0 : 1;
}
```

<u>Exemplul 3</u>

```
// algoritmi diversi

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

int main(){
 vector<int> coll;
 vector<int>::iterator pos;

    // inserare elemente
    coll.push_back(2);
    coll.push_back(5);
    coll.push_back(4);
    coll.push_back(1);
    coll.push_back(6);
    coll.push_back(3);

    // cautare minimum
    pos = min_element (coll.begin(), coll.end());
    cout << "min: " << *pos << endl;

    // cautare maximum
    pos = max_element (coll.begin(), coll.end());
    cout << "max: " << *pos << endl;

    // sortare
    sort (coll.begin(), coll.end());
    // afisare elemente
    cout << endl << "Elementele sortate: " << endl;
    for (pos=coll.begin(); pos!=coll.end(); ++pos) {
        cout << *pos << ' ';
    }
    // cautare primul element cu valoarea 3
    pos = find (coll.begin(), coll.end(), 3);

    // schimba ordinea elementelor incepand cu ultimul gasit
    reverse (pos, coll.end());

    // afisare elemente
    cout << endl << "Elementele dupa schimbarea ordinii: " << endl;
    for (pos=coll.begin(); pos!=coll.end(); ++pos) {
        cout << *pos << ' ';
    }
    cout << endl;
}//main
```

Exemplul 4

```
// algoritm de tip accumulate
#include <vector>
#include <algorithm>
#include <numeric>
#include <iostream>
using namespace std;

int mult (int initial, int element) ;


 int main ()
{
vector <int> v(5);
       for (int i = 0; i < v.size(); i++)
              v[i] = i + 1;
        int prod = accumulate (v.begin(), v.end(), 1, mult);
       cout << "Factorial = " << prod << endl;
}//main

int mult (int initial, int element) {
       return initial * element;
}
```

Exemplul 5

```
// algoritm de tip generator
#include <vector>
#include <algorithm>
#include <iostream>
#include <iterator>

using namespace std;

class  Fibonacci   {
  private:
       int v1;
       int v2;
  public:
       Fibonacci () : v1(0), v2(1) { }
       int operator () ();
};

int  Fibonacci::operator()  ()
```

```
{
int r = v1 + v2;
       v1 = v2;
       v2 = r;
       return v1;
}

int main (){
vector <int> v1 (10);
Fibonacci generator;

       generate_n (v1.begin (), v1.size (), generator);
       copy (v1.begin (), v1.end (),  ostream_iterator<int>(cout, " "));
       cout << endl;
}//main
```

Exemplul 6.

```
/*  Maps
- in this case of associative container the, Key is the type of those elements
that also have the
function of keys, and Compare is the type of the comparison object.
- The definition is a mapping of int numbers onto string objects, with the numbers
internally sorted in descending order:
map<int, string> aMap
- the elements of a map are pairs (see def) */

// algoritm folosind map
#include<map>
#include<string>
#include<iostream>
using namespace std;

// two typedefs for abbreviations
// comparison object: less<long>()
typedef map<long, string> MapType;
typedef MapType::value_type ValuePair;
int main()
{
MapType Map;
Map.insert(ValuePair(836361136, "Andrew"));
Map.insert(ValuePair(274635328, "Berni"));
Map.insert(ValuePair(260736622, "John"));
Map.insert(ValuePair(720002287, "Karen"));
Map.insert(ValuePair(138373498, "Thomas"));
Map.insert(ValuePair(135353630, "William"));
// insertion of Xaviera is not executed, because the key already exists.
Map.insert(ValuePair(720002287, "Xaviera"));
/* Owing to the underlying implementation, the output of the names is sorted
by numbers: */
cout << "Output:\n";
MapType::const_iterator iter = Map.begin();
while(iter != Map.end())
{
//number, name
cout<<"Number: "<<(*iter).first<<": "<<(*iter).second<<"\n";
++iter;
}
cout<<"\n\nOutput of the name after entering the ID\n";
cout<<"Number: ";
long Number; cin >> Number;
iter = Map.find(Number);
if(iter != Map.end())
{
cout<<"\"(*iter).second\" "<<(*iter).second;
cout<<" has the same result as \"Map[Number]\": "<<Map[Number]<<"\n";
}
else cout<<"Data was not found. \n";
}//main
```

Exemplu 7

