Porter Scobey

http://cs.stmarys.ca/~porter/csc/ref/stl/index_containers.html

Also, http://www.java2s.com/Tutorial/Cpp/CatalogCpp.htm

STL Containers

C++ Object Oriented Programming

Pei-yih Ting

NTOU CS

Sequential Containers

- **vector**
  The vector class implements a dynamic array that provides fast insertion at the end, fast random access to its components, and automatically resizes itself when components are inserted or deleted.
- **deque**
  The deque class implements a doubly-ended queue and, in its STL incarnation, has an interface which is very similar to that of the vector class. The major conceptual difference is that a deque provides fast insertion and deletion at *both* ends.
- **list**
  The list class implements a sequential container with fast insertions and deletions anywhere, but without random access to its components values.

std::vector

```
#include <vector> … <iostream> … <iterator>
using namespace std;
vector<int> v1; // empty
if (v1.empty()) cout << "v1 is empty\n";

vector<int> v2(5); // 5 elements, 0 initialized
v2.push_back(3); // 6 elements, 0, 0, 0, 0, 0, 3
for (vector<int>::size_type i=0;
      i<v2.size(); i++)  cout << v2[i] << " ";
cout << endl;

vector<double> v3(4, 3.14); // four elements
ostream_iterator<double> oiter(cout, " ");
copy(v3.begin(), v3.end(), oiter);
cout << endl;

int a[] = {10, 8, 2, 4, 6, 12};
vector<int> v4(a, a + sizeof(a)/sizeof(int));
for (vector<int>::iterator it=v4.begin();
      it!=v4.end(); it++) cout << *it;
cout << endl;
```

```
vector<int> v5(v4);
        // or vector<int> v5 = v4;
v5.erase(v5.begin()); // 8,2,4,6,12
        // inefficient operation
v5.erase(v5.begin()+2,v5.end()-1);
//8,2,12

// there is no sort() or find() function
#include <algorithm>
using namespace std;
vector<int>::iterator iter =
        find(v5.begin(), v5.end(), 2);
if (iter!=v5.end()) // found
   cout << *iter << endl; // 2

sort(v5.begin(), v5.end()); // 2, 8, 12
for (int i=0; i<v5.size(); i++)
   cout << v5[i] << " ";
cout << endl;
```

std::deque

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

int main ()
{
   deque<int> first, second (4,100),
      third(second.begin(),second.end()),
      fourth (third);

   int myints[] = {16,2,77,29};
   deque<int> fifth (myints, myints + 4);

   cout << "The contents of fifth are:";
   for (deque<int>::iterator it = fifth.begin();
         it!=fifth.end(); ++it)  cout << ' ' << *it;
   cout << endl;
```

```
   cout << "Size of fifth = " << fifth.size() << '\n

   fifth.push_front(1); // 1,16,2,77,29
   cout << "Value at the back end = " <<
             fifth.back() << endl;
   fifth.pop_back();

   fifth.push_back(9); // 1,16,2,77,9
   cout << "Value at the front end = " <<
              fifth.front() << endl;
   fifth.pop_front();

   fifth.erase(fifth.begin()); // 2,77,9
   fifth.erase(fifth.begin(), fifth.end()); // empty
   if (fifth.empty()) cout << "fifth is empty!\n";
}
```

```
The contents of fifth are: 16 2 77 29
Size of fifth = 4
Value at the back end = 29
Value at the front end = 1
fifth is empty!
```

std::list

```
#include <list>
#include <iterator>
#include <algorithm>
#include <iostream>
using namespace std;
…
list<int> list1;

list1.push_front(1); // 1
list1.push_front(2); // 2, 1
list1.insert(list1.end(), 3); // 2, 1, 3
// list1.push_back(3);

if (list1.size() == 0) … // bad idea
list1.clear(); // guaranteed to be empty
if (list1.empty()) … // good idea

int iary[] = {1,1,8,7,8,2,3,3};
list1.insert(list1.end(), iary, iary+8);
list1.reverse(); // 3,3,2,8,7,8,1,1
```

