11. Program set 8 – Standard template library

1) std::vector is a resizable, contiguous array managed with simple interfaces. The vector is a template library that does allocation and deletion by itself. In the program, v[i] does not if access is out of bounds. v.reserve(n) can be used to reserve n vector elements.

```
#include <iostream>
#include <vector>
int main() {
    std::vector<int> v;

    std::cout << "Number of elements that can be fit " <<
        "before the vector re-allocates " << v.capacity() <<
std::endl;

    v.push_back(3);

    std::cout << "Number of elements that can be fit " <<
        "before the vector re-allocates " << v.capacity() <<
std::endl;

    v.push_back(1);

    std::cout << "Number of elements that can be fit " <<
        "before the vector re-allocates " << v.capacity() <<
std::endl;

    v.push_back(4);

    for (std::size_t i=0;i<v.size();++i)
        std::cout<<v[i]<<" ";

    std::cout<<"\n";

    std::cout << "Number of elements that can be fit " <<
        "before the vector re-allocates " << v.capacity() <<
std::endl;

    v.push_back(5);

    std::cout << "Number of elements that can be fit " <<
        "before the vector re-allocates " << v.capacity() <<
std::endl;
```

```
    v.push_back(10);

    std::cout << "Number of elements that can be fit " <<
        "before the vector re-allocates " << v.capacity() <<
std::endl;


}
```

Random access and non-random access arrays

In random access arrays, it is possible to jump to any possible position in the array in one step. vector, deque, and raw arrays are examples of random access arrays. Jumping to a location such as using it + 5, it[10] is allowed.

In non-random access arrays, in order to get to a certain element, the array needs to be walked link by link. it + n or it[n] is not allowed. ++it is allowed (and in fewer cases, --it is also allowed if the array allows bidirectional traversal).

Iterators

Iterators can be thought of as generalized pointers that refer to an element in a sequence. It can be moved forward or sometimes backward and dereferenced to access the element at that location. Two iterators that point to the beginning and end of the sequence are returned by begin() and end() respectively.

2) std::vector with std::sort. The line `std::vector<int> v(a, a + 4)` copies a from starting to $(a + 3)^{rd}$ element, but not including $(a + 4)^{th}$ element. This is called a half-open convention, as the ending element in the copy is one less (or not equal) to the ending element of the array a. This half-open convention is used throughout the STL.

```
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    int a[] = {5,2,9,1};

    std::vector<int> v(a, a+4);

    std::sort(v.begin(), v.end());

    for (std::size_t i=0;i<v.size();++i)
```

```
        std::cout<<v[i]<<" ";
    std::cout<<"\n";

    std::sort(v.begin(), v.end(), std::greater<int>());

    for (std::size_t i=0;i<v.size();++i)
        std::cout<<v[i]<<" ";
    std::cout<<"\n";

    return 0;
}
```

3) std::list, a doubly-linked list.

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

int main() {
    std::list<int> L;
    L.push_back(1); L.push_front(2); L.push_back(3);
    for (std::list<int>::iterator it=L.begin(); it!=L.end(); ++it)
        std::cout<<*it<<" ";
    std::cout<<"\n";

    L.sort();
    for (std::list<int>::iterator it=L.begin(); it!=L.end(); ++it)
        std::cout<<*it<<" ";
    std::cout<<"\n";

    return 0;
}
```

4) std::deque — double-ended queue (fast at both ends)

```
#include <iostream>
#include <deque>

int main() {
    std::deque<int> d;      // segmented array under the hood

    d.push_front(2);        // [2]
    d.push_back(3);         // [2,3]
    d.push_front(1);        // [1,2,3]
```

```
    // Random-access indexing like vector
    for (std::size_t i = 0; i < d.size(); ++i)
       std::cout << d[i] << " ";
    std::cout << "\n";
}
```

5) std::set contains ordered, unique elements.

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

int main() {
    std::set<int> s;       // usually a balanced BST (e.g., red-black
tree)

    s.insert(3);
    s.insert(1);
    s.insert(3);           // duplicate ignored

    for (std::set<int>::iterator it = s.begin(); it != s.end(); ++it)
        std::cout << *it << " ";     // prints in sorted order: 1 3  5
    std::cout << "\n";
}
```