```
//
// STL.cpp
// Sample code to explain some advanced STL concepts. This is a shameless
// copy of online materials in order to create a simple example program to
// explain the basics of STL.
//
// Topics covered:
// ostream_iterator, copy, deque, insert_iterator, front_inserter,
// back_inserter, accumulate, count, count_if, find, find_if, generate,
// generate_n, fill, fill_n, transform, negate, mismatch, search, equal,
// for_each, swap, sort, merge, binary_search, includes, ptr_fun, set_union,
// set_intersection, set_difference.
//
// Prerequisite:
// Basic STL concepts: vectors, maps, sets, iterators, algorithms
//
// Author:
// Priyank Bolia (priyank.co.in)
//
// License:
// Priyank Bolia does not hold any copyrights for this work
// and the rights still remains with the original authors, who had done all
// the hard work.
//
// References:
// sgi.com/tech/stl/swap.html, msdn.microsoft.com
//

#define _CRT_SECURE_NO_WARNINGS  // Disable visual Studio warnings

#include <iterator>
#include <deque>
#include <iostream>
#include <algorithm>
#include <numeric>
#include <functional>
#include <vector>

using namespace std;

// Creation of a user-defined function object
// that inherits from the unary_function base class
class greaterthan : unary_function<int, bool>
```

```
{
  int num;

public:
  greaterthan(int n) : num(n) {}

  result_type operator()(argument_type i)
  {
    return (result_type)(i > num);
  }
};

// return the next Fibonacci number in the Fibonacci series.
int Fibonacci(void)
{
  static int r;
  static int f1 = 0;
  static int f2 = 1;
  r = f1 + f2 ;
  f1 = f2 ;
  f2 = r ;
  return f1 ;
}

template<class T> struct print : public unary_function<T, void>
{
  print(ostream& out) : os(out), count(0) {}
  void operator() (T x) { os << x << ' '; ++count; }
  ostream& os;
  int count;
};

inline bool lt_nocase(char c1, char c2) { return tolower(c1) < tolower(c2); }

int main ()
{
  int arr[4] = { 3,4,7,8 };  // Initialize a deque using an array.

  // A deque is very much like a vector: like vector, it is a sequence that
  // supports random access to elements, const time insertion and
  // removal of elements at the end of the sequence, and linear time
  // insertion and removal of elements in the middle. The main way in which
  // deque differs from vector is that deque also supports constant
  // time insertion and removal of elements at the beginning of the sequence.
  // Additionally, deque does not have any member functions analogous to
  // vector's capacity() and reserve(), and does not provide any of the
  // guarantees on iterator validity that are associated with those member
  // functions.
  deque<int> d(arr+0, arr+4);

  cout << "Start with a deque: ";  // Output the original deque.

  // An ostream_iterator is an Output Iterator that performs formatted
  // output of objects of type T to a particular ostream. Note that all
  // of the restrictions of an Output Iterator must be obeyed, including
  // the restrictions on the ordering of operator* and operator++
  // operations.
  copy(d.begin(), d.end(), ostream_iterator<int>(cout," "));
```

```
// OUTPUT: Start with a deque: 3 4 7 8

// Insert into the middle.
// Insert_iterator is an iterator adaptor that functions as an Output
// Iterator: assignment through an insert_iterator inserts an object
// into a Container. Specifically, if ii is an insert_iterator, then
// ii keeps track of a Container c and an insertion point p; the
// expression *ii = x performs the insertion c.insert(p, x).
insert_iterator<deque<int> > ins(d, d.begin()+2);
*ins = 5; *ins = 6;

// Output the new deque.
cout << endl << endl;
cout << "Use an insert_iterator: ";

