UEE1303
Objective-Oriented Programming

C++_Lecture 09:

Standard Template Library

C++: How to Program (Late Objects Version) 7th ed.

Agenda

- Understand basic data structure
  - Linked list vs. Array
  - Stacks & Queues
  - Trees
- Standard Template Library (STL)
  - Iterator
  - Containers
  - Generic Algorithms

Data Structure

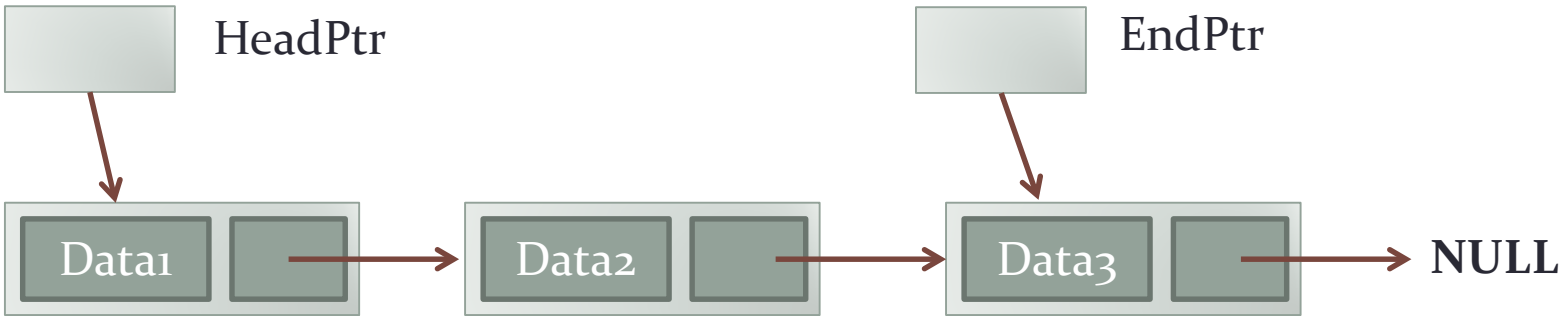
- A means of representing data to achieve efficiency in terms of
  - time complexity: to minimize runtime
  - memory complexity: to minimize memory usage
- A means of demonstrating relationships between data elements
- A means of enforcing a processing order
- A means of modeling real-world problems

Linked Lists vs. Arrays

- Placement and access of storage
  - Arrays: use contiguous storage + random access
  - Linked lists: use non-contiguous storage + sequential access
- Volume of storage
  - Arrays: fixed size
  - Linked lists: expand and contract as needed

Fundamental Linked List

- Made up of a series of dynamically allocated nodes, each containing a link to the next node
- Managed by a single pointer to the first node, and possibly a second pointer to the last
  - `HeadPtr`
  - `EndPtr` (optional)



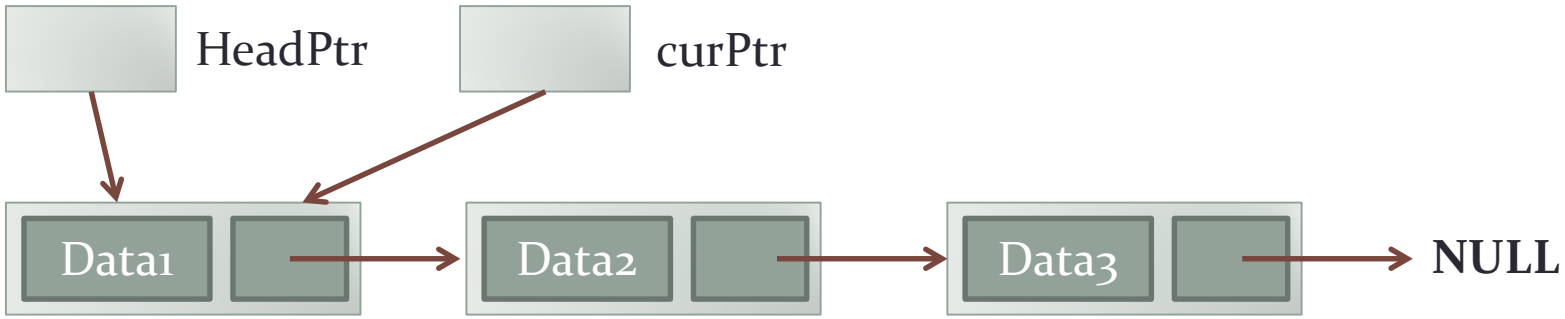
Linked List Usage

- Sequential Access
  - Each access must start at first node and traverse all others until the desired node is found
  - An illusion of random access can be created by

```
dataType getElement(int index);
```

- Any process that traverses the list or affects all elements in the list must use a secondary pointer to refer to "the *current* element"
  - `curPtr` (current pointer)

Linked List Traversal



- `curPtr` points to the element currently begin operated on
- Can traverse the list by following each node's `next` pointer
  - `curPtr = curPtr - > next;`

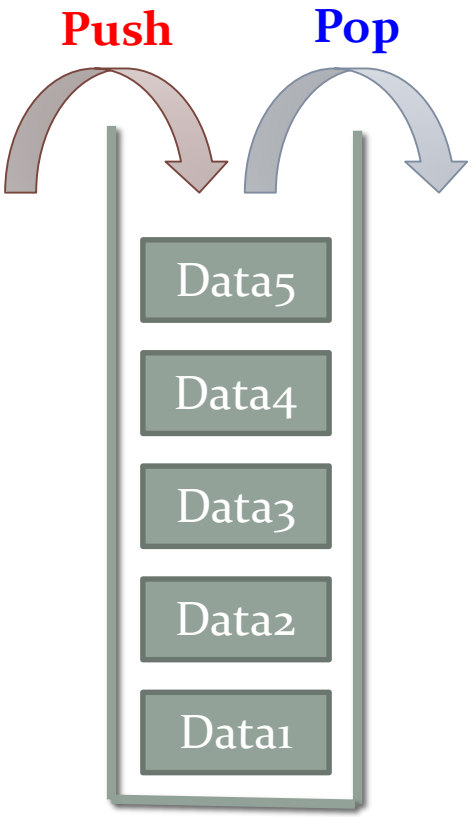
Stacks & Queues

- Establish and enforce an order for processing

- Stack
  - *Last in*, first out (LIFO)
- Queue
  - *First in*, first out (FIFO)
  - Priority Queue
- Deque (pronounced ‘deck’ or ‘de-Q’)
  - Double-ended queue

Stacks

- Add element
  - Push
- Remove element
  - Pop
- Only top element can be seen
  - First-in, last-out (FILO)



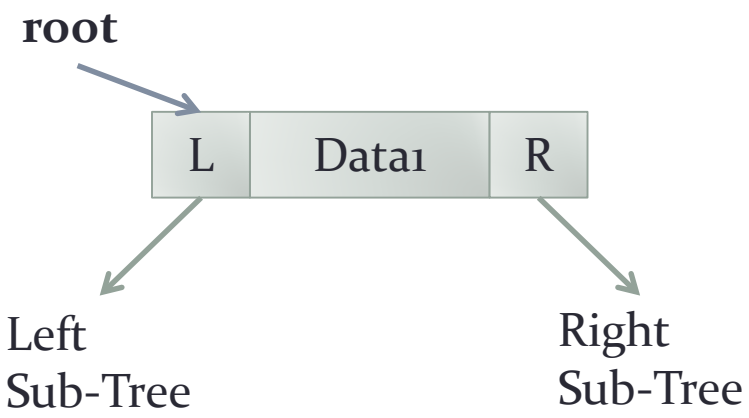
Queues

- FIFO: first-in first-out
- Can use array or linked list internally
  - Internal links are unreachable

