STL

Rupesh Nasre.

CS2810 OOAIA
January 2025

Standard Template Library

- Provides common data structures and algorithms
- Provides uniform interface (as much as possible)
  - e.g., push_back, iterators
- Relieves programmers of memory management
- Well-optimized, encouraged to use, reduce effort
- Four parts
  - Containers, algorithms, iterators, functors

Containers

- Sequence containers
  - vector, list, deque, array, forward_list (C++11)
- Associative containers
  - set, multiset, map, multimap
  - and their unordered variants
- Container adaptors
  - Restricted forms of first-class containers.
  - queue, priority_queue, stack
  - queue and stack are restrictions on deque.
  - priority_queue is a restriction on vector.

Source: geeksforgeeks.com

vector

- Dynamic arrays
- Contiguous storage
- Storage handled internally, without involving user
  - Interface vs. Implementation
- Insert at the end (push_back) takes different times.
- Delete at the end (pop_back) is constant time.
- Other inserts and deletes are linear in time in terms of size.

```
vector<int> v;
for (ii=0; ii<n; ++ii) v.push_back(ii);
```

vs

```
vector<int> v;
v.reserve(n);
for (ii=0; ii<n; ++ii) v.push_back(ii);
```

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

int main() {
    vector <int> g1;
    vector <int> :: iterator it;
    vector <int> :: reverse_iterator ir;

    for (int i = 1; i <= 5; i++)
        g1.push_back(i);
    cout << "Output of begin and end\t:\t";
    for (it = g1.begin(); it != g1.end(); ++it)
        cout << *it << '\t';

    cout << endl << endl;
    cout << "Output of rbegin and rend\t:\t";
    for (ir = g1.rbegin(); ir != g1.rend(); ++ir)
        cout << '\t' << *ir;
    vector<int> g2 = {5, 4, 3, 2, 1};
    vector<int> g3(5, 0);

    return 0;

}
```

list

- Slow traversal
- Fast insert, delete
- Internally, doubly linked.
- For singly linked, use **forward_list**
- Other functions: insert, erase, remove, reverse.
- Be careful about [ ].

```
void showlist(list <int> g) {
    list <int> :: iterator it;
    for (it = g.begin(); it != g.end(); ++it)
        cout << '\t' << *it;
    cout << '\n';
}
int main() {
    list <int> gqlist1, gqlist2;
    for (int i = 0; i < 10; ++i) {
        gqlist1.push_back(i * 2);
        gqlist2.push_front(i * 3);
    }
    cout << "\nList 1 (gqlist1) is : ";
    showlist(gqlist1);
    cout << "\nList 2 (gqlist2) is : ";
    showlist(gqlist2);
    cout << "\n1.front() : " << gqlist1.front();
    cout << "\n1.back() : " << gqlist1.back();
    return 0;
}
```

forward_list

- Singly linked list.
  - Less space.
- Cannot be iterated backward.
- Useful for
  - Chaining in hashing
  - Adjacency lists in graphs
  - ...
- Supports push/pop_front; does not support _back.

```
int main() {
    forward_list<int> flist1;
    forward_list<int> flist2;

    flist1.assign({1, 2, 3});

    // 5 elements with value 10
    flist2.assign(5, 10);

    cout << "First flist is: ";
    for (int &a: flist1)
        cout << a << " ";
    cout << endl;

    cout << "Second flist is: ";
    for (int &b: flist2)
        cout << b << " ";
    cout << endl;

    return 0;
}
```

array

- It is a class, so various C++ features apply.
- Size need not be passed as another parameter.
  - ar.size()
- Supported with C++11 *(g++ -std=c++11 file.cpp)*.

```
#include<iostream>
#include<array>
using namespace std;
int main() {
    array<int,6> ar = {1, 2, 3, 4, 5, 6};

    // Printing array elements using at()
    cout << "The array elemets are: ";
    for (int i=0; i<6; i++)
      cout << ar.at(i) << " ";
    cout << endl;

    // Printing array elements using operator[]
    cout << "The array elements are: ";
    for (int i=0; i<6; i++)
      cout << ar[i] << " ";
    cout << endl;

    return 0;
}
```

stack

- LIFO
- Insert at and remove from the same end.
- Can be implemented using deque.
- Applications: compilers, OS, ...

```
void showstack(stack <int> gq) {
    stack <int> g = gq;
    while (!g.empty()) {
        cout << '\t' << g.top();
        g.pop();
    }
    cout << '\n';
}
int main () {
    stack <int> gquiz;
    gquiz.push(10);
    gquiz.push(30);
    gquiz.push(20);
    gquiz.push(5);
    gquiz.push(1);

    cout << "The stack gquiz is : ";
    showstack(gquiz);

    cout << "\ngq.size() : " << gquiz.size();
    cout << "\ngq.top() : " << gquiz.top();

    return 0;
}
```

queue

- FIFO
- Insertion at the end
- Deletion from the front
- Can be implemented using deque.
- Applications: process scheduling, banks, graph traversal, ...