6) std::multiset – ordered, allows duplicates

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

int main() {
    std::multiset<int> ms;

    ms.insert(3);
    ms.insert(1);
    ms.insert(3);    // duplicates are stored

    std::cout << "count(3) = " << ms.count(3) << "\n";  // likely 2
}
```

7) std::map, ordered dictionary (key with value)

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

int main() {
    std::map<std::string, int> freq;  // keys sorted lexicographically

    ++freq["apple"];   // inserts "apple" with 0 then increments to 1
    ++freq["banana"];  // inserts "banana" with 0 then increments to 1
    ++freq["apple"];   // increments existing value to 2

    for (std::map<std::string,int>::iterator it = freq.begin(); it != 
freq.end(); ++it) {
        std::cout << it->first << ": " << it->second << "\n";
    }
}
```

8) std::multimap — one key to many values (grouped by key)

```
#include <iostream>
#include <map>
#include <string>
#include <utility>

int main() {
    std::multimap<std::string, int> mm;

    mm.insert(std::make_pair("grp", 1));
    mm.insert(std::make_pair("grp", 2));
    mm.insert(std::make_pair("other", 9));

    // equal_range returns the [first,last) subrange with key "grp"
    std::pair<std::multimap<std::string,int>::iterator,
              std::multimap<std::string,int>::iterator> r = 
mm.equal_range("grp");

    for (std::multimap<std::string,int>::iterator it = r.first; it != 
r.second; ++it) {
        std::cout << it->second << " ";
    }
    std::cout << "\n";
}
```

9) std::pair and std::make_pair used to create tiny 2-tuple

```cpp
#include <iostream>
#include <utility>
#include <string>

int main() {
    std::pair<std::string, int> p = std::make_pair(std::string("id"),
42);
    std::cout << p.first << " " << p.second << "\n";
}
```

10) std::stack – LIFO adaptor over a container. std::stack<T> is a container adaptor: it wraps another container and makes available only LIFO operations: push(const T&), pop(), top(), plus empty(), size().

```cpp
#include <iostream>
#include <stack>

int main() {
    std::stack<int> st;  // default underlying container is deque<int>

    st.push(10);
    st.push(20);

    std::cout << st.top() << "\n";   // 20 (last in, first out)
    st.pop();                        // removes 20
    std::cout << st.top() << "\n";   // 10
}
```

11) std::queue. std::queue<T> is a container adaptor that exposes FIFO (first-in, first-out) operations: push, pop, front, back, plus empty, size.

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

int main() {
    std::queue<int> q;   // default container is deque<int>

    q.push(1);
    q.push(2);
    q.push(3);
```

```
    std::cout << q.front() << " " << q.back() << "\n";
                                          // 1 (front), 3 (back)
    q.pop();                    // removes 1
    std::cout << q.front() << "\n";   // now 2
}
```

12) std::priority_queue – it is an adaptor for vector STL. By default, the maximum element is at the top.

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

#include <functional>

int main() {
    std::priority_queue<int> pq;
    pq.push(5);
    pq.push(1);
    pq.push(9);

    while(!pq.empty()){ std::cout<<pq.top()<<" "; pq.pop(); }
    std::cout<<"\n";

    std::priority_queue<int, std::vector<int>, std::greater<int> >
minq;
    minq.push(5);
    minq.push(1);
    minq.push(9);

    while(!pq.empty()){ std::cout<<pq.top()<<" "; pq.pop(); }
    std::cout<<"\n";

    return 0;
}
```

13) std::bitset. Bit set operations can be done by representing the binary as a string

```
#include <iostream>
#include <bitset>
#include <string>

int main() {
    std::bitset<8> b(std::string("00010110"));  // 8 bits: 0b00010110
```

```
    std::cout << "bits set = " << b.count() << "\n"; // number of 1s
    std::cout << "b[1] = " << b[1] << "\n";     // index from LSB at 0
    b.flip(2);                                      // toggle bit #2
    std::cout << b << "\n";                    // print as 8 chars 0/1
}
```

14) Iterators

```
#include <iostream>
#include <vector>

int main() {
    int a[] = {3, 1, 4};

    std::vector<int> v(a, a + 3);

    for (std::vector<int>::iterator it = v.begin(); it != v.end();
++it) {
        std::cout << *it << " ";     // dereference to get element
    }
    std::cout << "\n";
}
```