Deque

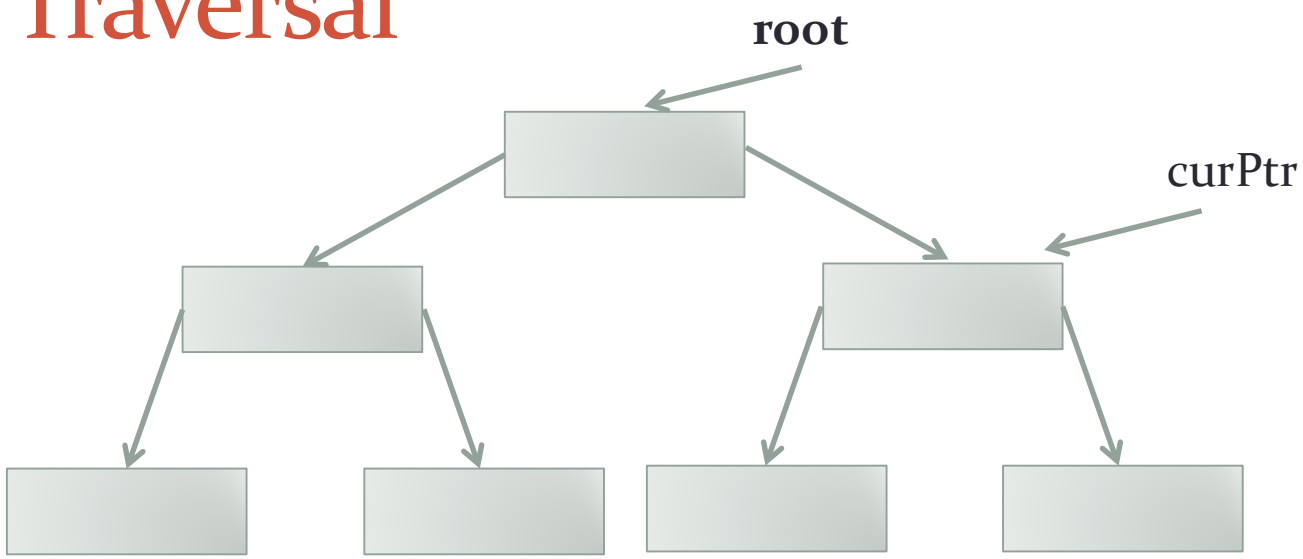
- Deque is a special type of queue
  - Data can be added/deleted from either end
  - Internal nodes are still inaccessible

Trees



- A "`root`" node with some number (usually 2) of links (pointers) to "`child`" nodes
- Each child node is the root of a "sub-tree"
  - inherently recursive in nature
- Many flavors of trees to choose from

Tree Traversal



- Binary trees (2 children) are the most common type
  - Access by `root`
- Use a reference pointer to access sub-nodes during traversal
  - `curPtr`

What is STL?

- STL = Standard Template Library
  - Based (heavily) on template programming
  - With a guarantee of performance
- A general-purpose library of generic algorithms and data structures in C++
  - container classes
  - generic algorithms
  - other components (ex: iterators)
- Part of the ISO Standard C++ Library

Why should I Use STL?

- Reduce development time
  - Data-structures already well-written and thoroughly debugged
- Code readability
  - Fit more meaningful stuff on one or a few pages
- Robustness
  - STL data structure grow automatically
- Portable code
- Maintainable code
- Easy to use

STL Basics

- Container Classes
  - Class templates which implement many of the classic data structure
- Iterators (a.k.a. *smart pointers*)
  - referencing devices similar to the pointers to the individual elements used in data structures
- Generic Algorithms
  - Template functions that implement frequently used algorithms (i.e., sorting, searching, etc.)

Iterators

- Container class provides its own iterator class
  - Implemented as a sub-class
  - Usage and syntax is very *similar to* that of pointers
- The container class provides methods for creating, initializing, and controlling iterators
  - `begin()`: return an iterator that points to the first element
  - `end()`: return an iterator that point just "behind" the last element
- Does NOT reference the last element

Iterators (cont.)

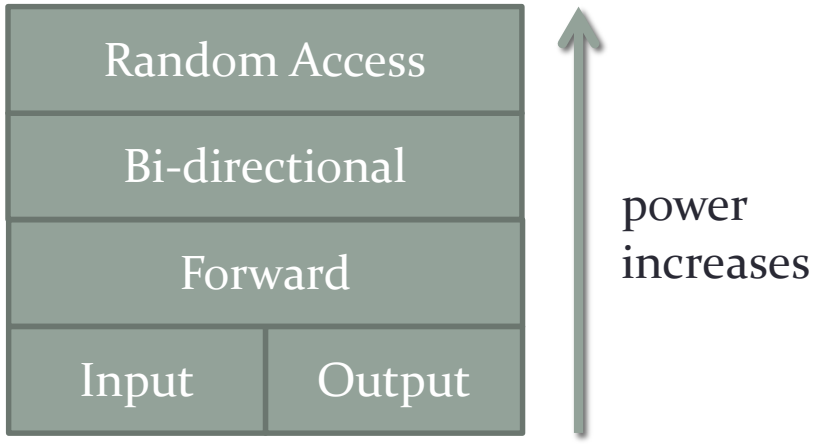
- Prefix and postfix increment and decrement
  - `++` moves the iterator to the next element
  - `--` moves the iterator to the previous element
- Equals and not-equals
  - `==` & `!=` compares two iterators
- Deference
  - `*` & `[]` returns the referenced data item
- Not all iterators provide all of these functions

Example of `vector`

```
#include <iostream>
#include <vector>
int main()
{
    vector<int> v1(10);
    vector<int>::iterator i;
    int n = 0;
    for (i = v1.begin(); i != v1.end(); i++)
        *i = n++;
    for (i = v1.begin(); i != v1.end(); i++)
        cout << *i << " ";
    cout << endl;
    return 0;
}
```

Basic Iterators

- Points to an object in a container
- Access to object by de-referencing
- Increment and decrement operators used to move forward and backward
- Category by move
  - input
  - output
  - forward
  - bi-directional
  - random-access



Ref: http://www.cplusplus.com/reference/iterator/

Type of Iterator Supported

Container	Type of iterator supported
Sequence containers (first class)	
`vector`	random access
`deque`	random access
`list`	bidirectional
Associative containers (first class)	
`set`	bidirectional
`multiset`	bidirectional
`map`	bidirectional
`multimap`	bidirectional
Container adapters	
`stack`	no iterators supported
`queue`	no iterators supported
`priority_queue`	no iterators supported

Input & Output Iterator

- Input iterator
  - used to read value from a sequence container
  - support dereference `*iter`, increment `++iter/iter++` and in-/equality `!=/==`

- Output iterator
  - used to write values once to a sequence container

Example of Stream Iterators

```
// Fig. 21.5: Fig21_05.cpp
#include <iostream>
#include <iterator>
using namespace std;
int main()
{
    cout << "Enter two integers: ";
    istream_iterator<int> inputInt(cin);
    int number1 = *inputInt;
    ++intputInt;
    int number2 = *inputInt;
    ostream_iterator<int> outputInt(cout);
    cout << "The sum is: ";
    *outputInt = number1 + number2;
    cout << endl;
    return 0;
}
```

Example of Stream Iterators (cont.)

