Lecture 7 & 8

STL
Vector, Set, Map, Stack, Queue, Deque

STL

STL - Standard Template Library

It provides four components called *algorithms*, *containers*, *functions*, and *iterators*.

There are three types of containers:

- sequence containers (e.g: vector, deque)
- associative containers (e.g: pair, set, map)
- container adaptors (e.g: stack, queue)

STL containers, ejudge reference

STL containers

Sequence containers

- `vector` - Rapid insertions and deletions at back. Direct access to any element.
- `deque` - Rapid insertions and deletions at front or back. Direct access to any element.

Associative containers

- `set` - Rapid lookup, no duplicates allowed.
- `map` - One-to-one mapping, no duplicates allowed, rapid key-based lookup.

Container adapters

- `stack` - Last-in, first-out (LIFO).
- `queue` - First-in, first-out (FIFO).

Iterators

Iterators are objects that point to elements within a container (arrays, vectors, lists, sets, maps, etc.) in the C++ STL. They allow you to traverse and manipulate the elements in these containers without needing to know the underlying structure of the container.

Types of iterators:

- Input Iterators
- Output Iterators
- Forward Iterators
- Bidirectional Iterators
- Random Access Iterators

Iterators supported by containers

Sequence containers

- `vector` - random access
- `deque` - random access

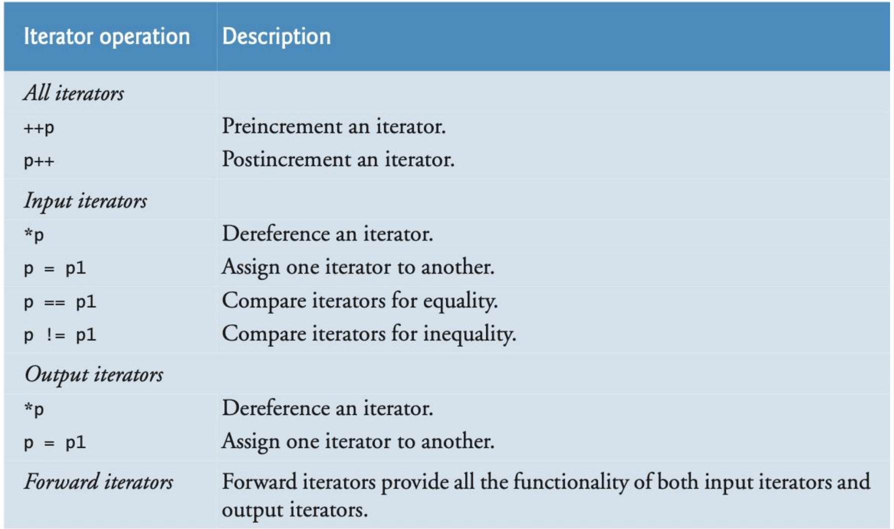
Associative containers

- `set` - bidirectional
- `map` - bidirectional

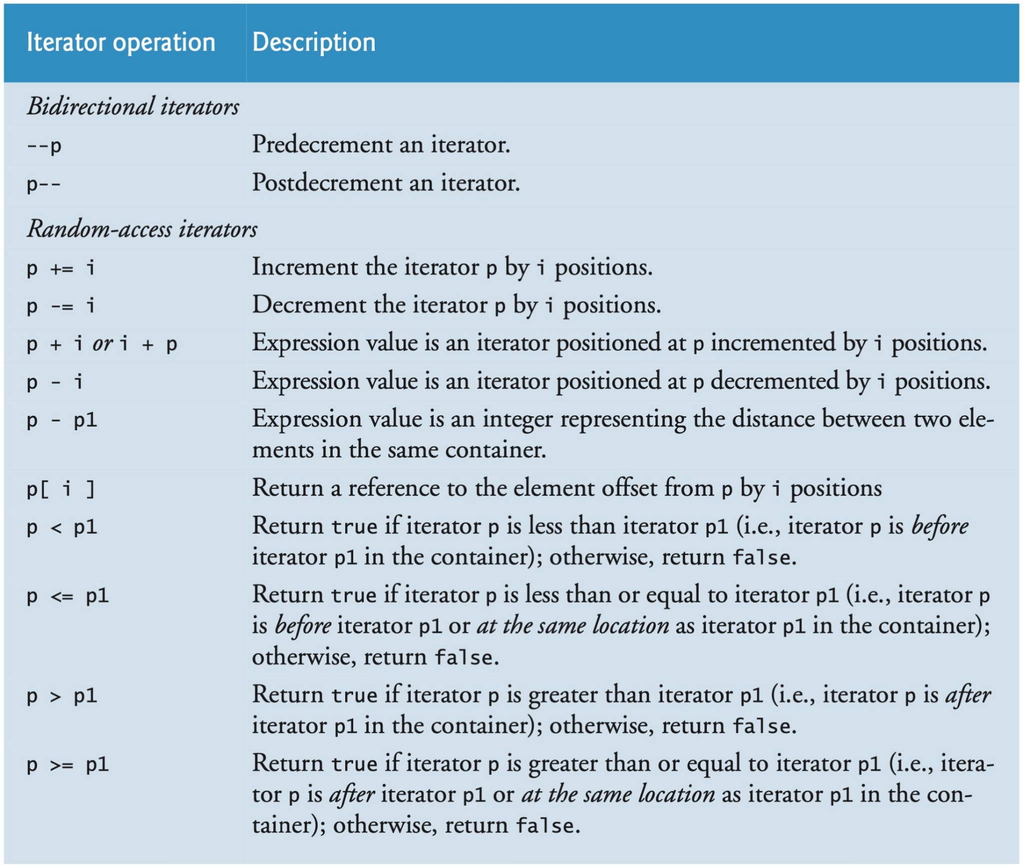
Container adapters

- `stack` - no iterators supported
- `queue` - no iterators supported

Operations, supported by iterators



Iterator operation	Description
All iterators	
`++p`	Preincrement an iterator.
`p++`	Postincrement an iterator.