```
void showq(queue <int> gq) {
    queue <int> g = gq;
    while (!g.empty()) {
        cout << '\t' << g.front();
        g.pop();
    }
    cout << '\n';
}
int main() {
    queue <int> gquiz;
    gquiz.push(10);
    gquiz.push(20);
    showq(gquiz);

    cout << "\ngq.size() : " << gquiz.size();
    cout << "\ngq.front() : " << gquiz.front();
    cout << "\ngq.back() : " << gquiz.back();

    gquiz.pop();
    showq(gquiz);

    return 0;
}
```

deque

- Expands and contracts efficiently at both the ends.
- Contiguous storage is not guaranteed.
- Internally, vector of arrays / vectors.
- Constant time for push/pop front/back.
- (amortized) Constant time for [ ].

```
void showdq(deque <int> g) {
    deque <int> :: iterator it;
    for (it = g.begin(); it != g.end(); ++it)
        cout << '\t' << *it;
    cout << '\n';
}
int main() {
    deque <int> gquiz;
    gquiz.push_back(10);
    gquiz.push_front(20);
    gquiz.push_back(30);
    gquiz.push_front(15);
    cout << "The deque gquiz is : ";
    showdq(gquiz);

    cout << "\ngq.size() : " << gquiz.size();
    cout << "\ngq.at(2) : " << gquiz.at(2);
    cout << "\ngq.front() : " << gquiz.front();
    cout << "\ngq.back() : " << gquiz.back();

    return 0;
}
```

priority_queue

- First element is the largest.
- Implemented as a heap.
- Stored in a vector.
- Applications: priority scheduling, flight landing, graph algorithms, ...

```
void showpq(priority_queue <int> gq) {
    priority_queue <int> g = gq;
    while (!g.empty()) {
        cout << '\t' << g.top();
        g.pop();
    }
    cout << '\n';
}
int main () {
    priority_queue <int> gquiz;
    gquiz.push(10);
    gquiz.push(30);
    gquiz.push(20);
    gquiz.push(5);
    gquiz.push(1);
    showpq(gquiz);
    cout << "\ngq.size() : " << gquiz.size();
    cout << "\ngq.top() : " << gquiz.top();
    return 0;
}
```

set

- Unique values
  - Cannot change elements
- Implemented as some BST.
- Other functions: erase, clear, find, upper_bound, lower_bound
- Applications requiring fast lookups

```
int main() {
    set <int, greater <int> > gquiz1;

    gquiz1.insert(40);
    gquiz1.insert(30);
    gquiz1.insert(60);
    gquiz1.insert(20);
    gquiz1.insert(50);
    gquiz1.insert(50);
    gquiz1.insert(10);
    set <int, greater <int> > :: iterator itr;
    cout << "\nThe set gquiz1 is : ";
    for (itr = gquiz1.begin();
         itr != gquiz1.end(); ++itr) {
        cout << '\t' << *itr;
    }
    cout << endl;
    return 0;
}
```

multiset

- Duplicate values
  - Cannot change elements
- Implemented as some BST.
- Other functions: erase, clear, find, upper_bound, lower_bound, count

```
int main() {
    multiset <int, greater <int> > gquiz1;

    gquiz1.insert(40);
    gquiz1.insert(30);
    gquiz1.insert(60);
    gquiz1.insert(20);
    gquiz1.insert(50);
    gquiz1.insert(50);
    gquiz1.insert(10);

    gquiz1.erase(50);
    gquiz1.erase(gquiz1.begin(),
                 gquiz1.find(30));

    return 0;
}
```

map

- Key-Value pairs
- Key is unique
  - *gquiz1[4]*
- Implemented as Red-Black trees
- Other functions: find, count, clear, erase
- Applications: relational DB, semi-structured data

```
int main() {
    map <int, int> gquiz1;

    gquiz1.insert(pair <int, int> (1, 40));
    gquiz1.insert(pair <int, int> (2, 30));
    gquiz1.insert(pair <int, int> (3, 60));
    gquiz1.insert(pair <int, int> (4, 20));
    gquiz1.insert(pair <int, int> (5, 50));
    gquiz1.insert(pair <int, int> (6, 50));
    gquiz1.insert(pair <int, int> (7, 10));

    map <int, int> :: iterator itr;
    cout << "\nThe map gquiz1 is : \n";
    for (itr = gquiz1.begin();
         itr != gquiz1.end(); ++itr)
        cout  <<  '\t' << itr->first
              <<  '\t' << itr->second << '\n';
    cout << endl;
    return 0;
}
```

multimap

- Multiple values can have the same key
- <key, value> is unique
  - *gquiz1[4]* is an error.
- Implemented as Red-Black trees
- Other functions: find, count, clear, erase

```
int main() {
    multimap <int, int> gquiz1;

    gquiz1.insert(pair <int, int> (1, 40));
    gquiz1.insert(pair <int, int> (2, 30));
    gquiz1.insert(pair <int, int> (3, 60));
    gquiz1.insert(pair <int, int> (4, 20));
    gquiz1.insert(pair <int, int> (5, 50));
    gquiz1.insert(pair <int, int> (6, 50));
    gquiz1.insert(pair <int, int> (6, 10));

    multimap <int, int> :: iterator itr;
    cout << "\nThe multimap gquiz1 is : \n";

    for (itr = gquiz1.begin();
         itr != gquiz1.end(); ++itr)
        cout  <<  '\t' << itr->first
              <<  '\t' << itr->second << '\n';
    cout << endl;
    return 0;
}
```