- screen output

```
Enter two integers: 12 25
The sum is: 37
```

Forward Iterator

- Both input and output iterator
- Read and write in one direction
- Read and write multiple times
- All standard library containers use this iterators

```
vector<int> iv;
iv.push_back(5);
iv.push_back(6);
…
vector<int>::iterator i;
for (i = iv.begin(); i != iv.end(); i++)
    cout << *iv << " ";
```

Bi-directional & Random Iterators

- Bi-directional iterator
  - quite similar to forward iterator
  - can also be decremented `--iter/iter--`
  - read and write forward and backward
- Random access iterator
  - allow access to any element
  - compare iterators using < and >
  - does NOT work with `list`
  - e.g. `vector` and `string` iterators

Random Access Example

```
#include <iostream>
#include <vector>
using std::cout;
using std::endl;
using std::vector;

int main() {
    vector<char> container;

    container.push_back('A');
    container.push_back('B');
    container.push_back('C');
    container.push_back('D');
    for (int i = 0; i < 4; i++)
        cout << "container[" << i << "] = "
             << container[i] << endl;
```

Random Access Example (cont.)

```
vector<char>::iterator p =
                          container.begin( );
cout << "The third entry is "
     << container[2] << endl;
cout << "The third entry is "
      << p[2] << endl;  // random access
cout << "The third entry is "
     << *(p + 2) << endl; // random access
iterator
cout << "Back to container[0].\n";
p = container.begin();
cout << "which has value "
     << *p <<  endl;
cout << "Two steps forward and one step
                                  back:\n";
```

Random Access Example (cont.)

```
    p++;  cout << *p << endl;
    p++;  cout << *p << endl;
    p--;  cout << *p << endl;

    return 0;
}
```

Random Access Example (cont.)

```
Container[0] == A
Container[1] == B
Container[2] == C
Container[3] == D
The third entry is C
The third entry is C
The third entry is C
Back to container[0].
which has value A
Two steps forward and one step back:
B
C
B
```

Example of Iterator Usages

- Declare
  - `list<int>::iterator li;`
- Front of container
  - `list<int> L;`
  - `li = L.begin();`
- Past the end
  - `li = L.end();`
- Increment
  - `list<int>::iterator li;`
  - `list<int> L;`
  - `li=L.begin();`
  - `++li; // second item;`
- De-reference
  - `*li = 10;`

Constant and Mutable Iterators

- Constant iterator: (const_iterator)
  - `*` produces read-only version of element
  - can use `*p` to assign to variable or output, but cannot change element in container
    - E.g., `*p = <anything>;` is illegal

- Mutable iterator:
  - `*p` can be assigned value
  - changes corresponding element in container
  - i.e.: **`*p`** returns an lvalue `*p = <something>;`

Reverse Iterators

- To cycle elements in reverse order
  - requires container with *bidirectional* iterators

- Might consider:

```
iterator p;
for (p=container.end();p!=container.begin();
                                        p--)
    cout << *p << " " ;
```

  - but recall: `end()` is just "sentinel"
  - might work on some systems, but not most

Reverse Iterators (cont.)

- To correctly cycle elements in reverse order:

```
reverse_iterator rp;
for ( rp=container.rbegin();
                    rp!=container.rend(); rp++)
    cout << *rp << " " ;
```

- `rbegin()`
  - returns an iterator that points to the last element
- `rend()`
  - return an iterator that points just “before” the first element
  - i.e., returns sentinel "end" marker

Example of Reverse and Constant Iterators

```
#include <vector>
using namespace std
int main()
{
    vector<int> v2(10);
    vector<int>::const_iterator c = v2.begin();
    vector<int>::reverse_iterator i;
    int n = 0;
    for (i = v2.rbegin(); i != v2.rend(); i++)
        *i = n++;
    for (n = 0; n < 10; n++)
        cout << c[n] << " ";
    cout << endl;
    return 0;
}
```

Iterator Operations

Iterator operation	Description
All iterators	
`++p`	Preincrement an iterator.
`p++`	Postincrement an iterator.
Input iterators	
`*p`	Dereference an iterator.
`p = p1`	Assign one iterator to another.
`p == p1`	Compare iterators for equality.
`p != p1`	Compare iterators for inequality.
Output iterators	
`*p`	Dereference an iterator.
`p = p1`	Assign one iterator to another.
Forward iterators	Forward iterators provide all the functionality of both input iterators and output iterators.

Iterator Operations (cont.)

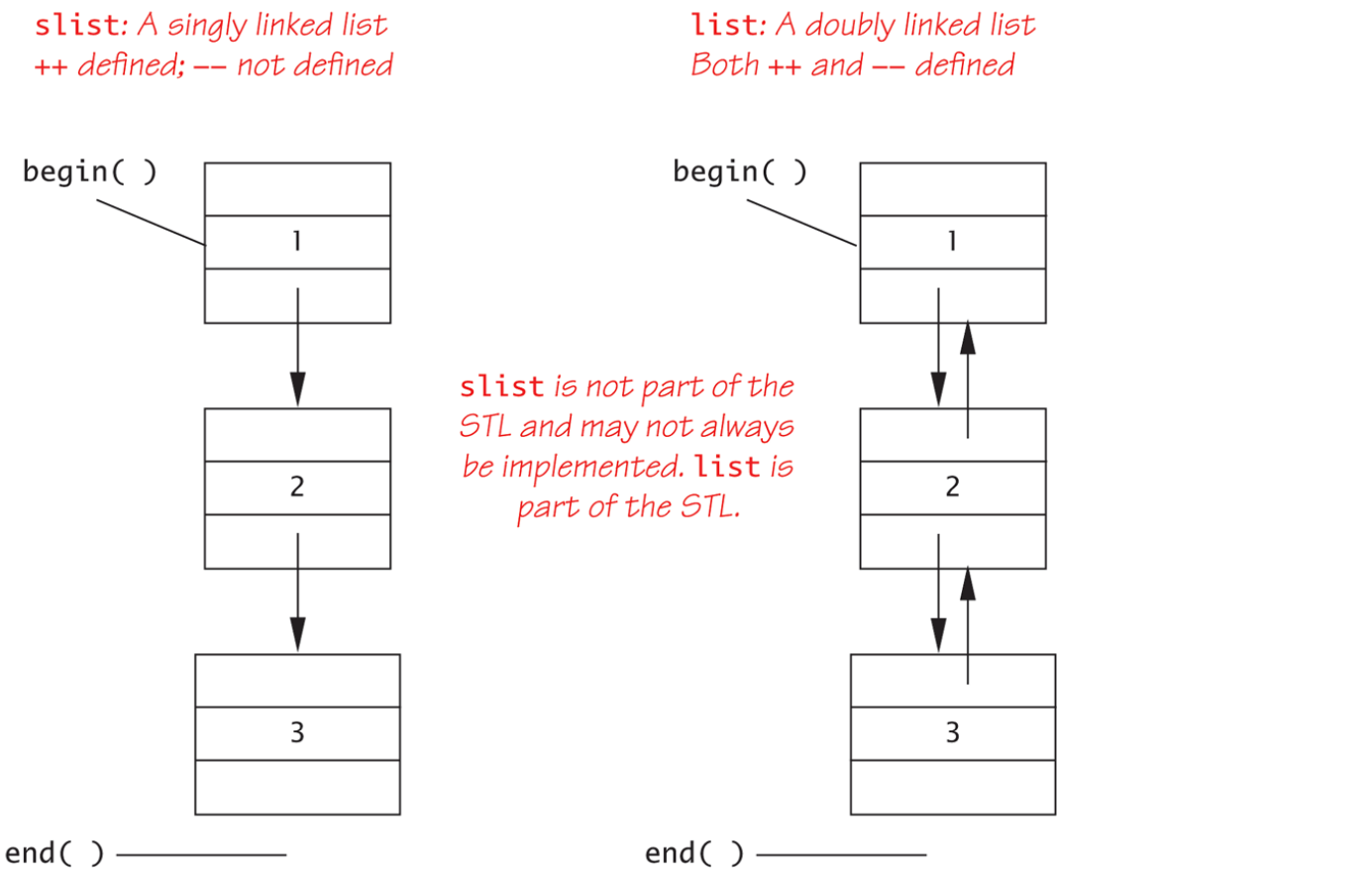
Iterator operation	Description
Bidirectional iterators	
`--p`	Predecrement an iterator.
`p--`	Postdecrement an iterator.
Random-access iterators	
`p += i`	Increment the iterator `p` by `i` positions.
`p -= i`	Decrement the iterator `p` by `i` positions.
`p + i` *or* `i + p`	Expression value is an iterator positioned at `p` incremented by `i` positions.
`p - i`	Expression value is an iterator positioned at `p` decremented by `i` positions.
`p - p1`	Expression value is an integer representing the distance between two elements in the same container.
`p[ i ]`	Return a reference to the element offset from `p` by `i` positions
`p < p1`	Return `true` if iterator `p` is less than iterator `p1` (i.e., iterator `p` is before iterator `p1` in the container); otherwise, return `false`.

