CS2810: Introduction to STL

Rupesh Nasre

IIT Madras

09/01/2017

Introduction

- STL is a generic library of Data Structures and Algorithms and stands for **S**tandard **T**emplate **L**ibrary
- Mainly composed of 2 types of libraries-
  - Containers/Data Structures
  - Algorithms

Containers

vector

- Vectors store their elements in contiguous memory locations like arrays.
- Unlike arrays, can grow in size dynamically.

Usage:

```
#include <vector>
```

Initialize as:

```
vector<typename> vect1;
vector<typename> vect2{e1,e2,e3,...};
vector<typename> vect3(size);
vector<typename> vect4(size,val);
```

Member Functions: Some important member functions are:

- **assign:**

```
vect1.assign(initial_size,initial_val);
vect2.assign(begin_iterator, end_iterator);
```

- **at** and **operator[]:**

```
vect.at(index); vect[index];
```

- **begin** and **end**:

```
vector<int>::iterator start= vect.begin();
vector<int>::iterator end = vect.end();
```

- **capacity** and **size**:

```
vect.capacity(); vect.size();
```

- **clear:** Removes all vector elements making its size 0 whereas capacity doesn't change.

  ```
  vect.clear();
  ```

- **data:**

  ```
  int* ptr= vect.data();
  ```

- **emplace**:

  ```
  vect.emplace(iterator,val);
  ```

- **push_back:**

  ```
  vect.push_back(val);
  ```

- **empty:** Returns **true** if the vector is empty i.e. size is 0.

  ```
  vect.empty();
  ```

- **erase:** Returns an iterator pointing to the element following the last element being erased.

```
vector<type>::iterator it1= vect.erase(some_iterator),
it2= vect.erase(start_iterator, end_iterator);
```

- **insert:**

```
vect.insert(curr_iter,val);
vect.insert(curr_iter,n,val);
vect.insert(curr_iter, src_start_iter,src_end_iter);
```

- **operator=**:

```
ex: vector<int> vect1=2,3,4;
int arr[]={10,34,90};
vector<int> vect2={arr,arr+2};
vect2= vect3;
vect2={vect3};
```

- **push_back** and **pop_back**:

  ```
  vect.push_back(); vect.pop_back();
  ```

- **resize**:

  ```
  vect.resize(n); vect.resize(n,val);
  ```

- **swap**:

  ```
  vect1.swap(vect2);
  ```

list

- Given the iterator, lists support constant time insertion and deletion of elements
- Direct access to any random index is inefficient as compared to vectors, arrays, deques etc..
- List elements are not stored in contiguous memory locations
- Require extra memory space to keep the linking information

usage:

```
#include <list>
```

Initialization:

```
list<typename> lst;
list<typename> lst(size,val);
list<typename> lst(arr,arr+n);
list<typename> lst(src_start_iter,src_end_iter);
```

Member Functions:

- *assign*, *begin*, *end*, *clear*, *empty*, *erase*, *insert*, *operator=*, *push_back*, *pop_back*, *resize*, *swap* can be used the same way as in vectors.
- **back** and **front**: Returns reference to the last and first elements respectively

  ```
  lst.back(); lst.front();
  ```

- **push_front**: Inserts a new element to the beginning of the list.

  ```
  lst.push_front(val);
  ```

- **merge**: Merges two lists depending on the comparison function while destroying the second list.

  ```
  lst1.merge(lst2);
  lst1.merge(lst2,comparison_function);
  ```

- **pop_front:**

  ```
  lst.pop_front();
  ```

- **remove**: Removes all the elements from the list which compare equal to the parameter.

  ```
  lst.remove(val);
  ```

- **remove_if**: Removes all those elements from the list for which function passed to it as argument returns **true**

  ```
  lst.remove_if(func());
  ```

- **reverse**:

  ```
  lst.reverse();
  ```

- **sort**:

  ```
  lst.sort(); lst.sort(comp());
  ```

- **splice**: Transfers some/all elements from one list to another while removing transferred elements from the first.

  ```
  lst1.splice(lst1_iter_pos,lst2);
  lst1.splice(lst1_iter_pos, lst2, lst2_iter_pos);
  lst1.splice(lst1_iter_pos,lst2,lst2_iter_pos1,
                        lst2_iter_pos2);
  ```

stack

- Stacks are designed to operate in LIFO fashion where elements are inserted and removed from one end.

Usage:

```
#include <stack>
```

Initialization:

```
stack<typename> var_name;
```

Member Functions:

- **push**:

  ```
  st.push(val);
  ```

- **pop**,**size**, **empty** and **top**:

  ```
  st.pop(); st.size(); st.empty(); st.top();
  ```

- **swap**:

  ```
  st1.swap(st2);
  ```

queue

- Elements are inserted from one end and extracted from the other.
- Queues operate in FIFO fashion.