Input iterators	
`*p`	Dereference an iterator.
`p = p1`	Assign one iterator to another.
`p == p1`	Compare iterators for equality.
`p != p1`	Compare iterators for inequality.
Output iterators	
`*p`	Dereference an iterator.
`p = p1`	Assign one iterator to another.
Forward iterators	Forward iterators provide all the functionality of both input iterators and output iterators.



Iterator operation	Description
Bidirectional iterators	
`--p`	Predecrement an iterator.
`p--`	Postdecrement an iterator.
Random-access iterators	
`p += i`	Increment the iterator `p` by `i` positions.
`p -= i`	Decrement the iterator `p` by `i` positions.
`p + i` *or* `i + p`	Expression value is an iterator positioned at `p` incremented by `i` positions.
`p - i`	Expression value is an iterator positioned at `p` decremented by `i` positions.
`p - p1`	Expression value is an integer representing the distance between two elements in the same container.
`p[ i ]`	Return a reference to the element offset from `p` by `i` positions
`p < p1`	Return `true` if iterator `p` is less than iterator `p1` (i.e., iterator `p` is *before* iterator `p1` in the container); otherwise, return `false`.
`p <= p1`	Return `true` if iterator `p` is less than or equal to iterator `p1` (i.e., iterator `p` is *before* iterator `p1` or *at the same location* as iterator `p1` in the container); otherwise, return `false`.
`p > p1`	Return `true` if iterator `p` is greater than iterator `p1` (i.e., iterator `p` is *after* iterator `p1` in the container); otherwise, return `false`.
`p >= p1`	Return `true` if iterator `p` is greater than or equal to iterator `p1` (i.e., iterator `p` is *after* iterator `p1` or *at the same location* as iterator `p1` in the container); otherwise, return `false`.