Basic Containers in STL

- Sequential Containers
  - elements are stored in whatever order they were added, unless sorted manually
  - ex: `list`, `vector`, and `deque`
  - some implementations also provide `slist` (not part of STL standard)
- Associative Containers
  - elements are sorted automatically according to some key field
  - ex: `set` and `map`
- Top: `stack`, `vector` , `list` and `map`

Two Kinds of Lists

Display 19.4 Two Kinds of Lists



Standard Library container classes

Standard Library container class	Description
Sequence containers	
`vector`	Rapid insertions and deletions at back. Direct access to any element.
`deque`	Rapid insertions and deletions at front or back. Direct access to any element.
`list`	Doubly linked list, rapid insertion and deletion anywhere.
Associative containers	
`set`	Rapid lookup, no duplicates allowed.
`multiset`	Rapid lookup, duplicates allowed.
`map`	One-to-one mapping, no duplicates allowed, rapid key-based lookup.
`multimap`	One-to-many mapping, duplicates allowed, rapid key-based lookup.
Container adapters	
`stack`	Last-in, first-out (LIFO).
`queue`	First-in, first-out (FIFO).
`priority_queue`	Highest-priority element is always the first element out.

Standard Library container header files

Standard Library container header files

`<vector>`	
`<list>`	
`<deque>`	
`<queue>`	Contains both `queue` and `priority_queue`.
`<stack>`	
`<map>`	Contains both `map` and `multimap`.
`<set>`	Contains both `set` and `multiset`.
`<valarray>`	
`<bitset>`	

`vector` Container Class

- Provides random-access iterators
  - forward and reverse
  - mutable and constant

- Member functions support:
  - iterator manipulation
  - direct manipulation

Ref: http://www.cplusplus.com/reference/vector/vector/

Container `vector`

- Like array, `vectors` in concept are used to store a homogeneous sequence of data objects
- But unlike arrays, vectors have no a predefined value for *size limit*

- Therefore, `vector` is made into a class template
  - part of Standard Template Library (STL)

Memory Allocation for `vector`

- By default, `vector` objects are created with a size of 0
  - a parameter to the constructor can override this behavior
- The size of a vector object can be changed at run-time
  - `push_back(T)`: member function adds storage at the end of the sequence for one new element
  - `resize(int)`: member function sets size, adding or deleting elements as necessary

Constructors of `vector`

- Default Constructor
  - `vector <T> v1;`
  - e.g., `vector <int> v1;`
- Allocation Constructor
  - `vector <T> v2(int);`
  - e.g., `vector<int> v2(5);`
- Copy Constructor
  - `vector <T> v3(vector <T>);`
  - e.g., `vector<int> v3 (vector v2);`

`vector` Methods

- `size_type vector::capacity();`
  - return number of elements for which memory has been allocated
- `size_type vector::size();`
  - Return number of elements physically existing in the vector
- `void vector::resize(size_type n, T x = T());`
  - reallocate memory, preserves contents if new size is larger than existing size

Ref: http://www.cplusplus.com/reference/vector/vector/capacity/

Ref: http://www.cplusplus.com/reference/vector/vector/resize/

`vector` Methods (cont.)

- `void vector::push_back(const T& x);`
  - append (insert) an element to the end of a vector, allocating memory for it if necessary
- `void vector::pop_back();`
  - erase the last element of the vector
- `const T& operator[](size_type pos) const;`
- `T& operator[](size_type pos);`
  - constant and non-constant [] operator

`vector` Methods (cont.)

- `vector::erase()` or `vector::clear()`
  - erase all elements in the vector
- `void vector::erase(iterator);`
  - erase the element indexed by the iterator
- `void vector::erase(beg_iterator, end_iterator);`
  - erase elements between the begin iterator (beg_iterator) and the end iterator (end_iterator)
- `bool vector::empty();`
  - return true if vector has no elements

Ref: http://www.cplusplus.com/reference/vector/vector/erase/

Example of vector

```
// Fig. 21.14: Fig21_14.cpp
// Demonstrating Standard Library vector class template
#include <iostream>
#include <vector>
using namespace std;
// prototype for function template printVector
template <typename T> void printVector (const vector <T>
&integers2);

int main () {
    const int SIZE = 6;
    int array[ SIZE] = {1, 2, 3, 4, 5, 6};
    vector< int > integers; // create vector of ints
    cout << "The initial size of integers is: "
         << integers.size()
         << "\nThe initial capacity of integers is: "
         << integers.capacity();
```

Example of `vector` (cont.)

```
    // function push_back is in every sequence collection
    integers.push_back(2);
    integers.push_back(3);
    integers.push_back(4);
    cout << "The size of integers is: "
         << integers.size()
         << "\nThe capacity of integers is: "
         << integers.capacity();
         << "\n\nOutput array using pointer notations: ";
    // display array using pointer notation
    for ( int *ptr = array; ptr != array + SIZE; ptr++)
        cout << *ptr << ' ';
    cout << "\nOutput vector using iterator notation: ";
    printVector( integers );
    vector< int >::const_reverse_iterator reverseIterator;
    vector< int >::const_reverse_iterator tempIterator
                                        = integers.rend();
```

Example of `vector` (cont.)

```
    // display vector in reverse order
    for ( reverseIterator = integers.rbegin();
          reverseIterator != tempIterator;
          ++reverseIterator)
        cout << *reverseIterator << ' ';
    cout << endl;
}
// function template for outputting vector elements
template <typename T>
void printVector (const vector <T> &integers2)
{
    vector <T>::const_iterator constIterator;
    // display vector elements using const_iterator
    for ( constIterator = integer2.begin();
          constIterator != integer2.end();
          ++constIterator)
        cout << *constIterator << ' ';
} // end function printVector
```

Example of `vector` (cont.)

```
The initial size of integers is: 0
The initial capacity of integers is: 0
The size of integers is: 3
The capacity of integers is: 4

Output array using pointer notation: 1 2 3 4 5 6
Output vector using iterator notation: 2 3 4
Reversed contents of vector integers: 4 3 2
```

Example2 of `vector`

```
// Fig. 21.15: Fig21_15.cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>
#include <stdexcept> // out-of-range exception
using namespace std;

int main () {
    const int SIZE = 6;
    int array[ SIZE] = {1, 2, 3, 4, 5, 6};
    vector< int > integers (array, array + SIZE);
    ostream_iterator< int > output (cout, " ");

    cout << "Vector integers contains: "
    copy (integers.begin(), integers.end(), output);
```

Example2 of `vector` (cont.)

```
     cout << "\nFirst element of integers: "
          << integers.front()
          << "\nLast element of integers is: "
          << integers.back();
    integers[0] = 7; // set first element to 7
    // set element at position 2 to 10
    integers.at(2) = 10;
    // insert 22 as 2nd element
I   integers.insert( integers.begin() + 1, 22);
    cout << "\nCoutents of vector integers after changes:"
    copy (integers.begin(), integers.end(), output);
    try   // access out-of-range element
    {
        integers.at(100) = 777;
    }
    catch ( out_of_range &outOfRange)
    {
        cout << "\n\nException: " << outOfRange.what();
    }
```