```
list1.sort(); // 1,1,2,3,3,7,8,8
list1.unique(); // 1,2,3,7,8

for (list<int>::iterator list1iter = list1.begin();
     list1iter != list1.end(); list1iter++) {
    cout << *list1iter << ' ';
} cout << endl;

int iary2[] = {6, 5, 4};
list<int> list2(iary2, iary2+3); // 6, 5, 4
list2.resize(list2.size() + list1.size());
list<int>::iterator it = list2.end();
copy(list1.begin(), list1.end(), it);
// 6, 5, 4, 1, 2, 3, 7, 8
list2.sort(); // 1, 2, 3, 4, 5, 6, 7, 8
ostream_iterator<int> oiter(cout, ",");
copy(list2.begin(), list2.end(), oiter);

list1.merge(list2); // 1,1,2,2,3,3,4,5,6,7,7,8,8
list1.unique(); // 1,2,3,4,5,6,7,8
```

Container Adaptors

- *container adaptor* is not a "first-class" container, but instead simply "adapts" one of the sequential first-class containers
- **stack**
  It adapts one of the sequential first-class containers, the deque by default. The deque interface is restricted (i.e., much of it is hidden) so that the required LIFO (Last In, First Out) behavior is provided.
- **queue**
  It adapts one of the sequential first-class containers, the deque by default. The deque interface is restricted (i.e., much of it is hidden) so that the required FIFO (First In, First Out) behavior is provided.
- **priority_queue**
  It adapts one of the sequential first-class containers, the vector by default. The vector interface is restricted (i.e., much of it is hidden) so that the required access-via-highest-priority priority-queue-like behavior is provided.

std::stack

```
#include <iostream>
#include <stack>
using std::cout; using std::stack;
int main () {
    stack<int> mystack;
    for (int i=0; i<5; ++i) mystack.push(i);
    cout << "Popping out elements...";
    while (!mystack.empty()) {
        cout << ' ' << mystack.top();
        mystack.pop();
    }
    cout << '\n';
    return 0;
}
```

Popping out elements... 4 3 2 1 0

std::queue

```
#include <iostream>       // std::cin, std::cout
#include <queue>          // std::queue
using namespace std;
int main () {
   queue<int> myqueue;
   for (int i=0; i<10; i++)
      myqueue.push(i); // at the back end
   cout << "myqueue contains: ";
   while (!myqueue.empty()) {
      cout << ' ' << myqueue.front();
      myqueue.pop();
   }
   cout << '\n';
   return 0;
}
```

myqueue contains: 0 1 2 3 4 5 6 7 8 9

std::priority_queue

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

int main() {
   priority_queue<int> pq1;
   pq1.push(19);
   pq1.push(35);
   pq1.push(46);
   pq1.push(11);
   pq1.push(27);

   priority_queue<int> pq2(pq1);

   cout << "\nThe priority queue pq1 contains "
       << pq1.size() << " elements.";
```

```
  while(!pq1.empty()) {
       cout << "Popping: ";
       cout << pq1.top() << "\n";
       pq1.pop();
    }
    cout << endl;

    pq2.push(75);
    pq2.push(5);
    while(!pq2.empty()) {
       cout << "Popping: ";
       cout << pq2.top() << "\n";
       pq2.pop();
    }
}
```

```
Popping: 46
Popping: 35
Popping: 27
Popping: 19
Popping: 11

Popping: 75
Popping: 46
Popping: 35
Popping: 27
Popping: 19
Popping: 11
Popping: 5
```

Associative Containers

C++98 C++11

(key,value)

- map/unordered_map

  ('a',5),('b',15),('c',7),('d',31) / ('b',15),('d',31),('c',7),('a',5)

- multimap/unordered_multimap

  ('a',5),('b',15),('c',7),('c',31) / ('b',15),('c',31),('c',7),('a',5)

- set/unordered_set

  key only

  'a','b','c','d' / 'b','d','c','a'

- multiset/unordered_multiset

  'a','b','c','c' / 'b','c','c','a'

- Associative: Elements are referenced by their *key* and not by their absolute position in the container.