// Copies elements from range [first, last) to the range
// [result, result + (last - first)).  That is, it performs the
// assignments *result = *first, *(result + 1) = *(first + 1), and so on.
// Generally, for every integer n from 0 to last - first, copy performs
// the assignment *(result + n) = *(first + n). Assignments are
// performed in forward order, i.e. in order of increasing n.
// The return value is result + (last - first)
copy(d.begin(), d.end(), ostream_iterator<int>(cout," "));
// OUTPUT: Use an insert_iterator: 3 4 5 6 7 8

// A deque of four 1s.
deque<int> d2(4, 1);

// Insert d2 at front of d.
// Front_insert_iterator is an iterator adaptor that functions as an
// Output Iterator: assignment through a front_insert_iterator inserts
// an object before the first element of a Front Insertion Sequence.
copy(d2.begin(), d2.end(), front_inserter(d));

// Output the new deque.
cout << endl << endl;
cout << "Use a front_inserter: ";
copy(d.begin(), d.end(), ostream_iterator<int>(cout," "));
// OUTPUT: Use a front_inserter: 1 1 1 1 3 4 5 6 7 8

// Insert d2 at back of d.
// Back_insert_iterator is an iterator adaptor that functions as an
// Output Iterator: assignment through a back_insert_iterator inserts
// an object after the last element of a Back Insertion Sequence.
copy(d2.begin(), d2.end(), back_inserter(d));

// Output the new deque.
cout << endl << endl;
cout << "Use a back_inserter: ";
copy(d.begin(), d.end(), ostream_iterator<int>(cout," "));
cout << endl << endl;
// OUTPUT: Use a back_inserter: 1 1 1 1 3 4 5 6 7 8 1 1 1 1

// Accumulate example
int A[] = {1, 2, 3, 4, 5};
const int N = sizeof(A) / sizeof(int);
deque<int> a(A+0, A+N);
```

```
// Adding the numbers
// Accumulate is a generalization of summation: it computes the sum
// (or some other binary operation) of init and all of the elements in
// the range [first, last).
cout << "The sum of ";
copy(a.begin(), a.end()-1, ostream_iterator<int>(cout, " + "));
cout << *(a.end()-1) << " is "
  << accumulate(a.begin(), a.end(), 0)
  << endl << endl;
// OUTPUT: The sum of 1 + 2 + 3 + 4 + 5 is 15

// Multiplying the numbers
// Multiplies<T> is a function object. Specifically, it is an Adaptable
// Binary Function. If f is an object of class multiplies<T> and x and y
// are objects of class T, then f(x,y) returns x*y. Similarly, there is:
// plus<T> (+), minus<T> (-), divides<T> (/), modulus<T> (%).
cout << "The product of ";
copy(a.begin(), a.end()-1, ostream_iterator<int>(cout, " + "));
cout << *(a.end()-1) << " is "
  << accumulate(a.begin(), a.end(), 1, multiplies<int>())
  << endl << endl;
// OUTPUT: The product of 1 * 2 * 3 * 4 * 5 is 120

// Count finds the number of elements in [first, last) that are
// equal to value.
cout << "Number of ones: "
  << (int)count(A, A + N, 1)
  << endl << endl;
// OUTPUT: Number of ones: 1

// Count_if finds the number of elements in [first, last) that satisfy
// the predicate pred.
cout << "Number of elements greater than 3: "
  << (int)count_if(A, A + N, greaterthan(3))
  << endl << endl;
// OUTPUT: Number of elements greater than 3: 2

// find returns the first iterator i in the range [first, last) such
// that *i == value.
// Returns last if no such iterator exists.
deque<int>::iterator find_result = find(a.begin(), a.end(), 2);
cout << "Find whether 2 is present in A: " << *(find_result)
  << endl << endl;
// OUTPUT: Find whether 2 is present in A: 2

// find_if returns the first iterator i in the range [first, last)
// such that pred(*i) is true.
// Returns last if no such iterator exists.
deque<int>::iterator find_if_result = find_if(a.begin(), a.end(),
  bind2nd(greater<int>(), 3));
// Binder2nd is a function object adaptor: it is used to transform an
// adaptable binary function into an adaptable unary function.
// Specifically, if f is an object of class
// binder2nd<AdaptableBinaryFunction>, then f(x) returns F(x, c),
// where F is an object of class AdaptableBinaryFunction and where
// c is a const. Both F and c are passed as arguments to
// binder2nd's constructor.
```