Example2 of `vector` (cont.)

```
    // erase first element
    integers.earse( integers.begin() );
    cout << "\n\nVector integers after erasing first
            elements: ";
    copy (integers.begin(), integers.end(), output);
    // erase remaining elements
    integers.earse( integers.begin(), integers.end() );
    cout << "\nAfter erasing elements, vector integers "
         << ( integers.empty() ? "is" : "is not" )
         << " empty";
    // insert elements from array
    integers.insert(integers.begin(), array, array + SIZE);
    cout << "\n\nContens of vector integers before clear:"
    copy (integers.begin(), integers.end(), output);
    // empty integers;
    integers.clear();
    cout << "\nAfter clear, vector integers "
         << ( integers.empty() ? "is" : "is not" )<<"empty";
}
```

Example2 of `vector` (cont.)

```
Vector integers contatins: 1 2 3 4 5 6
First element of integers: 1
Las element of integers: 6

Contents of vector integers after changes: 7 22 2 10 4 5 6

Exception: invalid vector<T> subscript

Vector integers after erasing first element: 22 2 10 4 5 6
After erasing all elements, vector integers is empty

Contents of vector integers before clear: 1 2 3 4 5 6
After clear, vector integers is empty
```

`list` Container Class

- Doubly-linked list
- Provides bi-directional iterators
  - forward and reverse
  - mutable and constant

- Member functions support:
  - iterator manipulation
  - direct manipulation

Ref: http://www.cplusplus.com/reference/list/list/

`list` Member Functions

- Iterator manipulation
  - `begin()`, `end()`, `rbegin()` and `rend()`
  - `insert()` and `erase()`

- Direct manipulation
  - `push_back()` and `push_front()`
  - `pop_back()` and `pop_front()`
  - `front()`, `back()` and `clear()`

Example of `list`

```
#include <iostream>
#include <list>
using std::cout;
using std::endl;
using std::list;

int main( )
{
    list<int> listObject;

    for (int i = 1; i <= 3; i++)
        listObject.push_back(i);
    cout << "List contains:\n";
    list<int>::iterator iter;
    for ( iter = listObject.begin();
          iter != listObject.end(); iter++)
        cout << *iter << " ";
    cout << endl;
```

Example of `list` (cont.)

```
    cout << "Setting all entries to 0:\n";
    for ( iter = listObject.begin();
          iter != listObject.end(); iter++)
        *iter = 0;
    cout << "List now contains:\n";
    for ( iter = listObject.begin();
          iter != listObject.end(); iter++)
        cout << *iter << " ";
    cout << endl;
    return 0;
}
```

```
List contains
1 2 3
Setting all entries to 0:
List now contains:
0 0 0
```

Insert Elements in `list`

- `iterator insert(iterator position, const T& x);`
  - insert `x` before the `position`
- `void insert(iterator position, size_type n, const T& x);`
  - insert `n` of `x` in front of the `position`

Ref: http://www.cplusplus.com/reference/list/list/insert/

Insert Elements in `list` (cont.)

- `void insert(iterator position, inputIterator first, inputIterator last);`
  - Insert the elements from `first` to `last` in front of `position`

- The same insert function can be used for `deque` and `vector`, but the insert function for `list` is the most efficient one

Using `insert()` in `list`

```
#include <iostream>
#include <list>
#include <vector>
using namespace std;
int main ()
{
    list<int> mylist;
    list<int>::iterator it;

    // set some initial values
    for (int i = 1; i <= 5; i++)
        mylist.push_back(i); // 1 2 3 4 5
    it = mylist.begin();
    ++it;   // "it" now points to number 2
    mylist.insert (it,10);   // 1 10 2 3 4 5
    // "it" still points to number 2
```

Using `insert()` in `list` (cont.)

```
    mylist.insert (it,2,20);
    // 1 10 20 20 2 3 4 5
    --it; // it points now to the second 20
    vector<int> myvector ;
    myvector.push_back(30);
    myvector.push_back(30);
    mylist.insert
        (it,myvector.begin(),myvector.end());
    // 1 10 20 30 30 20 2 3 4 5
    cout << "mylist contains:";
    for ( it = mylist.begin();
          it != mylist.end(); it++)
        cout << " " << *it;
    cout << endl;
}
```

Example of `list`

```
// Fig. 21.17: Fig21_17.cpp
// Standard library list class template test program.
#include <iostream>
#include <list>
#include <algorithm>
#include <iterator>
using namespace std;

template < typename T > void printList( const list< T >
&listRef);

int main()
{
    const int SIZE = 4;
    int array[SIZE] = {2, 6, 4, 8};
    list< int > values;
    list< int > otherValues;
```

Example of `list` (cont.)

```
    values.push_front(1);
    values.push_front(2);
    values.push_back(4);
    values.push_back(3);

    cout << "values contains: ";
    printList(values);

    values.sort();
    cout << "\nvalues after sorting contains: ";
    printList(values);
    // insert elements of array into othervalues
    otherValues.insert(otherValues.begin(), array,
                                            array + SIZE);
    cout << "\nAfter insert, otherValues contains: ";
    printList(otherValues);
```

```
values: 2 1 4 3
```

```
values: 1 2 3 4
```

```
otherValues: 2 6 4 8
```

Example of `list` (cont.)

```
    // remove otherValues elements and insert at end of
values
    values.splice(values.end(), otherValues);
    cout << "\nAfter splice, values contains: ";
    printList(values);

    values.sort();
    cout << "\nAfter sort, otherValues contains: ";
    printList(values);

    otherValues.insert(otherValues.begin(), array,
                                            array + SIZE);
    otherValues.sort();
    cout << "\nAfter insert and sort, otherValues
              contains: ";
    printList(otherValues);
```

```
values: 1 2 3 4 2 6 4 8
otherValues:
```

```
values: 1 2 2 3 4 4 6 8
otherValues:
```

```
otherValues: 2 6 4 8
```

```
otherValues: 2 4 6 8
```

Example of `list` (cont.)

```
    // remove otherValues elements and insert into values
in sorted order
    values.merge(otherValues);
    cout << "\n After merge:\n values contains: ";
    printList(values);
    cout << "\n otherValues contains: ";
    printList(otherValues);

    values.pop_front(); // remove element from front
    values.pop_back();  // remove element from back
    cout << "\nAfter pop_front and pop_back:\n values
contains: ";
    printList(values);

    values.unique();  // remove duplicate elements
    cout << "\nAfter unique, values contains: ";
    printList(values);
```

```
values: 1 2 2 2 3 4 4 4 6 6 8 8
otherValues:
```

```
values: 2 2 2 3 4 4 4 6 6 8
otherValues:
```

```
values: 2 3 4 6 8
otherValues:
```

Example of `list` (cont.)

```
    values.swap(otherValues); // swap elements
    cout << "\n After swap:\n values contains: ";
    printList(values);
    cout << "\n otherValues con
    printList(otherValues);
    // replace contents of values with elements of
otherValues
    values.assign(otherValues.begin(), otherValues.end() );
    cout << "\nAfter assign, values contains: ";
    printList(values);

    values.merge(otherValues);
    cout << "\nAfter merge, values contains: ";
    printList(values);
    values.remove(4);  // remove all 4s
    cout << "\nAfter remove(4), values contains: ";
    printList(values);
    cout << endl;
}
```

```
values:
otherValues: 2 3 4 6 8
```

```
values: 2 3 4 6 8
otherValues: 2 3 4 6 8
```

```
values: 2 2 3 3 4 4 6 6 8 8
```

```
values: 2 2 3 3 6 6 8 8
```