Utilities

- #include <utility>
  using namespace std;
- std::pair<type1, type2>
  - pair<string, double> product1;
    product = make_pair(string("hello"), 0.99);
  - pair<string, double> product2(string("hello"), 0.99);
  - pair<string, double> product3(product2);
  - cout << "Price of " << product1.first << " is " <<
    product1.second << endl;
- std::swap(x, y)
  - int x = 20, y=30;
    swap(x, y);
  - int array1[] = {1, 2, 3, 4}; int array2[] = {5, 6, 7, 8};
    swap(array1, array2);

Function Object

- also called *functors*
- object of a class that defines the “function call operator” operator()
- a *binary functor* that return a *bool* value is called *binary predicate* or *comparitor* or comparison function; a *unary functor* that return a bool value is called a *unary predicate*
- Built-in functors, #include <functional>
  - Arithmetic binary functors
    plus<T> f; // f(arg1, arg2) returns arg1+arg2
    minus<T> f; // f(arg1, arg2) returns arg1-arg2
    multiplies<T> f; // f(arg1, arg2) returns arg1*arg2
    divides<T> f; // f(arg1, arg2) returns arg1/arg2
    modulus<T> f; // f(arg1, arg2) returns arg1%arg2
  - Relational binary functors: equal_to<T>, not_equal_to<T>, greater<T>, greater_equal<T>, less<T>, less_equal<T>
  - Logical binary functors: logical_qand<T>, logical_or<T>
  - Arithmetic unary functor: negate<T>
  - Logical unary functor: logical_not<T>

std::map

- C++98, associative container
  - store elements formed by *key value* and a *mapped value* in the order defined by *key value,*
  - *key values* are generally used to sort and **uniquely** identify the elements
  - *mapped values* store the content associated to this *key* (modifiable)

```
template <class Key,                                   // map::key_type
          class T,                                     // map::mapped_type
          class Compare = less<Key>,                   // map::key_compare
          class Alloc = allocator<pair<const Key,T> > // map::allocator_type >
    class map;
typedef pair<const Key, T> value_type;
```

- The mapped values can be accessed by
  - their corresponding key using the *bracket operator* []
  - direct iteration on subsets based on their order
- implemented as ***binary search trees,*** provide an O(log(n)) access with find()

std::map example 1

- Include file and namespace

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

typedef pair<const Key, T> value_type;

- Define a map and insert values

```
map<string, int> myMap;
pair<map<string, int>::iterator, bool> result;
result=myMap.insert(MapType::value_type("id3",13)); cout<<result.second<<" ";
result=myMap.insert(MapType::value_type("id2",35)); cout<<result.second<<" ";
result=myMap.insert(MapType::value_type("id1",52)); cout<<result.second<<" ";
result=myMap.insert(MapType::value_type("id1",90)); cout<<result.second<<" ";
```

- Access values through an iterator

```
map<string, int>::iterator iter;
for (iter=myMap.begin(); iter!=myMap.end(); iter++)
    cout << "value for " << iter->first << " is " << iter->second << endl;
cout << endl;
```

std::map example 1 (cont'd)

- Access values using operator[]

```
for (i=0; i<myMap.size(); i++) {
    keyid = string("id")+char('1'+i);
    cout << "value for " << keyid << " is " << myMap[keyid] << endl;
}
cout << endl;
```

