```
/*
 * STL: Standard Template Library
 *   -- Data Structures and Algorithms
 */


// First Example:
using namespace std;

vector<int> vec;
vec.push_back(4);
vec.push_back(1);
vec.push_back(8);  // vec: {4, 1, 8}

vector<int>::iterator itr1 = vec.begin();  // half-open:  [begin, end)
vector<int>::iterator itr2 = vec.end();

for (vector<int>::iterator itr = itr1; itr!=itr2; ++itr)
   cout << *itr << " ";  // Print out:  4 1 8

sort(itr1, itr2);  // vec: {1, 4, 8}


/*
 * STL Headers
 */

#include <vector>
#include <deque>
#include <list>
#include <set>   // set and multiset
#include <map>   // map and multimap
#include <unordered_set>  // unordered set/multiset
#include <unordered_map>  // unordered map/multimap
#include <iterator>
#include <algorithm>
#include <numeric>    // some numeric algorithm
#include <functional>


/*
 * Vector
 */
vector<int> vec;   // vec.size() == 0
vec.push_back(4);
vec.push_back(1);
vec.push_back(8);  // vec: {4, 1, 8};    vec.size() == 3

// Vector specific operations:
cout << vec[2];     // 8  (no range check)
cout << vec.at(2);  // 8  (throw range_error exception of out of range)

for (int i; i < vec.size(); i++) {
   cout << vec[i] << " ";
```

```
for (list<int>::iterator itr = vec.begin(); itr!= vec.end(); ++itr)
   cout << *itr << " ";

for (it: vec)    // C++ 11
   cout << it << " ";

// Vector is a dynamically allocated contiguous array in memory
int* p = &vec[0];   p[2] = 6;




// Common member functions of all containers.
// vec: {4, 1, 8}
if (vec.empty()) { cout << "Not possible.\n"; }

cout << vec.size();   // 3

vector<int> vec2(vec);  // Copy constructor, vec2: {4, 1, 8}

vec.clear();    // Remove all items in vec;   vec.size() == 0

vec2.swap(vec);   // vec2 becomes empty, and vec has 3 items.


// Notes: No penalty of abstraction, very efficient.

/* Properties of Vector:
 * 1. fast insert/remove at the end: O(1)
 * 2. slow insert/remove at the begining or in the middle: O(n)
 * 3. slow search: O(n)
 */


/*
 * Deque
 */
deque<int> deq = { 4, 6, 7 };
deq.push_front(2);  // deq: {2, 4, 6, 7}
deq.push_back(3);   // deq: {2, 4, 6, 7, 3}

// Deque has similar interface with vector
cout << deq[1];  // 4


/* Properties:
 * 1. fast insert/remove at the begining and the end;
 * 2. slow insert/remove in the middle: O(n)
 * 3. slow search: O(n)
 */
```

```
/*
 * list
 *  -- double linked list
 */
list<int> mylist = {5, 2, 9 };
mylist.push_back(6);  // mylist: { 5, 2, 9, 6}
mylist.push_front(4); // mylist: { 4, 5, 2, 9, 6}


list<int>::iterator itr = find(mylist.begin(), mylist.end(), 2); // itr -
> 2
mylist.insert(itr, 8);   // mylist: {4, 5, 8, 2, 9, 6}
                         // O(1), faster than vector/deque
itr++;                   // itr -> 9
mylist.erase(itr);       // mylist: {4, 8, 5, 2, 6}   O(1)


/* Properties:
 * 1. fast insert/remove at any place: O(1)
 * 2. slow search: O(n)
 * 3. no random access, no [] operator.
 */

mylist1.splice(itr, mylist2, itr_a, itr_b );   // O(1)


/*
 *  Associative Container
 *
 *  Always sorted, default criteria is <
 *
 *  No push_back(), push_front()
 *
 *
 */



/*
 * set
 *
 * - No duplicates
 */
  set<int> myset;
  myset.insert(3);    // myset: {3}
  myset.insert(1);    // myset: {1, 3}
  myset.insert(7);    // myset: {1, 3, 7},  O(log(n))

  set<int>::iterator it;
  it = myset.find(7);  // O(log(n)), it points to 7
                  // Sequence containers don't even have find() member
function
  pair<set<int>::iterator, bool> ret;
  ret = myset.insert(3);  // no new element inserted
```

```
  if (ret.second==false)
     it=ret.first;       // "it" now points to element 3

  myset.insert(it, 9);  // myset:  {1, 3, 7, 9}   O(log(n)) => O(1)
                         // it points to 3
  myset.erase(it);         // myset:  {1, 7, 9}

  myset.erase(7);   // myset:  {1, 9}
     // Note: none of the sequence containers provide this kind of erase.



