STL Containers and Container Adaptors: Deque, Set, Map Stack, Queue, Priority_Queue

Revisit STL containers

- Sequence containers
  - C++ Vectors //allow random access, insert data at the end with push_back() (unless using insert() with iterator()
  - C++ Lists //doubly linked, no random access
  - C++ Double-Ended Queues //doubly linked, allow random access
- Associative Containers
  - C++ Bitsets
  - C++ Maps
  - C++ Multimaps
  - C++ Sets
  - C++ Multisets
- Container Adapters
  - C++ Stacks
  - C++ Queues
  - C++ Priority Queues

deque -- *double-ended queue* "deck"

- Sequence container
- Can expand in either direction
- Similar interface as vector, but allow insertion/deletion at the beginning.
- More complex internally. Not a single array, can be scattered in different chunks of storage. The container keeps the necessary information to provide direct access.
- For frequent insertion or removal of elements at positions other than beginning or the end, suggest list.

deque

- Constructors:
  - deque<data_type> deque_name; //empty
  - deque<data_type> deque_name(other_deque);
  - deque<data_type> deque_name(initial_size);
- Methods and operations:
  - push_front(value);
  - pop_front();
  - push_back(value);
  - pop_back();
  - front() //return a reference to the first element in the deque
  - back() //return a reference to the last element in the deque
  - at(index)
  - [index] //deque_name[i]

Example of push_front()

```
#include <iostream>
#include <deque>
int main() {
std::deque <int> mydeque(2, 100);
// std::deque <int>::iterator it; // but use auto
mydeque.push_front(200);
mydeque.push_front(300);
std::cout << "mydeque contains: ";
for(auto it = mydeque.begin(); it != mydeque.end(); ++it)
std::cout << ' ' << *it;
std::cout << '\n';
return 0; }
// mydeque contains: 300 200 100 100
```

Deque random access

```
#include <iostream>
#include <deque>
int main() {
using namespace std;
deque <int> deq;
for(int nCount = 0; nCount < 3; nCount++) {
   deq.push_back(nCount);
   deq.push_front(10 - nCount);
 }
for(int nIndex = 0; nIndex < deq.size(); nIndex++)
   cout << deq[nIndex] << " ";
return 0; }
// Output: 8 9 10 0 1 2
```

Associative Containers

- A set is a container that stores unique keys.
- A multiset allows multiple elements with the same key.
- A map stores key/value pairs. The key is used for sorting and indexing the data, and must be unique.
- A multimap allows multiple elements with the same key.

- Newer C++ standard has unordered_set and unordered_map, which allow faster look up but do no keep a sorted order.

Set

- Sets are containers that store **unique** elements following a specific order.
  https://cplusplus.com/reference/set/set/?kw=set

- Set and multiset are typically implemented using “binary search tree”. We will cover this later in the class.
- Unordered_set are typically implemented using “hash table”, we will also cover this later in the class.
- They are not required to be implemented in this manner, but it tends to match the requirement the best.

Example:

```
int myints[] = {75,23,65,42,13};
std::set<int> myset (myints,myints+5);

for (std::set<int>::iterator it=myset.begin(); it!=myset.end(); ++it)
        std::cout << ' ' << *it;
```

Output: 13 23 42 65 75 (sorted)

- Cannot change set value – elements are always const. Because otherwise may compromise the correct ordering.
- If you want to “change”, erase it, then insert a new one.

- The iterators provided by associative containers are bidirectional iterators.

Example of insert in a set

```
#include <iostream>
#include <set>
#include <vector>
using namespace std ;
int main() {
vector <int> v;
set <int> s;
v.push_back(2);
v.push_back(10);
for(int i = 7; i <= 9; i++)
   s.insert(i);
s.insert (v.begin (), v.end());

for(auto it = s.begin(); it != s.end(); ++it)
cout << *it << " ";
return 0; }
// Output: 2 7 8 9 10     <- sorted!
```

- mySet.erase(val); -- delete all elements with the value val
- mySet.erase(iteratorPos); -- delete the element at position pointed to by iteratorPos
- mySet.erase(iteratorBegin, iteratorEnd); -- delete the elments in the range of [begin, end).