- Search for a specified key

```
cout << "Please enter a key ID: "; getline(cin, keyid);
while (keyid != "") {
    iter = myMap.find(keyid); // auto iter = myMap.find(keyid);
    if (iter != myMap.end())
        cout << "value for " << iter->first << " is " << iter->second << endl;
    else
        cout << "key not found" << endl;
    cout << "Please enter a key ID: "; getline(cin, keyid);
}
```

std::map example 2

- The messages are always sent after being encoded with a password known to both sides. Having a fixed password is insecure, thus there is a need to change it frequently. However, a mechanism is necessary to send the new password. One of the mathematicians working in the cryptographic team had an idea that was to **send the password hidden within the message itself**. The receiver of the message only had to know the **size of the password** and then search for the password within the received text.
- A password with size N can be found by searching the text for the **most frequent substring with N characters**. After finding the password, all the substrings that coincide with the password are removed from the encoded text. Now, the password can be used to decode the message.
- **Example:** password size **N = 3**; the text message is '**baababacb**'

  The password would be **aba**.

  aba: 2, baa: 1, aab: 1, bab: 1, bac: 1, acb: 1

std::map example 2 (cont'd)

Simpler put, find the most frequently occurring substring of a specified length from a given string.

```
int i, passLen, cipherLen, max;
map<string, int> cont;
map<string, int>::iterator iter;
string cipher, answer;

cin >> passLen;
while (!cin.eof()) {
    cin >> cipher;
    cipherLen = cipher.size();
    cont.clear();

    for (i=0; i+passLen<=cipherLen; i++)
        cont[cipher.substr(i, passLen)]++;

    for (max=0,iter=cont.begin();
                    iter!=cont.end(); iter++)
        if (iter->second > max) {
            max = iter->second;
            answer = iter->first;
        }

    cout << answer << ":" << max  << endl;
    cin >> passLen;
}
```

```
1 thequickbrownfoxjumpsoverthelazydog
4 testingthecodetofindtheerrortestandtestaga
```

```
o:4
test:3
```

std::multimap

- C++98
- A std::multimap (multiple-key map) is equal to a std::map, but your keys are **not unique** any more. Therefore, for each key you can find a range of items instead of just one unique item.
- multimap::insert can insert any number of items with same key.

```
template < class Key,                                      // key_type
           class T,                                        // mapped_type
           class Compare = less<Key>,                      // key_compare
           class Alloc = allocator<pair<const Key,T> >     // allocator_type
         > class multimap;
```

- implemented as ***binary search trees***, provide an O(log(n)) access with equal_range()

std::multimap example

```
typedef multimap<char,int> MMCI;
typedef MMCI::iterator MMCI_ptr;
MMCI mymm;
mymm.insert(MMCI::value_type('a',10)); mymm.insert(MMCI::value_type('b',30));
mymm.insert(MMCI::value_type('d',70)); mymm.insert(MMCI::value_type('b',20));
mymm.insert(MMCI::value_type('c',60)); mymm.insert(MMCI::value_type('b',40));
mymm.insert(MMCI::value_type('c',50));
```

```
for (MMCI_ptr it=mymm.begin(); it!=mymm.end(); it++)
   cout << "(" << it->first << "," << it->second << ") ";
cout << endl;
```

(a,10) (b,30) (b,20) (b,40) (c,60) (c,50) (d,70)

```
cout << "mymm contains:\n";
for (ch='a'; ch<='d'; ch++) {
   pair<MMCI_ptr, MMCI_ptr> ret = mymm.equal_range(ch);
   cout << ch << " =>";
   for (iter=ret.first; iter!=ret.second; ++iter)
      cout << ' ' << iter->second;
   cout << '\n';
}
```

mymm contains:
a => 10
b => 30 20 40
c => 60 50
d => 70

std::multimap example (cont'd)

```
for (MMCI_ptr it1=mymm.begin(), it2=it1, end=mymm.end(); it1!=end; it1=it2) {
   cout << it1->first << " =>";
   for ( ; it2->first == it1->first; it2++)
      cout << ' ' << it2->second;
   cout << endl;
}
cout << "# remaining elements: " << mymm.size() << endl;
for (ch='a'; ch<='e'; ch++) {
   MMCI_ptr iter = mymm.find(ch);
   if (iter != mymm.end()) {
      cout << "key: " << iter->first << " value: " << iter->second << " erased"<< endl;
      mymm.erase(iter);
   }
   else cout << "key not found" << endl;
}
cout << "# remaining elements: " << mymm.size() << endl;
cout << "# elements with key 'b' erased: " << mymm.erase('b') << endl;
cout << "# remaining elements: " << mymm.size() << endl;
```

