Creative Software Programming Practice (week-7-1)

There are no assignments this week because next week is the exam(Oct. 21 Wed 3pm @ITBT 508).

There is face-to-face QA in next class(Oct. 15 Thu 1pm @ITBT 508). *You can also attend online if you want*.

Topics

1. STL
2. STL Containers
3. STL Iterator
4. STL List: Linked List
5. STL Stack and Queue
6. STL set
7. STL map

1. STL

The Standard Template Library (STL) is a set of C++ template classes to provide common programming data structures and functions such as lists, stacks, arrays, etc. It is a library of container classes, algorithms, and iterators. It is a generalized library and so, its components are parameterized. A working knowledge of template classes is a prerequisite for working with STL.

STL has four components

- Containers: Data structures that store objects of any type.
- Iterators: Used to manipulate container elements.
- Algorithms: Operations on containers for searching, sorting and many others
- Functions: overload the function call operator.

2. Containers

Seqeunce container

- vector: fast insertion at end, random access
- list: fast insertion anywhere, sequential access
- deque: fast insertion at either end, random access

Container adapter

- stack: LIFO (Last In First Out)
- queue: FIFO (First In First Out)

Associative container

- set: add or delete (unique items)
- map: mapping from key to value
- multimaps: mapping key to multiple values

std::vector

- Element are stored in contiguous storage(memory).
- Random access: Fast access to random element.
- Fast addition/removal of elements at the end of sequence.

```cpp
// vector.cc
#include <iostream>
#include <vector>

int main() {
    std::vector<int> v;

    // size is vector size, capacity is allocated memory space
    // Vectors allocate a larger amount of memory for efficiency.
    // This prevents reassignment every time the size increases.
    // Note that reallocating memory is expensive.
    std::cout << "size: " << v.size() << std::endl;
    std::cout << "capacity: " << v.capacity() << std::endl;
    for (int i = 0; i < 5; i++) {
        v.push_back(i);
        std::cout << "size: " << v.size() << std::endl;
        std::cout << "capacity: " << v.capacity() << std::endl;
    }
    // so, the size is 5, but capa is 8
    // Yes, Basically, capacity is allocated as much as a power of 2.

    // resize 7 < capacity 8
    // So, reallocation no required
    v.resize(7);
    std::cout << "size: " << v.size() << std::endl;
    std::cout << "capacity: " << v.capacity() << std::endl;

    // Executing pop back does not reduce capacity.
    v.pop_back();
    v.pop_back();
    v.pop_back();
    v.pop_back();
    std::cout << "size: " << v.size() << std::endl;
    std::cout << "capacity: " << v.capacity() << std::endl;

    for (auto e : v) {
        std::cout << e << ", ";
    }
    std::cout << std::endl;

    return 0;
}
```

3. STL Iterator

Iterator is a pointer-like object pointing to an element in the container.
Iterator provide a generalized way to traverse and access elements stored in a container.

- can be ++ or -- to move next, prev.
- dereferenced with *

- compared against another iterator with == or !=

Iterator are generated by STL container member functions

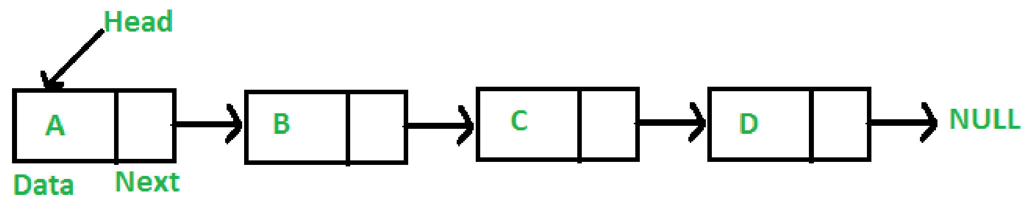
- begin(), end()
- rbegin(), rend() // reverse iterator

end() and rend() is not last element (next of last element)