Vector

`vector` is an indexed sequence container that encapsulates dynamic size arrays

- requires the `<vector>` header (`#include <vector>`)
- dynamic in size
- contiguous in memory

```
vector<int> v;
v.push_back(1);
v.push_back(2);
v.push_back(3);
for(int i = 0; i < v.size(); ++i) {
    cout << v[i];
}
```

vector, ejudge reference

Input when the amount of elements is not specified

```
vector<int> v;
int temp;
while(cin >> temp) {
    v.push_back(temp);
}
```

vector, ejudge reference

2D vector examples

```
vector<vector<int> > v1; // empty 2D vector

int n;
cin >> n;
vector<vector<int> > v2(n); // 2D vector with n empty rows

int m;
cin >> m;
vector<vector<int> > v3(n, vector<int>(m)); // 2D vector with n rows and m columns
```

vector, ejudge reference

Filling an empty 2D vector

```
vector<vector<int> > v1; // empty 2D vector
int n, m;
cin >> n >> m;
for(int i = 0; i < n; ++i) {
   vector<int> row;
   for(int j = 0; j < m; ++j) {
       int temp;
       cin >> temp;
       row.push_back(temp);
   }
   v1.push_back(row);
}
```

vector, ejudge reference

Filling a 2D vector with n rows and m columns

```
vector<vector<int> > v3(n, vector<int>(m));

for(int i = 0; i < v3.size(); ++i) {
    for(int j = 0; j < v3[i].size(); ++j) {
        cin >> v3[i][j];
    }
}
```

vector, ejudge reference

Outputting a 2D vector

```
vector<vector<int> > v3(n, vector<int>(m));

for(int i = 0; i < v3.size(); ++i) {
    for(int j = 0; j < v3[i].size(); ++j) {
        cout << v3[i][j] << ' ';
    }
    cout << endl;
}
```

vector, ejudge reference

Iterating over a vector using an iterator

```
vector<int>::iterator it1;
for(it1 = v.begin(); it1 != v.end(); ++it1) {
    cout << *it1 << ' ';
}

// this also works
for(vector<int>::iterator it2 = v.begin(); it2 != v.end(); ++it2) {
    cout << *it2 << ' ';
}
```

vector, ejudge reference

Sorting a vector

- Don’t forget to include `<algorithm>` library

```
vector<int> v;

v.push_back(9);
v.push_back(1);
v.push_back(3);

sort(v.begin(), v.end());
```

vector, ejudge reference

Comparator function

- Allows to specify the order of sorting, i.e. the required order of elements
- Takes 2 values of the same type as in the container being sorted and returns **`true/false`**
- Prototype - `bool cmp(data_type val1, data_type val2);`
- If the first value is smaller than the second value, i.e. the first one should go before the second, returns **`true`**. Otherwise **`false`**.

Compare, ejudge reference

Comparator examples

- **cmp1** sorts integers in ascending order (from smaller to larger)
- **cmp2** sorts integers in descending order (from larger to smaller)

```
bool cmp1(int a, int b) {
    if(a < b) return true;
    return false;
}

bool cmp2(int a, int b) {
    return a > b;
}
```

Compare, ejudge reference

Set

`set` is an associative container that stores unique elements following a specific order

- requires the `<set>` header (`#include <set>`)
- no duplicates allowed
- always sorted
- elements are constant, i.e. cannot be changed
- the value of an element also identifies it

set, ejudge reference || set, cplusplus.com

Set example

```
set<int> s;
for(int i = 0; i < 10; i++) s.insert(i);
set<int>::iterator it;
for(it = s.begin(); it != s.end(); it++) {
      cout << *it << " ";
}
```

set, ejudge reference || set, cplusplus.com

Iterating over a set

- Example 1:

```
set<int>::iterator it;
for(it = s.begin(); it != s.end(); it++) {
    cout << *it << " ";
}
```