mymm contains:
a => 10
b => 30 20 40
c => 60 50
d => 70

```
# remaining elements: 7
key: a value: 10 erased
key: b value: 30 erased
key: c value: 60 erased
key: d value: 70 erased
key not found
# remaining elements: 3
# elements with key 'b' erased: 2
# remaining elements: 1
```

std::unordered_map

- C++11, associative container
  - store elements formed by *key value* and a *mapped value*
  - *key values* are used to **uniquely** identify the elements
  - *mapped values* store the content associated to this *key* (modifiable)

```
template <class Key,                                        // key_type
          class T,                                          // mapped_type
          class Hash = hash<Key>,                           // hasher
          class Pred = equal_to<Key>,                       // key_equal
          class Alloc = allocator<pair<const Key,T> > // allocator_type >
    class unordered_map;
typedef pair<const Key, T> value_type;
```

- The mapped values can be accessed by
  - their corresponding key using the *bracket operator* [ ]
  - forward iterators
- Not sorted in any particular order, but organized into ***buckets*** (i.e. hash table) depending on their hash values to allow for fast access directly by their *key values*

std::unordered_multimap

- C++11
- Unordered multimaps are associative containers that store elements formed by the combination of a *key value* and a *mapped value*, much like unordered_map containers, but allowing different elements to have the same keys.

```
template < class Key,                                    // unordered_multimap::key_type
           class T,                                      // mapped_type
           class Hash = hash<Key>,                       // hasher
           class Pred = equal_to<Key>,                   // key_equal
           class Alloc = allocator< pair<const Key,T> // allocator_type
         > class unordered_multimap;
```

- Not sorted in any particular order, but organized into ***buckets*** depending on their hash values to allow for fast access directly by their *key values*
- Interestingly, equal_range() still works

std::set

- C++98
- The std::set is like an std::map, but it is not storing a key associated to a value. It stores **only the key type**, and assures you that it is **unique** within the set.

```
template < class T,                          // set::key_type/value_type
           class Compare = less<T>,  // set::key_compare/value_compare
           class Alloc = allocator<T>  // set::allocator_type
         > class set;
```

- implemented as ***binary search trees,*** provide an O(log(n)) access with find()

Specify the element order

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

class Student {
    friend class Comp;
public:
    Student(int num1, string name1)
        :num(num1), name(name1) {}
    void print(ostream &os) const
        {os << num << "\t" << name << endl;}
private:
    int num; string name;
};

class Comp {
public:
    bool operator() (Student s1, Student s2){
        if (s1.num < s2.num)
            return true;
        else
            return false;
    }
};
```

```
# of students 3
5       Ziale
10      Anwar
17      Tauman
17      Tauman
```

```
int main() {
    set<Student, Comp> myStudents;
    Student a1(10, "Anwar"), a2(5, "Ziale"),
                a3(17, "Tauman");
    myStudents.insert(a1);
    myStudents.insert(a3);
    myStudents.insert(a2);
    myStudents.insert(a1); // would merge

    cout << "#  of students " <<
                myStudents.size() << endl;
    set<Student, Comp>::iterator iter;
    for (iter=myStudents.begin();
              iter != myStudents.end(); iter++)
        iter->print(cout);