// multiset is a set that allows duplicated items
multiset<int> myset;

// set/multiset: value of the elements cannot be modified
*it = 10;  // *it is read-only


/* Properties:
 * 1. Fast search: O(log(n))
 * 2. Traversing is slow (compared to vector & deque)
 * 3. No random access, no [] operator.
 */




/*
 * map
 *
 * - No duplicated key
 */
map<char,int> mymap;
mymap.insert ( pair<char,int>('a',100) );
mymap.insert ( make_pair('z',200) );

map<char,int>::iterator it = mymap.begin();
mymap.insert(it, pair<char,int>('b',300));  // "it" is a hint

it = mymap.find('z');  // O(log(n))

// showing contents:
for ( it=mymap.begin() ; it != mymap.end(); it++ )
  cout << (*it).first << " => " << (*it).second << endl;


// multimap is a map that allows duplicated keys
multimap<char,int> mymap;

// map/multimap:
//  -- keys cannot be modified
//     type of *it:   pair<const char, int>
     (*it).first = 'd';  // Error
```

```
// Associative Containers: set, multiset, map, multimap
//
// What does "Associative" mean?


/*
 *  Unordered Container (C++ 11)
 *    - Unordered set and multiset
 *    - Unordered map and multimap

 *  Order not defined, and may change overtime
 *
 *  Default hash function defined for fundamental types and string.
 *
 *  No subscript operator[] or at()
 *  No push_back(), push_front()
 */

/*
 * unordered set
 */
  unordered_set<string> myset = { "red","green","blue" };
  unordered_set<string>::const_iterator itr = myset.find ("green"); //
O(1)
  if (itr != myset.end())   // Important check
     cout << *itr << endl;
  myset.insert("yellow");  // O(1)

  vector<string> vec = {"purple", "pink"};
  myset.insert(vec.begin(), vec.end());

// Hash table specific APIs:
  cout << "load_factor = " << myset.load_factor() << endl;
  string x = "red";
  cout << x << " is in bucket #" << myset.bucket(x) << endl;
  cout << "Total bucket #" << myset.bucket_count() << endl;


// unordered multiset: unordered set that allows duplicated elements
// unordered map: unordered set of pairs
// unordered multimap: unordered map that allows duplicated keys


// hash collision => performance degrade


/* Properties of Unordered Containers:
 * 1. Fastest search/insert at any place: O(1)
 *     Associative Container takes O(log(n))
 *     vector, deque takes O(n)
 *     list takes O(1) to insert, O(n) to search
 * 2. Unorderd set/multiset: element value cannot be changed.
 *    Unorderd map/multimap: element key cannot be changed.
 */
```

```
/*
 * Associative Array
 * - map and unordered map
 */
unordered_map<char, string> day = {{'S',"Sunday"}, {'M',"Monday"}};

cout << day['S'] << endl;    // No range check
cout << day.at('S') << endl; // Has range check

vector<int> vec = {1, 2, 3};
vec[5] = 5;   // Compile Error

day['W'] = "Wednesday";  // Inserting {'W', "Wednesday}
day.insert(make_pair('F', "Friday"));  // Inserting {'F', "Friday"}

day.insert(make_pair('M', "MONDAY"));  // Fail to modify, it's an
unordered_map
day['M'] = "MONDAY";                   // Succeed to modify



void foo(const unordered_map<char, string>& m) {
   //m['S'] = "SUNDAY";
   //cout << m['S'] << endl;
   auto itr = m.find('S');
   if (itr != m.end())
      cout << *itr << endl;
}
foo(day);


//   cout << m['S'] << endl;
//   auto itr = m.find('S');
//   if (itr != m.end() )
//      cout << itr->second << endl;


//Notes about Associative Array:
//1. Search time: unordered_map, O(1); map, O(log(n));
//2. Unordered_map may degrade to O(n);
//3. Can't use multimap and unordered_multimap, they don't have []
operator.