Example of `list` (cont.)

```
template < typename T > void printList( const list< T >
&listRef )
{
    if (listRef.empty())
        cout << "List is empty";
    else
    {
        ostream_iterator<T> output(cout, " ");
        copy(listRef.begin(), listRef.end(), output);
    }
}
```

Example of `list` (cont.)

```
values contains: 2 1 4 3
values after sorting contains: 1 2 3 4
After insert, otherValues contains: 2 6 4 8
After splice, values contains: 1 2 3 4 2 6 4 8
After sort, values contains: 1 2 2 3 4 4 6 8
After insert and sort, otherValues contains: 2 4 6 8
After merge:
   values contains: 1 2 2 2 3 4 4 4 6 6 8 8
   otherValues contains: List is empty
After pop_front and pop_back:
   values contains: 2 2 2 3 4 4 4 6 6 8
After unique, values contains: 2 3 4 6 8
After swap:
   values contains: List is empty
   otherValues contains: 2 3 4 6 8
After assign, values contains: 2 3 4 6 8
After merge, values contains: 2 2 3 3 4 4 6 6 8 8
After remove( 4 ), values contains: 2 2 3 3 6 6 8 8
```

Example of deque

```
// Fig. 21.18: Fig21_18.cpp
// Standard Library class deque test program.
#include <iostream>
#include <deque>
#include <algorithm>
#include <iterator>
using namespace std;

int main()
{
    deque< double > values;
    ostream_iterator< double > output( cout, " ");

    values.push_front(2.2);
    values.push_front(3.5);
    values.push_back(1.1);
```

Ref: http://www.cplusplus.com/reference/deque/deque/

Example of deque (cont.)

```
    cout << "values contains: ";

    for (unsigned int i = 0 ; i < values.size() ; i++)
        cout << values[i] << ' ';

    values.pop_front();
    cout << "\nAfter pop_front, values contains: ";
    copy(values.begin(), values.end(), output);

    values[1] = 5.4;
    cout << "\nAfter values[1] = 5.4, values contains: ";
    copy(values.begin(), values.end(), output);
    cout << endl;
}
```

```
values contains: 3.5 2.2 1.1
After pop_front, values contains: 2.2 1.1
After values[ 1 ] = 5.4, values contains: 2.2 5.4
```

Associative Containers

- Each data item has a key
  - include `set` and `map`
  - support bi-directional iterator
  - key-based fast retrieval of objects from collection => simple database
- Store data objects based on an ordering function
  - easy to look up a object based on a given key
  - objects stored using a tree organization

`set` Container Class

- Simplest container possible
- Stores elements without repetition
- 1st insertion places element in set
  - additional insertions after the first have no effect, so that no element appears more than once
- Capabilities:
  - add elements
  - delete elements
  - ask if element is in set

`set` Container Class (cont.)

- Designed to be efficient
  - stores values in sorted order
  - Can specify order:

```
set<T, ordering> s;
```

    - `ordering` is well-behaved ordering relation that returns `bool`
    - If none specified, use < relational operator

- Note that its insertion function is different from the insert function for sequential containers, such as `list`, `vector`, or `deque`

Ref: http://www.cplusplus.com/reference/set/

Example of `set`

```
#include <iostream>
#include <set>
using namespace std;

int main ()
{
    set<char> s;
    s.insert('A');
    s.insert('D');
    s.insert('D'); //2nd insertion of 'D'
    s.insert('C');
    s.insert('C'); //2nd insertion of 'C'
    s.insert('B');
```

Example of `set` (cont.)

```
    cout << "The set contains:\n";
    set<char>::const_iterator p;
    for ( p = s.begin(); p != s.end(); p++)
        cout << *p << " ";
    cout << endl;
    cout << "Removing C.\n";
    s.erase('C');
    for ( p = s.begin(); p != s.end(); p++)
       cout << *p; << " ";
    cout << endl;
    return 0;
}
```

```
The set contains: A B C D
Removing C
A B D
```

Example of `set`

```
// Fig. 21.20: Fig21_20.cpp
// Standard Library class set test program.
#include <iostream>
#include <set>
#include <algorithm>
#include <iterator>
using namespace std;

typedef set< double, less< double > > DoubleSet;

int main()
{
    const int SIZE = 5;
    double a[SIZE] = {2.1, 4.2, 9.5, 2.1, 3.7};
    DoubleSet doubleSet(a, a + SIZE);
    ostream_iterator< double > output(cout, " ");
```

Ref: http://www.cplusplus.com/reference/functional/less/

Example of `set` (cont.)

```
    cout << "doubleSet contains: ";
    copy(doubleSet.begin(), doubleSet.end(), output);
    // p represents pair containing const_iterator and
bool
    pair< DoubleSet::const_iterator, bool > p;
    // insert 13.8 in doubleSet;
    // p.first represents location of 13.8 in doubleSet
    // p.second represents whether 13.8 was inserted
    p = doubleSet.insert(13.8);
    cout << "\n\n" << *(p.first)
         << (p.second ? " was" : " was not" )
         << " inserted";
    cout << "\ndoubleSet contains: ";
    copy(doubleSet.begin(), doubleSet.end(), output);
```

Ref: http://www.cplusplus.com/reference/set/set/insert/

Ref: http://www.cplusplus.com/reference/utility/pair/

Example of `set` (cont.)

```
    // insert 9.5 in doubleSet;
    p = doubleSet.insert(9.5);
    cout << "\n\n" << *(p.first)
         << (p.second ? " was" : " was not" )
         << " inserted";
    cout << "\ndoubleSet contains: ";
    copy(doubleSet.begin(), doubleSet.end(), output);
    cout << endl;
}
```

```
doubleSet contains: 2.1 3.7 4.2 9.5

13.8 was inserted
doubleSet contains: 2.1 3.7 4.2 9.5 13.8

9.5 was not inserted
doubleSet contains: 2.1 3.7 4.2 9.5 13.8
```

Example of `multiset`

```
// Fig. 21.19: Fig21_19.cpp
// Standard Library class multiset
#include <iostream>
#include <set>
#include <algorithm>
#include <iterator>
using namespace std;

typedef multiset< int, less< int > > Ims;

int main()
{
    const int SIZE = 10;
    int a[SIZE] = {7, 22, 9, 1, 18, 30, 100, 22, 85, 13 };
    Ims intMultiset;
    ostream_iterator< int > output(cout, " ");
```

Ref: http://www.cplusplus.com/reference/set/multiset/

Example of `multiset` (cont.)

```
    cout << "There are currently " << intMultiset.count(15)
         << " values of 15 in the multiset\n";

    intMultiset.insert(15);
    intMultiset.insert(15);
    cout << "After inserts, there are "
         << intMultiset.count(15)
         << " values of 15 in the multiset\n\n";
    // iterator that cannot be used to change elements
    Ims::const_iterator result;

    result = intMultiset.find(15);
    // if iterator not at end
    if (result != intMultiset.end())
        cout << "Found value 15\n";
    result = intMultiset.find(20);
    if (result == intMultiset.end())
        cout << "Did not find value 20\n";
```

Example of `multiset` (cont.)

```
    // insert elements of array a into intMultiset
    intMultiset.insert(a, a + SIZE);
    cout << "\nAfter insert, intMultiset contains:\n";
    copy(intMultiset.begin(), intMultiset.end(), output);
    // determine lower and upper bound of 22
    cout << "\n\nLower bound of 22: "
         << *(intMultiset.lower_bound(22) );
    cout << "\nUpper bound of 22: "
         << *(intMultiset.upper_bound(22) );
    // p represents pair of const_iterator
    pair< Ims::const_iterator, Ims::const_iterator > p;

    p = intMultiset.equal_range(22);

    cout << "\n\nequal_range of 22:"
         << "\n Lower bound: " << *(p.first)
         << "\n Upper bound: " << *(p.second);
    cout << endl;
}
```