```
cout << "Find number greater than 3 in A: " << *(find_if_result)
  << " " << *(find_if_result++) << endl << endl;
// OUTPUT: Find number greater than 3 in A: 5 4

cout << "Fibonacci series: ";
// The generate_n algorithm traverses the range [First, First + n),
// assigning to each element the value returned by Gen.
// The return value is first + n.
generate_n(ostream_iterator<int>(cout, " "), 10, Fibonacci);
cout << endl << endl;
// OUTPUT: Fibonacci series: 1 1 2 3 5 8 13 21 34 55

const int NUM = 10;
vector<int> V1(NUM);
vector<int> V2(NUM);

// Generate assigns the result of invoking gen, a function object that
// takes no arguments, to each element in the range [first, last).
generate(V1.begin(), V1.end(), rand);

// Fill assigns the value to every element in the range
// [first, last). That is, for every iterator i in [first, last),
// it performs the assignment *i = value.
fill(V2.begin(), V2.end(), 0);

// Fill_n assigns the value to every element in the range
// [first, first+n). That is, for every iterator i in [first, first+n),
// it performs the assignment *i = value.
// The return value is first + n.
fill_n(back_inserter(V2), 5, 42);

// Transform performs an operation on objects.
// It performs the operation op(*i1, *i2) for each iterator i1 in the
// range [first1, last1) and assigns the result to *o, where i2 is the
// corresponding iterator in the second input range and where o is the
// corresponding output iterator. That is, for each n such that
// 0 <= n < last1 - first1, it performs the assignment
// *(result + n) = op(*(first1 + n), *(first2 + n). The return value is
// result + (last1 - first1).
transform(V1.begin(), V1.end(), V2.begin(), negate<int>());
// If f is an object of class negate<T> and x is an object of class T,
// then f(x) returns -x.

cout << "Result of negate on randomly generated values: ";
copy(V2.begin(), V2.end(), ostream_iterator<int>(cout, " "));
cout << "\nThe last 42 were filled with fill_n function."
   << endl << endl;
// OUTPUT: Result of negate on randomly generated values: -41 -18467
// -6334 -26500 -19169 -1 5724 -11478 -29358 -26962 -24464 42 42 42 42 42
// The last 42 were filled with fill_n function.

// mismatch
int A1[] = { 3, 1, 4, 1, 5, 9, 3 };
int A2[] = { 3, 1, 4, 2, 8, 5, 7 };
const int NMIS = sizeof(A1) / sizeof(int);

// Mismatch finds the first position where the two ranges [first1, last1)
// and [first2, first2 + (last1 - first1)) differ.
```

```
pair<int*, int*> result = mismatch(A1, A1 + NMIS, A2);
// Pair<T1,T2> is a heterogeneous pair: it holds one object of type T1
// and one of type T2. A pair is much like a Container, in that it "owns"
// its elements. It is not actually a model of Container, though, because
// it does not support the sDarkorangedard methods (such as iterators)
// for accessing the elements of a Container.
cout << "The first mismatch is in position " << (int)(result.first - A1)
  << endl;
cout << "Values are: " << *(result.first) << ", " << *(result.second)
  << endl << endl;
// OUTPUT: The first mismatch is in position 3
// Values are: 1, 2

// search
const char S1[] = "Hello, world!";
const char S2[] = "world";
const int N1 = sizeof(S1) - 1;
const int N2 = sizeof(S2) - 1;

// Search finds a subsequence within the range [first1, last1) that is
// identical to [first2, last2) when compared element-by-element. It
// returns an iterator pointing to the beginning of that subsequence,
// or else last1 if no such subsequence exists
const char* p = search(S1, S1 + N1, S2, S2 + N2);
cout << "Found subsequence \"" << S2 << "\" at character " << (int)(p - S1)
  << " of sequence \"" << S1 << "\"." << endl << endl;
// OUTPUT: Found subsequence "world" at character 7 of sequence
// "Hello, world!".