/*
 * Array
 */
int a[3] = {3, 4, 5};
array<int, 3> a = {3, 4, 5};
a.begin();
a.end();
a.size();
a.swap();
array<int, 4> b = {3, 4, 5};
```

```
/*
 * Container Adaptor
 *  - Provide a restricted interface to meet special needs
 *  - Implemented with fundamental container classes
 *
 *  1. stack:  LIFO, push(), pop(), top()
 *
 *  2. queue:  FIFO, push(), pop(), front(), back()
 *
 *  3. priority queue: first item always has the greatest priority
 *                     push(), pop(), top()
 */


/*
 * Another way of categorizing containers:
 *
 * 1. Array based containers: vector, deque
 *
 * 2. Node base containers: list + associative containers + unordered
containers
 *
 * Array based containers invalidates pointers:
 *    - Native pointers, iterators, references
 */

 vector<int> vec = {1,2,3,4};
 int* p = &vec[2];   // p points to 3
 vec.insert(vec.begin(), 0);
 cout << *p << endl;   // 2, or ?


/*
 * Iterators
 */
// 1. Random Access Iterator:  vector, deque, array
vector<int> itr;
itr = itr + 5;  // advance itr by 5
itr = itr - 4;
if (itr2 > itr1) ...
++itr;   // faster than itr++
--itr;

// 2. Bidirectional Iterator: list, set/multiset, map/multimap
list<int> itr;
++itr;
--itr;

// 3. Forward Iterator: forward_list
forward_list<int> itr;
++itr;

// Unordered containers provide "at least" forward iterators.
```

```
// 4. Input Iterator: read and process values while iterating forward.
int x = *itr;

// 5. Output Iterator: output values while iterating forward.
*itr = 100;


// Every container has a iterator and a const_iterator
set<int>::iterator itr;
set<int>::const_iterator citr;  // Read_only access to container elements

set<int> myset = {2,4,5,1,9};
for (citr = myset.begin(); citr != myset.end(); ++citr) {
   cout << *citr << endl;
   //*citr = 3;
}
for_each(myset.cbegin(), myset.cend(), MyFunction);  // Only in C++ 11



// Iterator Functions:
advance(itr, 5);       // Move itr forward 5 spots.   itr += 5;
distance(itr1, itr2);  // Measure the distance between itr1 and itr2


/* Iterator Adaptor (Predefined Iterator)
 *  - A special, more powerful iterator
 * 1. Insert iterator
 * 2. Stream iterator
 * 3. Reverse iterator
 * 4. Move iterator (C++ 11)
 */

// 1. Insert Iterator:
vector<int> vec1 = {4,5};
vector<int> vec2 = {12, 14, 16, 18};
vector<int>::iterator it = find(vec2.begin(), vec2.end(), 16);
insert_iterator< vector<int> > i_itr(vec2,it);
copy(vec1.begin(),vec1.end(),  // source
     i_itr);                   // destination
     //vec2: {12, 14, 4, 5, 16, 18}
// Other insert iterators: back_insert_iterator, front_insert_iterator


// 2. Stream Iterator:
vector<string> vec4;
copy(istream_iterator<string>(cin), istream_iterator<string>(),
            back_inserter(vec4));

copy(vec4.begin(), vec4.end(), ostream_iterator<string>(cout, " "));

// Make it terse:
copy(istream_iterator<string>(cin), istream_iterator<string>(),
            ostream_iterator<string>(cout, " "));
```

```
// 3. Reverse Iterator:
vector<int> vec = {4,5,6,7};
reverse_iterator<vector<int>::iterator> ritr;
for (ritr = vec.rbegin(); ritr != vec.rend(); ritr++)
   cout << *ritr << endl;   // prints: 7 6 5 4


/*
 * Algorithms
 *   - mostly loops
 */
vector<int> vec = { 4, 2, 5, 1, 3, 9};
vector<int>::iterator itr = min_element(vec.begin(), vec.end()); // itr -
> 1

// Note 1: Algorithm always process ranges in a half-open way: [begin,
end)
sort(vec.begin(), itr);  // vec: { 2, 4, 5, 1, 3, 9}

reverse(itr, vec.end());  // vec: { 2, 4, 5, 9, 3, 1}   itr => 9


// Note 2:
vector<int> vec2(3);
copy(itr, vec.end(),  // Source
     vec2.begin());   // Destination
     //vec2 needs to have at least space for 3 elements.