- Example 2:

```
set<int>::iterator it;
for(it = s.begin(); it != s.end(); it++) {
    int element = *it;
    cout << element << " ";
}
```

set, ejudge reference || set, cplusplus.com

Mistakes to avoid

- Don’t use range-based for loops, i.e. `for(int element : s)`.
  - They ***do not*** work on KBTU computers, therefore you are not able to use them during the quizzes. Use **iterators** instead.
- Don’t use `auto` keyword - it also ***does not*** work on KBTU computers. More importantly, it does not benefit your learning - you need to be aware of what types your program uses.
- Don’t use operators `<`, `>`, `<=`, `>=`, `+`, `-` with set iterators. Set iterators are of type *Bidirectional*, so instead use `==`, `!=`, `++` and `--`.
- Don’t use indexes with sets - they do not support indexation. Once again, use ***iterators***.

set, ejudge reference || set, cplusplus.com

Pair

`pair` provides a way to store two heterogeneous (different) objects as a single unit

- to create a pair, use `make_pair(val1, val2)` function
- to access the first or the second value in a pair, use `pair.first` and `pair.second`

```
pair<int, bool> p1 = make_pair(1, true);
pair<string, double> p2 = make_pair("Value of Pi:", 3.14);
cout << p1.first << " " << p1.second << endl;
cout << p2.first << " " << p2.second << endl;
```

pair, ejudge reference || pair, cplusplus.com

Map

`map` is an associative container that stores key-value pairs with unique keys, which are following a specific order

- requires `<map>` header (`#include <map>`)
- no duplicate keys allowed
- always sorted
- the value of an element is identified by its key

map, ejudge reference || map, cplusplus.com

Map example

```
map<string, int> m;
for(int i = 1; i <= 5; i++) {
    string s; cin >> s;
    m[s] = i;
}
map<string, int>::iterator it;
for(it = m.begin(); it != m.end(); it++) {
   cout << it->first << " " << it->second << endl;
}
```

map, ejudge reference || map, cplusplus.com

Iterating over a map

- Example 1:

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

- Example 2:

```
map<string, int>::iterator it;
for(it = m.begin(); it != m.end(); it++) {
   cout << (*it).first << " " << (*it).second << endl;
}
```

- Example 3:

```
map<string, int>::iterator it;
for(it = m.begin(); it != m.end(); it++) {
   pair<string, int> p = *it;
   cout << p.first << " " << p.second << endl;
}
```

map, ejudge reference || map, cplusplus.com

Mistakes to avoid

- Don’t use range-based for loops, i.e. `for(pair<int, int> element : m)`.
  - They *__do not__* work on KBTU computers, therefore you are not able to use them during the quizzes. Use **iterators** instead.
- Don’t use `auto` keyword - it also *__does not__* work on KBTU computers. More importantly, it does not benefit your learning - you need to be aware of what types your program uses.
- Don’t use operators `<, >, <=, >=, +, -` with map iterators. Map iterators are of type *Bidirectional*, so instead use `==, !=, ++` and `--`.
- Don’t forget that accessing an element with a key `key`, i.e. `m[key]`, creates an element with this key and a default value.
  - If you don’t want to create a new element if the key does not exist in a map, use `m.count(key)` or `m.find(key)` methods.
- Don’t use indexes with maps - they do not support indexation. Square brackets operator [] is used only for keys in a map, not for indexes. Once again, use ***iterators***.

map, ejudge reference || map, cplusplus.com

Vector of pairs

Compared to a map, a vector of pairs:

- Does not have keys/values - each element is a value represented by a pair
- Allows duplicates
- Preserves the same order of elements as in the input
- Allows to use indexes
  - Does not support key-based lookup
- Can be sorted
  - Does not sort elements when they're added or removed

Use this container when the aforementioned traits are desirable.

```
vector<pair<string, int> > v;
```

vector, ejudge reference

Vector of pairs example

```
vector<pair<string, int> > v;

int n;
cin >> n;
for(int i = 0; i < n; i++) {
   string s;
   int x;
   cin >> s >> x;
   pair<string, int> p = make_pair(s, x);
   v.push_back(p);
}

for(int i = 0; i < v.size(); ++i) {
   cout << v[i].first << ' ' << v[i].second << endl;
}
```

vector, ejudge reference

Stack

`stack` is a container adapter that works on the FILO (First In Last Out) principle