Usage:

```
#include <queue>
```

Initialization:

```
queue<typename> var_name;
```

- **back**: Returns a reference to the last element in the queue.

```
que.back();
```

- **empty**:

```
que.empty();
```

- **front**:

```
que.front();
```

- **pop**:

```
que.pop();
```

- **push**:

```
que.push();
```

- **size**:

```
que.size();
```

- **swap**:

```
que1.swap(que2);
```

priority_queue

- Elements are stored based on certain priority and only the element with the highest priority can be retrieved at any time.

Usage:

```
#include <queue>
```

Initialization:

```
priority_queue<typename> pq;
priority_queue<typename, vector<typename>,
                         comparator > pq;
```

- **empty**: Returns true if priority_queue is empty

  ```
  pq.empty();
  ```
- **pop**: Removes the highest-priority element from the queue

  ```
  pq.pop();
  ```
- **push**:

  ```
  pq.push();
  ```
- **size**:

  ```
  pq.size();
  ```
- **swap**: Swaps contents of the two underlying containers

  ```
  pq1.swap(pq2);
  ```
- **top**: Returns the elements with the highest priority

  ```
  pq.top();
  ```

deque

- Stands for **D**ouble-**e**nded-**que**ues, which allow insertion and deletion from both ends.
- Both vectors and dequeues provide the similar interface except vector elements are stored in contiguous memory locations and need to be reallocated occasionally whereas dequeue elements can be stored in scattered memory locations.

Usage:

```
#include <deque>
```

Initialization:

```
typename arr[]={e1,e2,e3,...};
deque<typename> deq1, deq2(arr,arr+n),
      deq3{arr,arr+n};
```

- functions *assign*, *at*, *begin*, *back*, *end*, *emplace*, *empty*, *erase*, *insert*, *operator=*, *operator[]*, *pop_back*, *push_back*, *resize*, *size*, *swap* are same as that of vectors.
- **front**: Returns a reference to the first element

  ```
  deq.front();
  ```
- **push_front**:

  ```
  deq.push_front();
  ```
- **pop_front**:

  ```
  deq.pop_front();
  ```

set

- Sets store unique elements following a specific order indicated by the comparison object.
- The value of the elements in the set cannot be modified but they can be inserted and removed.
- Sets are generally implemented as Binary Search Trees.

Member Functions:

- Functions *begin*,*end*,*empty*,*erase*,*clear*,*operator=*,*insert*,*swap*,*size* are the same.
- **count**: Returns the count of the value passed as parameter 0 if not present.

  ```
  s.count(val);
  ```

- **equal_range**: Returns a pair of iterators representing the bounds of the range containing the elements equal to the parameter.

  ```
  pair<set<tyepname>::const_iterator,
       set<typename>::const_iterator> p_iter =
                                  set_var.equal_range(val);
  ```

- **find**: Searches for a value and returns a *const_iterator* to it if found,*set::end* otherwise.

```
set<typename>::const_iterator iter= s.find(val);
```

- **lower_bound**: Returns an iterator pointing to the first element which would not come before *val*(even if equivalent to *val*).

```
set<typename>::iterator iter=s.lower_bound(val);
```

- **upper_bound**: Returns an iterator pointing to the first element which would come after *val*.

```
set<typename>::iterator iter= s.upper_bound(val);
```

map

- Maps store elements in the form of *key-value* pairs following a specific order indicated by its comparison object.
- Key values are used to sort the *key-value* pairs.
- Typically implemented using Balanced Binary Search Trees.

Usage:

```
#include <map>
```

Initialization:

```
map<T1,T2> myMap;
map<T1,T2> myMap(map_iterator_start,map_iterator_end);
map<T1,T2> myMap(map_obj); map<T1,T2,comp_func_object> myMa
map<T1,T2,bool(*)(T1,T2)> myMap(comp_func_ptr);
```

- **at** or **operator[]**: Returns the reference to the mapped value corresponding to the key, throws *out_of_range* exception if key is not found.

```
T2 &var= myMap.at(key);
T2 var= myMap.at(key);
T2 var= myMap[key];
```

- **begin** and **end**: Return iterators pointing to the first and past-the-last elements of the map

```
map<T1,T2>::iterator iter= myMap.begin();
map<T1,T2>::iterator iter2= myMap.end();
```

- Functions *clear*, *size*, *empty*, *upper_bound*, *lower_bound*, *find*, *swap*, *count*,*insert* are same as for set.

- **erase**: Removes either a single element or a range of elements. Can use both, iterator(s) or keys to remove elements.

```
map<T1,T2>::iterator iter= myMap.find(key);
myMap.erase(iter);
myMap.erase(key);
myMap.erase(map_iter_start,map_iter_end);
```

- **operator=**: Supports assignment of contents of one to another.
- **equal_range**: Returns a pair of iterators pointing to the first and last elements of a range having all keys equivalent to the *key*

```
pair<map<T1,T2>::iterator,map<T1,T2>::iterator> ret
                                    = myMap.equal_range(key);
```