// Note 3:
vector<int> vec3;
copy(itr, vec.end(), back_inserter(vec3));  // Inserting instead of
overwriting
                  // back_insert_iterator      Not efficient

vec3.insert(vec3.end(), itr, vec.end());  // Efficient and safe



// Note 4: Algorithm with function
bool isOdd(int i) {
   return i%2;
}

int main() {
   vector<int> vec = {2, 4, 5, 9, 2}
   vector<int>::iterator itr = find_if(vec.begin(), vec.end(), isOdd);
                                   // itr -> 5
}

// Note 5: Algorithm with native C++ array
int arr[4] = {6,3,7,4};
sort(arr, arr+4);
```

```
// Vector pitfalls:
//
// Reallocate vector
// Remove items


/*
 * Reasons to use C++ standard library:
 * 1. Code reuse, no need to re-invent the wheel.
 * 2. Efficiency (fast and use less resources). Modern C++ compiler are
usually
 *    tuned to optimize for C++ standard library code.
 * 3. Accurate, less buggy.
 * 4. Terse, readable code; reduced control flow.
 * 5. Standardization, guarenteed availability
 * 6. A role model of writing library.
 * 7. Good knowledge of data structures and algorithms.
 */


/*
 * vector
 */
  class Dog;
// Example 1:
  vector<Dog> vec(6);  // vec.capacity() == 6, vec.size() == 6,
                       // 6 Dogs created with default constructor

// Example 2:
  vector<Dog> vec; // vec.capacity() >= 0, vec.size() == 0
  vec.resize(6);   // vec.capacity() >= 6, vec.size() == 6,
                   // 6 Dogs created with default constructor

// Example 3:
  vector<Dog> vec;
  vec.reserve(6);   // vec.capacity() >= 6, vec.size() == 0,
                    // no default constructor invoked

/*
 * Strategy of avoiding reallocation:
 * 1. If the maximum number of item is known, reserve(MAX);
 * 2. If unknown, reserve as much as you can, once all data a inserted,
 *    trim off the rest.
 */

/*
 * deque
 *
 * - No reallocation
 *   deque has no reserve() and capacity()
 * - Slightly slower than vector
 */
```

```
/*
 * Which one to use?
 *
 * - Need to push_front a lot?  -> deque
 *
 * - Performance is important?   -> vector
 */


/*
 * 1. Element type
 *   - When the elements are not of a trivial type, deque is not much
less
 *   efficient than vector.
 */


/*
 * 2. Memory Availability
 *   Could allocation of large contiguous memory be a problem?
 *   - Limited memory size
 *   - Large trunk of data
 */


/*
 * 3. Frequency of Unpredictable Growth
 */
  vector<int> vec;
  for (int x=0; x<1025; x++)
     vec.push_back(x);   // 11 reallocations performed (growth ratio = 2)

            //   workaround: reserve()

/*
 * 4. Invalidation of pointers/references/iterators because of growth
 */
  vector<int> vec = {2,3,4,5};
  int* p = &vec[3]
  vec.push_back(6);
  cout << *p << endl;  // Undefined behavior

  deque<int> deq = {2,3,4,5};
  p = &deq[3];
  deq.push_back(6);
  cout << *p << endl;  // OK
  // push_front() is OK too
  // deque: inserting at either end won't invalidate pointers

// Note: removing or inserting in the middle still will invalidate
//       pointers/references/iterators

/*
 * 5. Vector's unique feature: portal to C
```

```
 */
  vector<int> vec = {2,3,4,5};

  void c_fun(const int* arr, int size);

  c_fun(&vec[0], vec.size());

  // Passing data from a list to C
  list<int> mylist;
  ...
  vector<int> vec(mylist.gegin(), mylist.end());
  c_fun(&vec[0], vec.size());

  // NOTE: &vector[0] can be used as a raw array.
  // Exception: vector<bool>
  void cpp_fun(const bool* arr, int size);
  vector<bool> vec = {true, true, false, true};
  cpp_fun(&vec[0], vec.size()); // Compiler Error: &vec[0] is not a bool
pointer

       // workaround: use vector<int>, or bitset


/*
 * Summary:
 * 1. Frequent push_front()    - deque
 * 2. High performance         - vector
 * 3. Non-trivial data type    - deque
 * 4. Contiguous memory        - deque
 * 5. Unpredictable growth     - deque
 * 6. Pointer integrity        - deque
 * 7. Talk to C                - vector
 */


  // Backups
  vector<int> vec = {2,3,4,5};

  cout << vec.capacity() << endl;

  vec.push_back(6);


/*
 * Vector Reallocation:
 * 1. A new memory space of 8 int is allocated (Assume growth factor is
2)
 * 2. A new vector is constructed with {2,3,4,5,6} at the new memory
space.
 * 3. The old memory is release.
 */
```