Example of `multiset` (cont.)

```
There are currently 0 values of 15 in the multiset
After inserts, therer are 2 values of 15 in the multiset

Found value 15
Did not find value 20

After insert, inMultiset contains:
1 7 9 13 15 15 18 22 22 30 85 100
Lower bound of 22: 22
Upper bound of 22: 30

equal_range of 22:
   Lower bound: 22
   Upper bound: 30
```

Ref: http://www.cplusplus.com/reference/set/multiset/equal_range/

map Container Class

- A function given as set of ordered pairs
  - For each value first, at most one value second in map
- Example map declaration:

```
map<string, int> numberMap;
```

  - `string` values known as keys, the `numberMap` object can associate a unique **`int`** value
- Stores in sorted order, like `set`
  - ordering is on key values only
  - second value can have no ordering impact

Ref: http://www.cplusplus.com/reference/map/map/

Example of map

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

int main ()
{
    map<string, string> planets;
    planets["Mercury"] = "Hot planet";
    planets["Venus"] = "Atmosphere of sulfuric acid";
    planets["Earth"] = "Home";
    planets["Mars"] = "The Red Planet";
    planets["Jupiter"] = "Largest planet in our
                                        solar system";
    planets["Saturn"] = "Has rings";
    planets["Uranus"] = "Tilts on its side";
    planets["Neptune"] = "1500 mile per hour winds";
```

Example of map (cont.)

```
    cout << "Entry for Mercury - "
         << planets["Mercury"] << endl << endl;
    if (planets.find("Mercury") !=
                                planets.end())
        cout << "Mercury is in the map.\n";
    if (planets.find("Ceres") ==
                                planets.end())
        cout <<"Ceres is not in the map.\n";
    cout <<"Iterating through all planets:" << endl;
    map<string, string>::const_iterator iter;
    for ( iter = planets.begin();
          iter != planets.end(); iter++){
        cout << iter->first << " - "
             << iter->second << endl;
    }
    return 0;
}
```

Example of `map` (cont.)

```
Entry for Mercury - Hot planet

Mercury is in the map.
Ceres is not in the map.

Iterating through all planets:
Earth - Home
Jupiter - Largest planet in our solar system
Mars - The Red Planet
Mercury - Hot planet
Neptune - 1500 mile per hour winds
Saturn - Has rings
Uranus - Tilts on its side
Venus - Atmosphere of sulfuric acid
```

Example2 of map

```
// Fig. 21.22: Fig21_22.cpp
// Standard Library class set test program.
#include <iostream>
#include <map>
using namespace std;

typedef map< int, double, less<int> > Mid;

int main()
{
    Mid pairs;
    pairs.insert(Mid::value_type(15, 2.7));
    pairs.insert(Mid::value_type(30, 111.11));
    pairs.insert(Mid::value_type(5, 1010.1));
    pairs.insert(Mid::value_type(10, 22.22));
    pairs.insert(Mid::value_type(25, 33.333));
    pairs.insert(Mid::value_type(5, 77.54)); // dup ignore
    pairs.insert(Mid::value_type(20, 9.345));
```

Example2 of map (cont.)

```
    cout << "pairs contains:\nKey\tValue\n";

    for (Mid::const_iterator iter = pairs.begin();
         iter != pairs.end() ; ++iter)
        cout << iter->first << '\t'
             << iter->second << '\n';
    // use subscripting to change value for key 25
    pairs[ 25 ] = 9999.99;
    // use subscripting to insert value for key 40
    pairs[ 40 ] = 8765.43;

    cout << "\nAfter subscript operations, pairs
             contains:\nKey\tValue\n";
    for (Mid::const_iterator iter = pairs.begin();
         iter != pairs.end() ; ++iter)
        cout << iter->first << '\t'
             << iter->second << '\n';
    cout << endl;
}
```

Example2 of map (cont.)

```
pairs contains:
Key      Value
5        1010.1
10       22.22
15       2.7
20       9.345
25       33.333
30       111.11

After subscript operations, pairs contains:
Key      Value
5        1010.1
10       22.22
15       2.7
20       9.345
25       9999.99
30       111.11
40       8765.43
```

Example of `multimap`

```
// Fig. 21.21: Fig21_21.cpp
// Standard Library class multimap test program.
#include <iostream>
#include <map>
using namespace std;

typedef multimap< int, double, less<int> > Mmid;

int main()
{
    Mmid pairs;

    cout << "There are currently " << pairs.count(15)
         << " pairs with key 15 in the multimap\n";

    pairs.insert(Mmid::value_type(15, 2.7));
    pairs.insert(Mmid::value_type(15, 99.3));
```

Ref: http://www.cplusplus.com/reference/map/multimap/

Example of `multimap` (cont.)

```
    cout << "After inserts, there are "
         << pairs.count(15) << " pairs with key 15\n\n";
    // insert five value_type objects in pairs
    pairs.insert(Mmid::value_type(30, 111.11));
    pairs.insert(Mmid::value_type(10, 22.22));
    pairs.insert(Mmid::value_type(25, 33.333));
    pairs.insert(Mmid::value_type(20, 9.345));
    pairs.insert(Mmid::value_type(5, 77.54));

    cout << "Multimap pairs contains:\nKey\tValue\n";

    for (Mmid::const_iterator iter = pairs.begin();
         iter != pairs.end() ; ++iter)
        cout << iter->first << '\t'
             << iter->second << '\n';

    cout << endl;
}
```

Example of `multimap` (cont.)

```
There are currently 0 pairs with key 15 in the multimap
After inserts, there are 2 pairs with key 15

Multimap pairs contains:
Key     Value
5       77.54
10      22.22
15      2.7
15      99.3
20      9.345
25      33.333
30      111.11
```

Algorithm

- Generic & rich standalone template functions
  - Operate on containers
  - Perform container access through iterators
  - generally unaware of containers
- Category of Algorithms
  - non-modifying sequence algorithm
  - modifying algorithms
  - sorting/searching algorithms
  - generalized numeric algorithms

Non-modifying Sequence Operations

- Apply to sequence containers
  - NOT modify container's contents
  - search for elements in sequences, check for equality and to count sequence elements
- Include:
  - `for_each()`, `count()`, `mismatch()`, `equal()`, `search()`, `find()`, `adjacent_find()`, `find_if()`, `find_end()` and more

Ref: http://www.cplusplus.com/reference/algorithm/

Generic `find` Function

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

int main ()
{
    vector<char> line;
    cout << "Enter a line of text:\n";
    char next;
    cin.get(next);
    while (next != '\n') {
        line.push_back(next);
        cin.get(next);
    }
    vector<char>::const_iterator where;
    where = find(line.begin(),line.end(), 'e');
```

Generic `find` Function (cont.)

```
    vector<char>::const_iterator p;
    cout << "You entered the following
         before you entered your first e:\n";
    for (p = line.begin(); p != where; p++)
        cout << *p ;
    cout << endl;
    cout << "You entered the following after
            that:\n";
    for (p = where; p != line.end(); p++)
        cout << *p ;
    cout << endl;

    cout << "End of demonstrations.\n";
    return 0;
}
```

Generic `find` Function (cont.)

- Ouput - 1

```
Enter a line of text
A line of text.
You entered the following before you entered
your first e:
A lin
You entered the following after that:
e of text.
End of demonstration
```

Generic `find` Function (cont.)