15) Output iterator adapter. std::back_inserter + std::copy

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>

int main() {
    int a[] = {1, 2, 3};

    std::vector<int> v;                // empty; size grows during copy
    std::copy(a, a + 3, std::back_inserter(v));
                                    // internally calls v.push_back(...)

    for (std::size_t i = 0; i < v.size(); ++i)
       std::cout << v[i] << " ";
    std::cout << "\n";
}
```

16) std::find + std::count does linear search/count

```
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    int a[] = {1, 3, 3, 7};

    std::vector<int> v(a, a + 4);
    std::vector<int>::iterator it = std::find(v.begin(), v.end(), 3);

    if (it != v.end()) {
        std::cout << "first 3 at index " << (it - v.begin()) << "\n";
    }
    std::cout << "count(3) = " << std::count(v.begin(), v.end(), 3) <<
"\n";
}
```

17) Erase–remove idiom. remove() method in this example arranges all elements so that all instances of integer 2 are at the end, then returns an end iterator so that all 2s are excluded. remove() method actually shrinks the vector so that all 2s are deleted.

```
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    int a[] = {1, 2, 3, 2, 4};
    std::vector<int> v(a, a + 5);
    // remove returns new logical end after moving survivors forward
    v.erase(std::remove(v.begin(), v.end(), 2), v.end());
    for (std::size_t i = 0; i < v.size(); ++i) std::cout << v[i] << " ";
    std::cout << "\n";
}
```

18) Deduplicate with std::unique. sort() method sorts the vector in ascending order. unique() method keeps all unique integers and moves repeated integers to the end, and returns an end iterator pointing just after the end of the unique elements. erase() method deletes the duplicates at the end of the vector.

```
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    int a[] = {3, 1, 3, 2, 2, 1};
```

```
    std::vector<int> v(a, a + 6);

    std::sort(v.begin(), v.end());                          // 1 1 2 2 3 3
    v.erase(std::unique(v.begin(), v.end()), v.end());     // 1 2 3

    for (std::size_t i = 0; i < v.size(); ++i)
       std::cout << v[i] << " ";
    std::cout << "\n";
}
```

19) Binary search helpers – equal_range

```
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    int a[] = {1,2,2,2,5};

    std::vector<int> v(a,a+5);
    std::sort(v.begin(), v.end());

    std::pair<std::vector<int>::iterator, std::vector<int>::iterator>
r =
        std::equal_range(v.begin(), v.end(), 2);

    std::cout<<"2s in ["<<(r.first - v.begin())<<", "<<(r.second -
v.begin())<<")\n";
}
```

20) std::stable_sort with custom comparator (sort by length, keep order)

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <string>

class ByLen {
    bool operator()(const std::string& a, const std::string& b) const
       {
        if (a.size() != b.size()) return a.size() < b.size();
```

```
        return a < b; // tiebreak (required for strict weak ordering)
    }
};

int main() {
    std::string a[] = {"bb","a","ccc","aa"};

    std::vector<std::string> v(a, a + 4);
    std::stable_sort(v.begin(), v.end(), ByLen());

    for (std::size_t i = 0; i < v.size(); ++i)
       std::cout << v[i] << " ";
    std::cout << "\n";
}
```

21) std::transform + toupper – element-wise mapping

```
#include <iostream>
#include <algorithm>
#include <string>
#include <cctype>

int up(int c) {
       return std::toupper((unsigned char)c);
}

int main() {
    std::string s = "MiT";
    std::transform(s.begin(), s.end(), s.begin(), up);
                                                  // in-place upper
    std::cout << s << "\n";
}
```

22) <numeric>, std::accumulate for sum/product

```
#include <iostream>
#include <vector>
#include <numeric>
#include <functional>

int main() {
    int a[] = {1,2,3,4};
    std::vector<int> v(a,a+4);
    int sum  = std::accumulate(v.begin(), v.end(), 0);
```

```
                                                        // 0 is init
    int prod = std::accumulate(v.begin(), v.end(), 1,
std::multiplies<int>()); // 1 is init
    std::cout << sum << " " << prod << "\n";   // 10 24
}
```