// equal
int AE1[] = { 3, 1, 4, 1, 5, 9, 3 };
int AE2[] = { 3, 1, 4, 2, 8, 5, 7 };
const int NEQU = sizeof(AE1) / sizeof(int);

// Equal returns true if the two ranges [first1, last1) and
// [first2, first2 + (last1 - first1)) are identical when compared
// element-by-element, and otherwise returns false.
cout << "Result of comparison: " << equal(AE1, AE1 + NEQU, AE2)
  << endl << endl;
// OUTPUT: Result of comparison: 0

// for_each
int AFOREACH[] = {1, 4, 2, 8, 5, 7};
const int NFOREACH = sizeof(A) / sizeof(int);

// For_each applies the function object f to each element in the range
// [first, last); f's return value, if any, is ignored. Applications are
// performed in forward order,  i.e. from first to last. For_each returns
// the function object after it has been applied to each element.
print<int> P = for_each(AFOREACH, AFOREACH + NFOREACH, print<int>(cout));
cout << endl << P.count << " objects printed." << endl << endl;
// OUTPUT: 1 4 2 8 5
// 5 objects printed.

cout << "Swapping contents of vector V2 to V1: ";
// Swap Assigns the contents of a to b and the contents of b to a. This
// is used as a primitive operation by many other algorithms.
swap(V1, V2);
```

```
copy(V1.begin(), V1.end(), ostream_iterator<int>(cout, " "));
cout << endl << endl;
// OUTPUT: Swapping contents of vector V2 to V1: -41 -18467 -6334 -26500
// -19169 -15724 -1147 8 -29358 -26962 -24464 42 42 42 42 42

// Sort sorts the elements in [first, last) into ascending order,
// meaning that if i and j are any two valid iterators in [first, last)
// such that i precedes j, then *j is not less than *i. Note: sort is
// not guaranteed to be stable. That is, suppose that *i and *j are
// equivalent: neither one is less than the other. It is not guaranteed
// that the relative order of these two elements will be preserved by sort.
cout << "Sorted array 1: ";
sort(AE1, AE1 + NEQU);
copy(AE1, AE1 + NEQU, ostream_iterator<int>(cout, " "));
cout << endl << endl;
// OUTPUT: Sorted array 1: 1 1 3 3 4 5 9
cout << "Sorted array 2: ";
sort(AE2, AE2 + NEQU);
copy(AE2, AE2 + NEQU, ostream_iterator<int>(cout, " "));
cout << endl << endl;
// OUTPUT: Sorted array 2: 1 2 3 4 5 7 8

// Merge combines two sorted ranges [first1, last1) and [first2, last2)
// into a single sorted range. That is, it copies elements from
// [first1, last1) and [first2, last2) into
// [result, result + (last1 - first1) + (last2 - first2)) such that the
// resulting range is in ascending order. Merge is stable, meaning both
// that the relative order f elements within each input range is
// preserved, and that for equivalent elements in both input ranges the
// element from the first range precedes the element from the second.
// The return value is result + (last1 - first1) + (last2 - first2).
cout << "Merged sorted array: ";
merge(AE1, AE1 + NEQU, AE2, AE2 + NEQU,
  ostream_iterator<int>(cout, " "));
cout << endl << endl;
// OUTPUT: Merged sorted array: 1 1 1 2 3 3 3 4 4 5 5 7 8 9

// Binary_search is a version of binary search: it attempts to find the
// element value in an ordered range [first, last) It returns true if
// an element that is equivalent to value is present in [first, last)
// and false if no such element exists.
cout << "Searching for 7 in AE1: "
  << (binary_search(AE1, AE1 + NEQU, 7) ? "present" : "not present")
  << endl << endl;
// OUTPUT: Searching for 7 in AE1: not present

int AI1[] = { 1, 2, 3, 4, 5, 6, 7 };
int AI2[] = { 1, 4, 7 };

const int NI1 = sizeof(AI1) / sizeof(int);
const int NI2 = sizeof(AI2) / sizeof(int);

