# C++ Programming

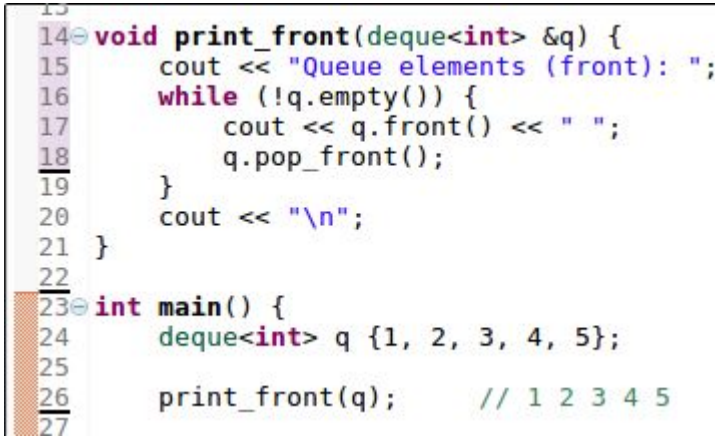
# STL Iterators

**Mostafa S. Ibrahim**
*Teaching, Training and Coaching since more than a decade!*

*Artificial Intelligence & Computer Vision Researcher*
*PhD* from Simon Fraser University - Canada
*Bachelor / Msc* from Cairo University - Egypt
Ex-(Software Engineer / ICPC World Finalist)

# Recall printing Deque

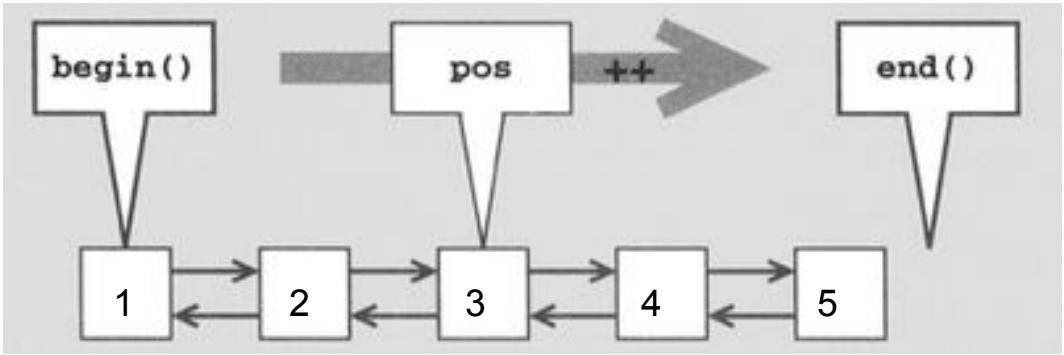
- We can print forward or backward: e.g. .front, .pop_front
- But I don’t want it to be freed?
  - Ok Don’t use &. Make a copy
  - But this is waste of time and memory?
  - Ok Use **iterators**!



```
14  void print_front(deque<int> &q) {
15      cout << "Queue elements (front): ";
16      while (!q.empty()) {
17          cout << q.front() << " ";
18          q.pop_front();
19      }
20      cout << "\n";
21  }
22
23  int main() {
24      deque<int> q {1, 2, 3, 4, 5};
25
26      print_front(q);     // 1 2 3 4 5
27
```

# Iterators

- Iterator is an object, but think of it like a *position in an array*
  - begin() = first element
  - end() = AFTER last element
- Moving forward/backward: use ++pos, --pos
- Want the current value? *pos
  - You can print / change




Img src

# Using iterators: Iterate forward

- deque<int>::iterator it
  - It is an object of type iterator
  - But specifically deque iterator
- We can deal with it with ++ and --
- *it = value
  - Called dereference operator
- q.begin() is pointing to the begin
- Are we done iterating?
  - q.end() = AFTER the content
  - So we can use that to know we are done

```
28  void print_front(deque<int> &q) {
29      cout << "Queue elements (front): ";
30
31      deque<int>::iterator it = q.begin();
32
33      while(it != q.end()) {
34          cout<<*it<<" ";
35          ++it;
36      }
37
38      cout << "\n";
39  }
40
41  int main() {
42      deque<int> q {1, 2, 3, 4, 5};
43
44      print_front(q);      // 1 2 3 4 5
45
```