    iter = myStudents.find(Student(17, ""));
    if (iter != myStudents.end())
        iter->print(cout);
    else
        cout << "Not found!" << endl;
    return 0;
}
```

Map from std::set

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

class Person {
public:
    Person(string name, int age){
        this->name = name;
        this->age = age;
    }
    string getName() const { return name; }
    int getAge() const       { return age; }
    bool operator<(const Person& other) const {
        return name < other.name;
    }
private:
    string name;
    int age;
};

int main() {
    Person a[] = { Person("William", 23),
                   Person("John", 20),
                   Person("Alice", 18),
                   Person("Peter", 24),
                   Person("Bob", 17) };
    set<Person> s(a, a+5);
    set<Person>::iterator p = s.begin();
    while (p != s.end()) {
        cout << p->getName() << " is " <<
                p->getAge() << " years old.\n";
        ++p;
    }
   p = s.find(Person("Alice", 99));
   if (p != s.end())
       cout << p->getName() << " is " <<
         p->getAge() << " years old.\n";
   else
       cout << "Not found!" << endl;
    return 0;
}
```

std::multiset

- C++98
- Multisets are associative containers that store elements following a specific order, and where multiple elements can have equivalent values.
- In a multiset, the value of an element also identifies it (the value is itself the *key*, of type T). The value of the elements in a multiset cannot be modified once in the container (the elements are always const), but they can be inserted or removed from the container.

```
template < class T,                        // multiset::key_type/value_type
           class Compare = less<T>,        // multiset::key_compare/value_compare
           class Alloc = allocator<T> >    // multiset::allocator_type
           > class multiset;
```

- implemented as ***binary search trees,*** provide an O(log(n)) access equal_range()

std::unordered_set

- C++11
- Unordered sets are containers that store ***unique*** elements in ***no particular order***, and which allow for ***fast retrieval*** of individual elements based on their key value.

```
template < class Key,                          // key_type/value_type
           class Hash = hash<Key>,             // hasher
           class Pred = equal_to<Key>,         // key_equal
           class Alloc = allocator<Key>        // allocator_type
         > class unordered_set;
```

- Not sorted in any particular order, but organized into ***buckets*** depending on their hash values to allow for fast access directly by their *key values*

std::unordered_multiset

- C++11
- Unordered multisets are containers that store elements in no particular order, allowing fast retrieval of individual elements based on their value, much like unordered_set containers, but allowing different elements to have equivalent values.

```
template < class Key,                            // key_type/value_type
           class Hash = hash<Key>,               // hasher
           class Pred = equal_to<Key>,           // key_equal
           class Alloc = allocator<Key>          // allocator_type
         > class unordered_multiset;
```

- Not sorted in any particular order, but organized into ***buckets*** depending on their hash values to allow for fast access directly by their *key values*
- Interestingly, equal_range() still works

Container Properties

	Associative	Ordered	Unique keys	Allocator -aware
Set	✓	✓	✓	✓
Multiset	✓	✓	✗	✓
Unordered_set	✓	✗	✓	✓
Unordered_Multiset	✓	✗	✗	✓
Map	✓	✓	✓	✓
Multimap	✓	✓	✗	✓
Unordered_map	✓	✗	✓	✓
Unordered_Multimap	✓	✗	✗	✓

Associative Container Member Functions

member function	map	multimap	set	multiset
operator [ ]	✓	✗	✗	✗
=, ==, !=, <, <=, >, >=	✓	✓	✓	✓
empty()	✓	✓	✓	✓
size()	✓	✓	✓	✓
max_size()	✓	✓	✓	✓
swap(otherLikeContainer)	✓	✓	✓	✓
begin(), end()	✓	✓	✓	✓
rbegin(), rend()	✓	✓	✓	✓
insert(val) insert(iter, val) insert(iter, start, end)	✓	✓	✓	✓

Member Functions (cont'd)

member function	map	multimap	set	multiset
erase(iter) erase(start, end) erase(key)	✓	✓	✓	✓
clear()	✓	✓	✓	✓
key_comp()	✓	✓	✓	✓
value_comp()	✓	✓	✓	✓
count(key)	✓	✓	✓	✓
equal_range(key)	✓*	✓	✓*	✓
lower_bound(key)	✓	✓	✓	✓
upper_bound(key)	✓	✓	✓	✓
find(key)	✓	✓	✓	✓
get_allocator()	✓	✓	✓	✓

*the range returned will contain a single element at most.