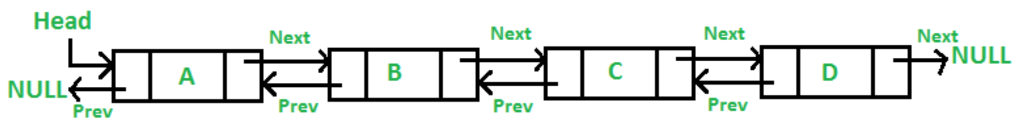
```
// vector.cc
#include <iostream>
#include <vector>

int main() {
    std::vector<int> v(5);

    v[0] = 1;
    v[1] = 2;
    v[2] = 3;
    v[3] = 4;
    v[4] = 5;

    for (std::vector<int>::iterator it = v.begin(); it != v.end(); it++) {
        std::cout << *it << ", ";
    }
    std::cout << std::endl;

    std::vector<int>::iterator it = v.begin();

    std::cout << *it << std::endl;          // the value of v[0]
    std::cout << *(it + 3) << std::endl;    // the value of v[3]

    // insert 10 before v[3], so new v[3] will be 10
    v.insert(v.begin() + 3, 10);
    for (auto e : v) {
        std::cout << e << ", ";
    }
    std::cout << std::endl;

    // remove v[4] the value is 4.
    v.erase(v.begin() + 4);
    for (auto e : v) {
        std::cout << e << ", ";
    }
    std::cout << std::endl;

    // iterator loop
    for(std::vector<int>::iterator it = v.begin(); it != v.end(); it++) {
        std::cout << *it << ", ";
    }
    std::cout << std::endl;

    // reverse iterator
    for (std::vector<int>::reverse_iterator it = v.rbegin();
         it != v.rend(); it++) {
        std::cout << *it << ", ";
    }
    std::cout << std::endl;
    return 0;
}
```

4. Linked List

Linked List



Double Linked List



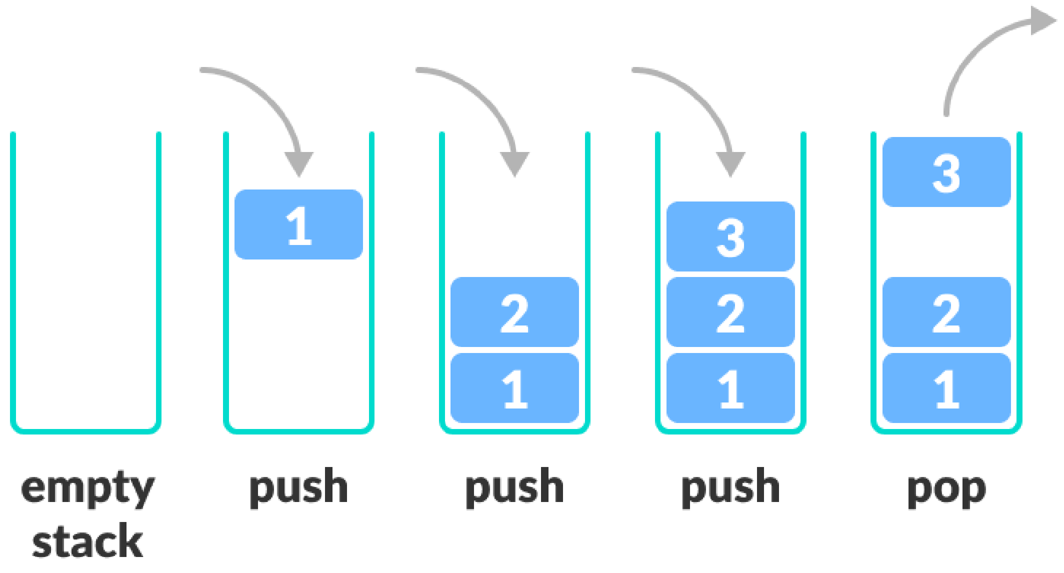
Since the linked list is a structure where nodes are connected, each node points to the next (or previous) node. So, you can easily remove nodes or insert nodes.