# Using iterators: Iterate backward

- To iterate backward we use rbegin (r for reverse)
- Think of it as if the content is reversed
  - So u again u move with ++

```
21 void print_back1(deque<int> &q) {
22     cout << "Queue elements (back): ";
23
24     // reverse iterator
25     deque<int>::reverse_iterator it = q.rbegin();
26
27     while(it != q.rend()) {
28         cout<<*it<<" ";
29         ++it;
30     }
31     cout << "\n";
32 }
33
34 void print_back2(deque<int> &q) {
35     cout << "Queue elements (back): ";
36
37     for(auto it = q.rbegin(); it != q.rend(); ++it)
38         cout<<*it<<" ";
39     cout << "\n";
40 }
41
```

# Iterator arithmetic

```
55 void lets_play() {
56     deque<int> q {1, 2, 3, 4, 5};
57
58     auto it = q.begin() + 3;     // FORTH element position
59     cout<<*it<<"\n";         // 4
60
61     cout<<*(it--)<<"\n";     // 4 then move to 3rd position
62     cout<<*it<<"\n";         // 3
63
64     cout<<*(--it)<<"\n";     // 2
65     cout<<*it<<"\n";         // 2 Now on 2nd
66
67     cout<<*(it + 3)<<"\n";   // 5th position
68     it += 3;
69     cout<<*it<<"\n";         // 2 Now on 5th position
70
71
72     // reset all to 10
73     for(auto it = q.begin(); it != q.end(); ++it)
74         *it = 10;
75 }
76
```

# Const iterator

- cbegin/cend instead of begin/end
- The same as we did, just iterator is const,
- But you can’t change value while iterating
  - Good for communicating intentions
- Similarly: crbegin, crend

```
55 void print_front_const(deque<int> &q) {
56     cout << "Queue elements (front): ";
57
58     deque<int>::const_iterator it = q.cbegin();
59
60     while(it != q.cend()) {
61         cout<<*it<<" ";
62         //*it = 10;      // can't - CONST
63         ++it;
64     }
65
66     cout << "\n";
67 }
```

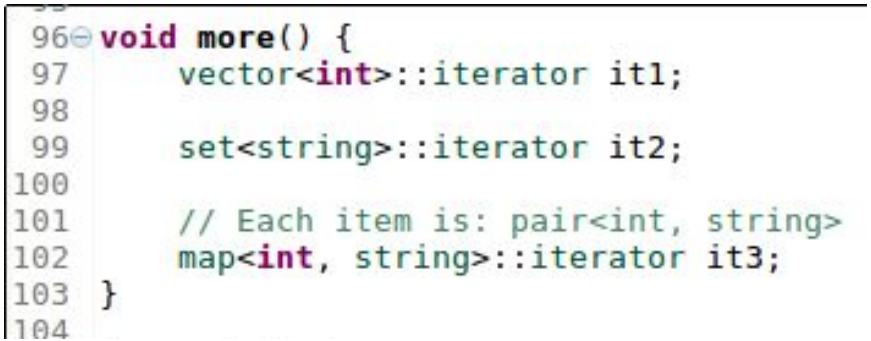
# Many iterators

- You can actually iterate on most data structures
  - can’t on stack, queue, priority_queue
- So let’s iterate on string
  - E.g. string::iterator it;

```
42 int count_lowers(const string & str)
43 {
44     int cnt = 0;
45     for(auto it = str.begin(); it != str.end(); ++it)
46     {
47         char ch = *it;
48         cnt += (islower(ch) > 0);
49     }
50     return cnt;
51 }
```

# Next

- In next videos, we will learn other data structures
  - They also support iterators!
- You will see **operations** by these containers
  - E.g. erase element using iterator or insert element/group of elements
  - Algorithms that takes iterator begin/end to e.g. search



"Acquire knowledge and impart it to the people."

"Seek knowledge from the Cradle to the Grave."