Example of erase on a set

```
#include <iostream>
#include <set>
using namespace std;
int main() {
  set <int> s;
  for(int i = 1; i <= 9; i++)
     s.insert(i);
  s.erase(5);
  auto it = s.begin();
  ++ it;
  it = s.erase(it);
  s.erase(it, s.find(7));  //erase a sequence!
  for (it = s.begin(); it != s.end(); ++it)
    cout << *it << " ";
  return 0; }
// Output: 1 7 8 9
```

set algorithms in STL (not member functions of the set container)

- includes(begin, end, begin2, end2); //return true if first sequence is contained in the second, false otherwise
- set_union(begin1, end1, begin2, end2, outputIterator); //union, can be overloaded to add a comparator
- set_intersection(begin1, end1, begin2, end2, outputiterator) //elements that are present in both sets
- set_difference(begin1, end1, begin2, end2, outputiterator)

- Can be applied to other sequence containers, as long as sorted.

set_intersection example

```
#include <iostream> // std::cout
#include <algorithm> // std::set_intersection, std::sort
#include <vector> // std::vector
int main ()
{
   int first[] = {5,10,15,20,25};
    int second[] = {50,40,30,20,10};
    std::vector<int> v(10); // 0 0 0 0 0 0 0 0 0 0
    std::vector<int>::iterator it;
    std::sort (first,first+5); // 5 10 15 20 25
    std::sort (second,second+5); // 10 20 30 40 50
    it=std::set_intersection (first, first+5, second, second+5, v.begin()); // 10 20 0 0 0 0 0 0 0 0
    v.resize(it-v.begin()); // 10 20  (shrink to 2, it points to the end of intersection)
   std::cout << "The intersection has " << (v.size()) << " elements:\n";
    for (it=v.begin(); it!=v.end(); ++it)
          std::cout << ' ' << *it; std::cout << '\n';
    return 0;
}
```

```
(THIS EXAMPLE IS OPTIONAL)
const int N = 6;
const char* a[N] = {"isomer", "ephemeral", "prosaic", "nugatory", "artichoke", "serif"};
const char* b[N] = {"flat", "this", "artichoke", "frigate", "prosaic", "isomer"};
set<const char*, ltstr> A(a, a + N); //to be simplified – ltstr is a predefined comparator
set<const char*, ltstr> B(b, b + N);
set<const char*, ltstr> C;

cout << "Set A: ";
copy(A.begin(), A.end(), ostream_iterator<const char*>(cout, " "));
cout << endl;
cout << "Set B: ";
copy(B.begin(), B.end(), ostream_iterator<const char*>(cout, " "));
cout << endl;

cout << "Union: ";
set_union(A.begin(), A.end(), B.begin(), B.end(), ostream_iterator<const char*>(cout, " "), ltstr());
cout << endl;

cout << "Intersection: ";
set_intersection(A.begin(), A.end(), B.begin(), B.end(), ostream_iterator<const char*>(cout, " "), ltstr());
cout << endl;

set_difference(A.begin(), A.end(), B.begin(), B.end(), inserter(C, C.begin()), ltstr());
cout << "Set C (difference of A and B): ";
copy(C.begin(), C.end(), ostream_iterator<const char*>(cout, " "));
cout << endl;
```

STL map

Maps

- C++ maps are sorted associative containers that contain unique key/value pairs.

- The container unordered_map does not do sorting. Faster on direct access. Less efficient for iteration.

Map Constructors & Destructors

- #include <map>
- map();
- map(const map& m);
- map(iterator start, iterator end);
- map(iterator start, iterator end, const key_compare& cmp);
- map(const key_compare& cmp);
- ~map();

Map operators

- TYPE & operator[](const key_type& key);
- map operator=(const map& c2);
- bool operator==(const map& c1, const map& c2);
- bool operator!=(const map& c1, const map& c2);
- bool operator<(const map& c1, const map& c2);
- bool operator>(const map& c1, const map& c2);
- bool operator<=(const map& c1, const map& c2);
- bool operator>=(const map& c1, const map& c2);

map methods

- begin — returns an iterator to the beginning of the map
- clear — removes all elements from the map
- count — returns the number of elements matching a certain key
- empty — true if the map has no elements
- end — returns an iterator just past the last element of a map
- erase — removes elements from a map
- find — returns an iterator to specific elements
- insert — insert items into a map
- key_comp — returns the function that compares keys
- equal_range — returns iterators to the first and just past the last elements matching a specific key