// Includes tests whether one sorted range includes another sorted
// range. That is, it returns true if and only if, for every element
// in [first2, last2), an equivalent element is also present
// in [first1, last1). Both [first1, last1) and [first2, last2)
// must be sorted in ascending order.
cout << "AI2 contained in AI1: "
```

```
  << (includes(AI1, AI1 + NI1, AI2, AI2 + NI2) ? "true" : "false")
  << endl << endl;
// OUTPUT: AI2 contained in AI1: true

// Ptr_fun takes a function pointer as its argument and returns a
// function pointer adaptor, a type of function object. It is actually
// two different functions, not one (that is, the name ptr_fun is
// overloaded). If its argument is of type Result (*)(Arg) then ptr_fun
// creates a pointer_to_unary_function, and if its argument is of type
// Result (*)(Arg1, Arg2) then ptr_fun creates a
// pointer_to_binary_function.
vector <char*> vec1;
vector <char*>::iterator Iter1, RIter;

vec1.push_back ( "Open" );
vec1.push_back ( "up" );
vec1.push_back ( "the" );
vec1.push_back ( "pearly" );
vec1.push_back ( "gates" );

cout << "Original sequence contains: " ;
for ( Iter1 = vec1.begin( ) ; Iter1 != vec1.end( ) ; Iter1++ )
  cout << *Iter1 << " ";
cout << endl;
// OUTPUT: Original sequence contains: Open up the pearly gates

// To search the sequence for "pearly"
// use a pointer_to_function conversion
// find_if returns the first iterator i in the range [first, last)
// such that pred(*i) is true.
// Returns last if no such iterator exists.
RIter = find_if( vec1.begin( ), vec1.end( ),
  not1 ( bind2nd (ptr_fun ( strcmp ), "pearly" ) ) );

if ( RIter != vec1.end( ) )
{
  cout << "The search for 'pearly' was successful.\n";
  cout << "The next character string is: "
    << *++RIter << "." << endl << endl;
}
// OUTPUT: The search for 'pearly' was successful.
// The next character string is: gates.

// Set_union constructs a sorted range that is the union of the sorted
// ranges [first1, last1) and [first2, last2). The return value is the
// end of the output range.
int AS1[] = {1, 3, 5, 7, 9, 11};
int AS2[] = {1, 1, 2, 3, 5, 8, 13};

const int SN1 = sizeof(AS1) / sizeof(int);
const int SN2 = sizeof(AS2) / sizeof(int);

cout << "Union of AS1 and AS2: ";
set_union(AS1, AS1 + SN1, AS2, AS2 + SN2,
  ostream_iterator<int>(cout, " "));
cout << endl << endl;
// OUTPUT: Union of AS1 and AS2: 1 1 2 3 5 7 8 9 11 13
```

```
  // Set_intersection constructs a sorted range that is the intersection
  // of the sorted ranges [first1, last1) and [first2, last2). The return
  // value is the end of the output range.
  char AS3[] = {'a', 'b', 'b', 'B', 'B', 'f', 'h', 'H'};
  char AS4[] = {'A', 'B', 'B', 'C', 'D', 'F', 'F', 'H' };
  const int SN3 = sizeof(AS3);
  const int SN4 = sizeof(AS4);

  cout << "Intersection of AS3 and AS4: ";
  set_intersection(AS3, AS3 + SN3, AS4, AS4 + SN4,
    ostream_iterator<char>(cout, " "), lt_nocase);
  cout << endl << endl;
  // OUTPUT: Intersection of AS3 and AS4: a b b f h

  // Set_difference constructs a sorted range that is the set difference
  // of the sorted ranges [first1, last1) and [first2, last2). The return
  // value is the end of the output range.
  cout << "Difference of AS3 and AS4: ";
  set_difference(AS3, AS3 + SN3, AS4, AS4 + SN4,
    ostream_iterator<char>(cout, " "), lt_nocase);
  cout << endl << endl;
  // OUTPUT: Difference of AS3 and AS4: B B H
  return 0;
}
```