- STL List is implemented as a double-linked list. (not contiguous storage(memeory))
- Sequetial access (you must access nodes sequentially, not random access like array)
- Fast addition/removal of element anywhere of nodes

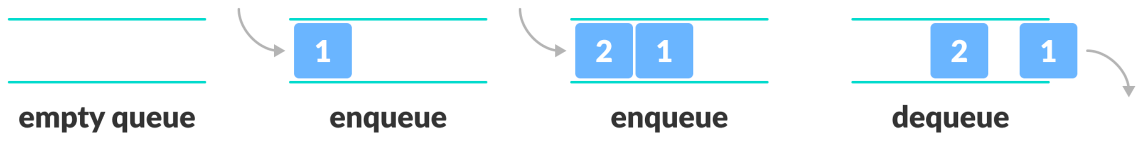
```
#include <iostream>
#include <list>

int main() {
    std::list<int> l;
    l.push_back(3);
    l.push_back(4);
    l.push_back(5);
    l.push_back(6);

    for (auto e : l) {
        std::cout << e << ", ";
    }
    std::cout << std::endl;

    // list also have iterator
    std::list<int>::iterator it = l.begin();

    // but list is not cotingous, so you cannot add iterator like vector
    // std::cout << *(it + 3) << std::endl;

    // you must access items sequentially
    it++;
    std::cout << *it << std::endl;

    return 0;
}
```

5. STL Stack and Queue

Stack is Last In First Out data structure.



Queue is First In First Out data structure.



*But **Cafe Queue** do not always come out FIFO order*

```
// stack.cc
#include <stack>
#include <iostream>

int main() {
    std::stack<int> stack;

    stack.push(10);
    stack.push(20);
    stack.push(30);

    std::cout << stack.top() << std::endl;
    stack.pop();
    std::cout << stack.top() << std::endl;
    stack.pop();
    std::cout << stack.top() << std::endl;
    stack.pop();

    return 0;
}

// queue.cc
```

```
#include <queue>
#include <iostream>

int main() {
    std::queue<int> queue;

    queue.push(10);
    queue.push(20);
    queue.push(30);

    std::cout << queue.front() << std::endl;
    queue.pop();
    std::cout << queue.front() << std::endl;
    queue.pop();
    std::cout << queue.front() << std::endl;
    queue.pop();

    return 0;
}
```

6. STL set

Set is container for unique keys. If you insert duplicated key, container ignore duplicate. Only store unique keys. Elements are referenced by their key, and maintained in sorted key order.

- *If you want to multiple key, you can use multiset*
- *If the order doesn't matter, you can use unordered_set.*

```
#include <set>
#include <iostream>

int main() {
    std::set<int> s;

    s.insert(3);
    s.insert(3);
    s.insert(4);
    s.insert(5);
    s.insert(5);
    s.insert(3);

    std::cout << "size: " << s.size() << std::endl;

    // if you using unordered_set, than key is not sorted
    // but default set is sorting key
    for (std::set<int>::iterator it = s.begin(); it != s.end(); it++) {
        std::cout << (*it) << ", ";
    }
    std::cout << std::endl;

    // you can check key exists with find
    // if result of find is end() means there is no key!
    if (s.find(3) != s.end()) {
        std::cout << "3 in set" << std::endl;
    }
    if (s.find(8) == s.end()) {
        std::cout << "8 is not in set" << std::endl;
    }

    return 0;
}
```

7. STL map

Map is container for key-value. Also map store unique key.

- *If you want to multiple key, you can use multimap*
- *If the order doesn't matter, you can use unordered_map.*

```
#include <map>
#include <iostream>

int main() {
    // map require two template variable <key, value>
    std::map<int, int> m;

    m.insert({3, 9});
    m.insert({2, 4});
    m.insert({1, 1});
    m.insert({2, 14});

    std::cout << "size: " << m.size() << std::endl;

    // if you using unordered_map, than key is not sorted
    // but default map is sorting key
    for (std::map<int, int>::iterator it = m.begin(); it != m.end(); it++) {
        std::cout << it->first << ": " << it->second << std::endl;
    }

    // you can check key exists with find
    // if result of find is end() means there is no key!
    if (m.find(3) != m.end()) {
        std::cout << "3 in map" << std::endl;
    }
    if (m.find(8) == m.end()) {
        std::cout << "8 is not in map" << std::endl;
    }

    return 0;
}
```

No assignments this week. I'm sorry for lot of practice today.