This slide is optional

- lower_bound returns an iterator to the first element greater than or equal to a certain value
- max_size returns the maximum number of elements that the map can hold
- rbegin returns a reverse_iterator to the end of the map
- rend returns a reverse_iterator to the beginning of the map
- size returns the number of items in the map
- upper_bound returns an iterator to the first element greater than a certain value
- value_comp returns the function that compares values

the [ ] operator for a map

- Each element is composed of a *key* and a *mapped* value

- Maps are also unique among associative containers in that they implement the *direct access operator (operator[])* which allows for direct access of the mapped value, but know that the [] is not referring to a specific position (as in vector).

- If *k* does not match the key of any element in the container, the function[] inserts a new element with that key and returns a reference to its mapped value. Notice that this always increases the container size by one.

Example:

```
map<string, int> ages;
ages["Homer"] = 38;
ages["Marge"] = 37;
ages["Lisa"] = 8;
ages["Maggie"] = 1;
ages["Bart"] = 11;

cout << "Bart is " << ages["Bart"] << " years old" << endl;
```

Iterator for map and the *pair* class

- Iterators to elements of map containers access to both the *key* and the *mapped value*. For this, the class defines what is called its value_type, which is a pair class
- pair: its first value corresponding to the const version of the *key* type (template parameter *Key*) and its second value corresponding to the *mapped value* (template parameter *T*):
  - typedef pair<const Key, T> value_type;
- A pair is an object that contains two values.
- It is a struct type. All members are public: first, second

Example:

pair<type1, type2> nameofPair;

pair<V1, V2> make_pair(x, y); → convenient function to construct a pair with first element is x, second element is y.

Iterator for map

- Iterators of a map container point to elements of this *value_type*.
- For an iterator called *it* that points to an element of a map, its key and mapped value can be accessed respectively with:
  - map<Key,T>::iterator it;
  - (*it).first; // the key value (of type Key)
  - (*it).second; // the mapped value (of type T)
  - (*it); // the "element value" (of type pair<const Key,T>)
- Other direct access operator, such as -> or [] can be used, for example:
  - it->first; // same as (*it).first (the key value)
  - it->second; // same as (*it).second (the mapped value)

Example of Map

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

int main() {
  map <int, string> mymap;
  mymap.insert(make_pair(4, "apple"));     //or  mymap[4] = “apple”;
  mymap.insert(make_pair(1, "orange"));
  mymap.insert(make_pair(3, "grapes"));
  mymap.insert(make_pair(2, "peach"));
  auto it = mymap.begin();
  while(it != mymap.end()) { // *it will be a std::pair
    cout << it->first << "=" << it->second << " ";
    it++; }
  return 0;}
```

- `// Output: 1=orange 2=peach 3=grapes 4=apple // pairs, sorted by key in it->first`

Example 2

```
#include <map>
#include <iostream>
using namespace std;
void print_map(const map<string, int>& myMap)
{
 for(auto mapIt = myMap.begin(); mapIt != myMap.end(); ++mapIt)
    cout << mapIt->first << " : " << mapIt->second << endl;
}
int main()
{
  map<string, int> ages;
  ages["John"] =38;
  ages["Jane"] = 3;
  ages["Amy"] = 4;
  print_map(ages);

  ages["John"] ++;
  print_map(ages);
  return 0;
}
```

Revisit the example of phone book of list

```
struct Entry { string name; int number; };
const int N = 12;
list<Entry> phone_book;
void print_entry(const string&);
int main()
{
   for (int i = 0; i < N; i++) {
      Entry e;
      cin >> e.name >> e.number;
      phone_book.push_back(e);
   }
   cout << "Print the entire phone book" << endl;
   for(auto i = phone_book.begin(); i != phone_book.end(); i++)
         cout << i->name << ' ' << i->number << endl;
   cout << "Add jack 815111 to the phone book list. " << endl;
   Entry e = { "jack", 815111 };
   phone_book.push_back(e);
   cout << "Print the entry for jack: " << endl;
   print_entry("jack");
   return 0;
}
```