- Ouput - 2

```
Enter a line of text
I will not!
You entered the following before you entered
your first e:
I will not!
You entered the following after that:

End of demonstration
```

Modifying Sequence Algorithms

- STL functions that change container contents
  - Include:
    - `copy()`, `copy_backward()`, `swap()`, `fill()`, `generate()`, `partition()`, `replace()`, `reverse()`, `rotate()`, `swap_ranges()`, `transform()`, `unique()`…
- Adding/removing elements from containers can affect other iterators!
  - `list`, `slist` guarantee no iterator changes
  - `vector`, `deque` make NO such guarantee
- Always watch which iterators are assured to be changed/unchanged

Example of `replace`

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

int main () {
    int myints[] = { 10, 20, 30, 30, 20, 10, 10, 20 };
    vector< int > myvector (myints, myints+8);
    // 10 20 30 30 20 10 10 20
    replace (myvector.begin(), myvector.end(), 20, 99);
    // 10 99 30 30 99 10 10 99
    cout << "myvector contains:";
    for ( vector< int >::iterator it = myvector.begin();
          it != myvector.end(); ++it)
        cout << ' ' << *it;
    cout << '\n';
    return 0;
}
```

Example of swap

```
// Fig. 21.32: Fig21_32.cpp
// Standard Library algorithm iter_swap, swap and
swap_ranges.
#include <iostream>
#include <algorithm>
#include <iterator>
using namespace std;

int main()
{
    const int SIZE = 10;
    int a[SIZE] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    ostream_iterator< int > output(cout," ");

    cout << "Array a contains:\n   ";
    copy(a, a + SIZE, output);
```

Example of swap (cont.)

```
    // swap elements at locations 0 and 1 of array a
    swap(a[0], a[1]);
    cout << "\nArray a after swapping a[0] and a[1]
              using swap:\n   ";
    copy(a, a + SIZE, output);
    // swap iterators to swap elements
    iter_swap( &a[0], &a[1]);
    cout << "\nArray a after swapping a[0] and a[1]
              using iter_swap:\n   ";
    copy(a, a + SIZE, output);
    // swap elements in first five elements of array a
    // with elements in last five elements of array a
    swap_ranges(a, a + 5, a + 5);
    cout << "\nArray a after swapping the first five
        elements\n" << "with the last five elements:\n";
    copy(a, a + SIZE, output);
    cout << endl;
}
```

Example of swap (cont.)

```
Array a contains:
   1 2 3 4 5 6 7 8 9 10
Array a after swapping a[0] and a[1] using swap:
   2 1 3 4 5 6 7 8 9 10
Array a after swapping a[0] and a[1] using iter_swap:
   1 2 3 4 5 6 7 8 9 10
Array a after swapping the first five elements
with the last five elements:
   6 7 8 9 10 1 2 3 4 5
```

Merge Algorithm

- STL generic set operation functions
- All assume containers stored in sorted order
- Containers `set`, `map`, `multiset`, `multimap`
  - DO store in sorted order, so all set functions apply
- others, like `vector`, are not sorted
  - should not use set functions

- Set algorithms include
  - `includes`, `set_union`, `set_intersection`, and `set_difference`

Example of `includes`

```
#include <iostream>
#include <algorithm>
using namespaces std;
bool myfunction (int i, int j) { return i < j; }

int main () {
    int c1[] = {5,10,15,20,25,30,35,40,45,50};
    int c2[] = {40,30,20,10};

    sort (c1, c1 + 10);
    sort (c2, c2 + 4);
    // using default comparison:
    if ( includes(c1, c1 + 10, c2, c2 + 4) )
        cout << "container includes continent!\n";
    // using myfunction as comp:
    if ( includes(c1, c1 + 10, c2, c2 + 4, myfunction) )
        cout << "container includes continent!\n";
    return 0;
}
```

Sorting/Searching Algorithms

- Used to either search or sort container contents
- Versions
  - use < operator for comparison
  - use user-defined comparison *functor* (not covered,; self-study)
- Include:
  - `sort()`, `stable_sort()`, `partial_sort()`, `nth_element()`, `equal_range()`, `binary_search()`, `lower_bound()`…

Example of `sort()`

```
// sort algorithm example
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
bool myfunction (int i, int j)
{
    return (i < j);
}
class comp {
public:
    bool operator()(int x, int y)
    {return x < y;}
};
```

Example of `sort()` (cont.)

```
int main () {
    int myints[] ={32,71,12,45,26,80,53,33};
    vector<int> myvec (myints, myints + 8);
    // 32 71 12 45 26 80 53 33

    // using default comparison (operator <)
    sort(myvec.begin(), myvec.begin() + 4);
    // 12 32 45 71 26 80 53 33
    // using function as comp
    sort(myvec.begin() + 4, myvec.end(), myfunction);
    // 12 32 45 71 26 33 53 80
    // using object as comp
    sort(myvec.begin(), myvec.end(), comp());
    // 12 26 32 33 45 53 71 80
```

Example of `sort()` (cont.)

```
    // print out content
    cout << "myvector contains:";
    for (vector< int >::iterator it = myvec.begin();
         it != myvec.end(); ++it)
        cout << ' ' << *it;
    cout << '\n';
    return 0;
}
```

Example of search

```
// Fig. 21.31: Fig21_31.cpp
// Standard Library search and sort test program.
#include <iostream>
#include <algorithm>
#include <vector>
#include <iterator>
using namespace std;

bool greater10( int value );

int main()
{
    const int SIZE = 10;
    int a[SIZE] = {10, 2, 17, 5, 16, 8, 13, 11, 20, 7};
    vector< int > v(a, a+SIZE);
    ostream_iterator< int > output(cout," ");
```

Example of `search` (cont.)

```
    cout << "Vector v contains: ";
    copy(v.begin(), v.end(), output);
    // locate first occurrence of 16 in v
    vector< int >::iterator location;
    location = find(v.begin(), v.end(), 16);
    if (location != v.end())
        cout << "\n\nFound 16 at location "
             << (location - v.begin());
    else
        cout << "\n\n16 not found";
     // locate first occurrence of 100 in v
    location = find(v.begin(), v.end(), 100);
    if (location != v.end())
        cout << "\nFound 100 at location "
             << (location - v.begin());
    else
        cout << "\n100 not found";
```

Example of `search` (cont.)

```
    // locate first occurrence of value greater than 10
    location = find_if(v.begin(), v.end(), greater10);

    if (location != v.end())
        cout << "\n\nThe first value greater than 10 is "
             << *location << "\nfound at location "
             << (location - v.begin() );
    else
        cout << "\n\nNo values greater than 10 were found";

    sort(v.begin(), v.end());
    cout << "\n\nVector v after sort: ";
    copy(v.begin(), v.end(), output);

    if (binary_search( v.begin(), v.end(), 13))
        cout << "\n\n13 was found in v";
    else
        cout << "\n\n13 was not found in v";
```

Example of `search` (cont.)

```
    if (binary_search( v.begin(), v.end(), 100))
        cout << "\n100 was found in v";
    else
        cout << "\n100 was not found in v";
    cout << endl;
}

bool greater10(int value)
{
    return value > 10;
}
```

Example of `search` (cont.)

```
Vector v contains: 10 2 17 5 16 8 13 11 20 7
Found 16 at location 4
100 not found
The first value greater than 10 is 17
found at location 2
Vector v after sort: 2 5 7 8 10 11 13 16 17 20
13 was found in v
100 was not found in v
```

Summary

- STL is a powerful library
  - includes many generic containers and generic algorithms
- Iterator is 'generalization' of a pointer
  - used to move through elements of container
- Container classes with iterators have:
  - member functions `end()`and `begin()` to assist cycling
- Main kinds of iterators:
  - forward, bi-directional, random-access
- A few containers, generic algorithms provided
  - some algorithms work on specific containers and some on all containers
- Iterators provide common mechanism to access elements in any container

References

- Paul Deitel and Harvey Deitel, “C++ How to Program (late objects version)” Seventh Edition
  - Chapter 21

- W. Savitch, “Absolute C++,” Fourth Edition
  - Chapter 19