23) Set algorithms, union & intersection on sorted ranges

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>

int main() {
    int A[] = {1,2,4}, B[] = {2,3,4};
    std::vector<int> a(A,A+3), b(B,B+3), out;

    std::set_union(a.begin(), a.end(), b.begin(), b.end(),
std::back_inserter(out));
    for (std::size_t i=0;i<out.size();++i)
       std::cout<<out[i]<<" "; std::cout<<"\n";

    out.clear();

    std::set_intersection(a.begin(), a.end(), b.begin(), b.end(),
std::back_inserter(out));
    for (std::size_t i=0;i<out.size();++i) std::cout<<out[i]<<" ";
std::cout<<"\n";
}
```

24) std::partition, split by predicate given by the () constructor in IsEven structure

```
#include <iostream>
#include <vector>
#include <algorithm>

struct IsEven {
       bool operator()(int x) const {
              return x % 2 == 0;
       }
};

int main() {
    int A[] = {1,2,3,4,5,6};
```

```
    std::vector<int> v(A,A+6);

    std::partition(v.begin(), v.end(), IsEven());
                                            // evens first, odds after

    for (std::size_t i=0;i<v.size();++i)
       std::cout<<v[i]<<" ";
    std::cout << "\n";
}
```

25) std::remove_if + erase removes using the predicate given by the LessThan3 struct

```
#include <iostream>
#include <vector>
#include <algorithm>

struct LessThan3 {
       bool operator()(int x) const {
              return x < 3;
       }
};

int main() {
    int A[] = {1,2,3,4};
    std::vector<int> v(A,A+4);

    v.erase(std::remove_if(v.begin(), v.end(), LessThan3()), v.end());

    for (std::size_t i=0;i<v.size();++i)
       std::cout<<v[i]<<" ";
    std::cout << "\n";
}
```

26) std::map with custom comparator (descending keys)

```
#include <iostream>
#include <map>
#include <string>
#include <functional>

int main() {
    std::map<int, std::string, std::greater<int> > m;
                                            // sort keys high to low
```

```
    m[2] = "two";
    m[1] = "one";
    m[3] = "three";
    for (std::map<int,std::string, std::greater<int> >::iterator it =
m.begin(); it != m.end(); ++it)
        std::cout << it->first << " ";
    std::cout << "\n";
}
```

27) std::stable_partition, split by predicate, keep order

```
#include <iostream>
#include <algorithm>
#include <string>

struct IsUpper {
       bool operator()(char c) const {
              return c >= 'A' && c <= 'Z';
       }
};

int main() {
    std::string s = "aAbBcC";

    std::stable_partition(s.begin(), s.end(), IsUpper());

    std::cout << s << "\n";
       // "ABCabc" — uppercase chunk keeps original order (A,B,C)
}
```

28) std::rotate

```
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    int A[] = {1,2,3,4,5};
    std::vector<int> v(A,A+5);

// move [begin, begin+2) to the end: [1,2 | 3,4,5]  becomes[3,4,5,1,2]
    std::rotate(v.begin(), v.begin() + 2, v.end());

    for (std::size_t i=0;i<v.size();++i)
```

```
        std::cout<<v[i]<<" ";
    std::cout << "\n";
}
```

29) std::replace / std::replace_if in-place value change

```
#include <iostream>
#include <vector>
#include <algorithm>

struct IsOdd {
        bool operator()(int x) const {
                return x % 2 == 1;
        }
};

int main() {
    int A[] = {1,2,3,4,5};
    std::vector<int> v(A,A+5);

    std::replace(v.begin(), v.end(), 2, 20);        // replace 2 to 20
    std::replace_if(v.begin(), v.end(), IsOdd(), 99);
                                            // replace all odd to 99

    for (std::size_t i=0;i<v.size();++i)
        std::cout<<v[i]<<" ";
    std::cout << "\n";
}
```

30) Stream iterators + std::map – word frequency counter

```
#include <iostream>
#include <iterator>
#include <string>
#include <map>

int main() {
    std::istream_iterator<std::string> in(std::cin);
                                            // read words from stdin
    std::istream_iterator<std::string> end;

    std::map<std::string, int> freq;

    while (in != end) {
```

```
        ++freq[*in];   // increment count for the read word
        ++in;          // advance the input iterator
    }
    for (std::map<std::string,int>::iterator it = freq.begin(); it !=
freq.end(); ++it)
        std::cout << it->first << ": " << it->second << "\n";
}
```