```
//If using a map, no need to loop
void print_entry(const string& s)
{
   for (auto i = phone_book.begin(); i != phone_book.end(); i++) {
      const Entry& e = *i;
      if (s == e.name) {
         cout << e.name << ' ' << e.number << endl;
}
```

How would you rewrite it using a map?

Solution

```
map<string, int> phonebook;
string name;
int num;
 for (int i = 0; i < N; i++) {
       cin >> name >> number;
       phone_book[name] = num;
   }
  for(auto it = phonebook.begin(); it != phonebook.end(); ++it)
    cout << it->first << " : " << it->second << endl;
  //add jack
 phonebook[”jack”] = 815111;
  //print the entry for jack
  cout << phonebook[“jack”];
```

Size, capacity, max_size

```
#include <iostream>
#include <vector>
int main() {
std::vector<int> myvector;
for(int i = 0; i<100; i++) myvector.push_back(i);
std::cout << "size: " << (int) myvector.size() << '\n';
std::cout << "capacity: " << (int) myvector.capacity() << '\n';
std::cout << "max_size: " << (int) myvector.max_size() << '\n';
return 0; }
// size: 100
// capacity: 141  (max# that can be inserted without reallocation)
// max_size: 1073741823  (max possible size)
```

Container adaptors

- Encapsulated object of a specific container class as the underlying container, provide a *specific* set of member functions to access its elements.
- They are not first-class containers.
- They do not support iterators.

stack

- Container adaptor. Uses an underlying container. By default, deque is used. The containers vector and list may also be used to instantiate a stack.

  template<class T, class Container = deque<T>> class stack;

- LIFO -- Last In First Out.
- Methods:
  - bool empty() const;
  - size_type size() const;
  - value_type &top(); //return the element. Do not remove the element.
  - void push(const value_type & val)
  - void pop() //remove top element. Do not return the element.

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

Student exercise

- *Write a C++ program that initializes the content of a stack to a sequence of numbers (from 0 to 100) and then pops the elements one by one until it is empty, and print out the sum of these numbers.*

An implementation of a stack using a vector or (linked) list or a deque

- The underlying container needs to support
  - empty, size, back, push_back and pop_back

- Push() and pop() can be implemented using:
  - push --- push_back()
  - pop --- pop_back() (return type void, does not return the popped value.)

queue

- Container adaptor. Uses an underlying container. By default, deque is used. list may also be used to instantiate a stack.

  template<class T, class Container = deque<T>> class queue;

- FIFO
- Methods:
  - bool empty()
  - size_type size()
  - value_type &front()
  - value_type &back()
  - void push(const value_type& val) //insert an element at the back of the queue
  - void pop() //remove the element at the front of the queue. Implemented using pop_front(). The element removed is the "oldest" element in the queue (the first element that was pushed in).

Example:

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

Exercise: Add odd number to q1, even number to q2

- *queue <double> q1, q2;*
- *int value;*
- *cout << "\n Enter an integer > 0 or 0 to quit:" ;*
- *cin >> value;*
- *while(value != 0) {*
- *// what goes here?*
- *cout << "\n Enter an integer > 0 or 0 to quit:" ;*
- *cin >> value; }*

Answer

```
queue <double> q1, q2;
int value;
cout << "\n Enter an integer > 0 or 0 to quit:" ;
cin >> value;
while(value != 0) {
  if(value % 2 == 0) q1.push(value);
else q2.push(value);
cout << "\n Enter an integer > 0 or 0 to quit:" ;
cin >> value; }
```

piority_queue

- Container adaptor. The first element is always the highest priority.
- Default underlying container is vector. Can also use deque.
- Useful for situations need to consider priority.
- Will be discussed in heap later in the course.
- Methods:
  - empty()
  - size()
  - top()
  - push(): insert an element at the right location: push_back, then reorder using heapsort
  - pop(); implemented using pop_back()

Example

```
#include <iostream>
#include <queue>
using namespace std;
int main() {
  priority_queue <double> mypq;
  mypq.push (2.5);
  mypq.push (9.2);
  mypq.push (1.3);
  mypq.push (5.5);
  while(!mypq.empty()) {
    cout << mypq.top() << ", ";
    mypq.pop(); }
return 0; }
// Output: 9.2, 5.5, 2.5, 1.3,
```