- requires `<stack>` header (`#include <stack>`)
- the first value added to the stack will be accessed the last
- supports only `push()` to add elements to the top and `pop()` to remove them, also from the top
- access is only possible to the top element, using `top()`

stack, ejudge reference

Stack example

```
stack<int> s;
for(int i = 1; i <= 5; i++) s.push(i);
while(!s.empty()) {
    cout << s.top() << endl;
    s.pop();
}
```

stack, ejudge reference

Queue

`queue` is a container adapter that works on the FIFO (First In First Out) principle

- requires `<queue>` header (`#include <queue>`)
- the first value added to the queue will be accessed the first
- supports only `push()` to add elements to the back and `pop()` to remove them from the front
- access only to the first element with `front()` and last element with `back()`

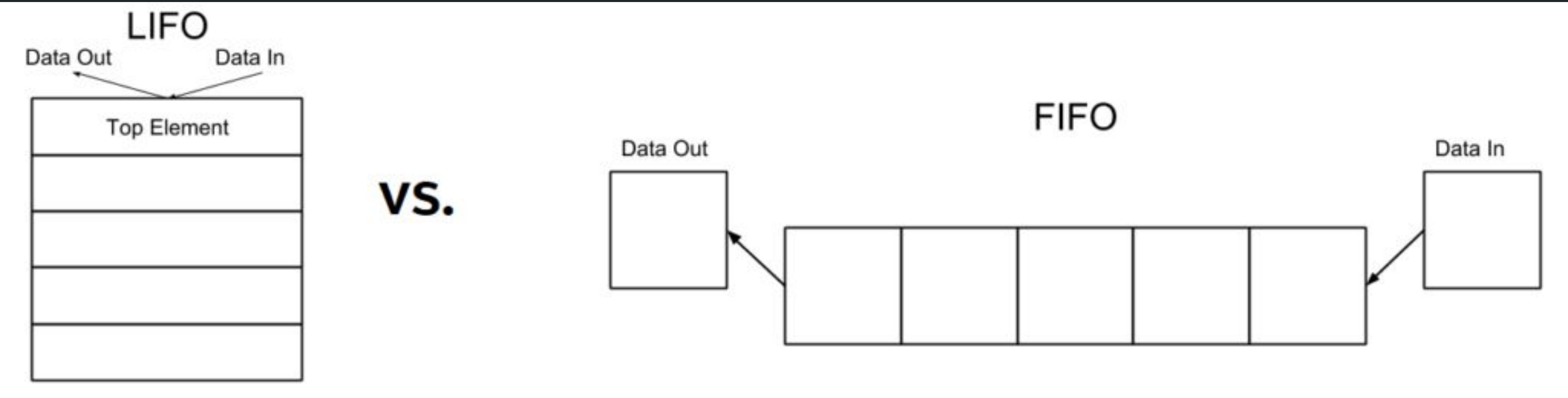
queue, ejudge reference

Queue example

```
queue<int> q;
for(int i = 1; i <= 5; ++i) q.push(i);
while(!q.empty()) {
    cout << q.front() << endl;
    q.pop();
}
```

queue, ejudge reference

LIFO vs FIFO



Deque

`deque` is an indexed sequence container allows quick insertions both to the beginning and end

- requires `<deque>` header (`#include <deque>`)
- dynamic in size
- not contiguous in memory, opposed to vector
- allows quick insertion not only at the end, but also at the start of the container

deque, ejudge reference

Deque example

```
deque<int> dq;
for(int i = 1; i <= 5; ++i) dq.push_back(i);
for(int i = 0; i < dq.size(); ++i) {
   cout << dq[i] << " ";
}
cout << endl;
```

deque, ejudge reference