



CS 225

Data Structures

September 23 – Iterators

G Carl Evans



CS 225 So Far...

List ADT

- Linked Memory Implementation (“Linked List”)
 - $O(1)$ insert/remove at front/back
 - $O(1)$ insert/remove after a given element
 - $O(n)$ lookup by index
- Array Implementation (“Array List”)
 - $O(1)$ insert/remove at front/back
 - $O(n)$ insert/remove at any other location
 - $O(1)$ lookup by index



CS 225 So Far...

Queue ADT

- [Order]:

- [Implementation]:

- [Runtime]:



CS 225 So Far...

Stack ADT

- [Order]:

- [Implementation]:

- [Runtime]:

Queue.h

```
1 #pragma once
2
3 template <typename T>
4 class Queue {
5     public:
6         void enqueue(T e);
7         T dequeue();
8         bool isEmpty();
9
10    private:
11        T *items_;
12        unsigned capacity_;
13        unsigned count_;
14 };
15
16
17
18
19
20
21
22
```

What type of implementation is this Queue?

How is the data stored on this Queue?

Queue.h

```
1 #pragma once
2
3 template <typename T>
4 class Queue {
5     public:
6         void enqueue(T e);
7         T dequeue();
8         bool isEmpty();
9
10    private:
11        T *items_;
12        unsigned capacity_;
13        unsigned count_;
14 };
15
16
17
18
19
20
21
22
```

What type of implementation is this Queue?

How is the data stored on this Queue?



```
Queue<int> q;
q.enqueue(3);
q.enqueue(8);
q.enqueue(4);
q.dequeue();
q.enqueue(7);
q.dequeue();
q.dequeue();
q.enqueue(2);
q.enqueue(1);
q.enqueue(3);
q.enqueue(5);
q.dequeue();
q.enqueue(9);
```

Queue.h

```
1 #pragma once
2
3 template <typename T>
4 class Queue {
5     public:
6         void enqueue(T e);
7         T dequeue();
8         bool isEmpty();
9
10    private:
11        T *items_;
12        unsigned capacity_;
13        unsigned count_;
14 };
15
16
17
18
19
20
21
22
```



Queue<char> q;

...

q.enqueue(m);

q.enqueue(o);

q.enqueue(n);

...

q.enqueue(d);

q.enqueue(a);

q.enqueue(y);

q.enqueue(i);

q.enqueue(s);

q.dequeue();

q.enqueue(h);

q.enqueue(a);

Implications of Design

1.

```
class ListNode {  
    public:  
        T & data;  
        ListNode * next;  
        ...  
};
```

2.

```
class ListNode {  
    public:  
        T * data;    ...  
};
```

3.

```
class ListNode {  
    public:  
        T data;    ...  
};
```


Implications of Design

| | Storage by Reference | Storage by Pointer | Storage by Value |
|---|----------------------|--------------------|------------------|
| Who manages the lifecycle of the data? | | | |
| Is it possible for the data structure to store NULL? | | | |
| If the data is manipulated by user code while in our data structure, is the change reflected in our data structure? | | | |
| Speed | | | |

Data Lifecycle

Storage by reference:

```
1 Cube c;  
2 myStack.push(c);
```

Storage by pointer:

```
1 Cube c;  
2 myStack.push(&c);
```

Storage by value:

```
1 Cube c;  
2 myStack.push(c);
```

Data Modifications

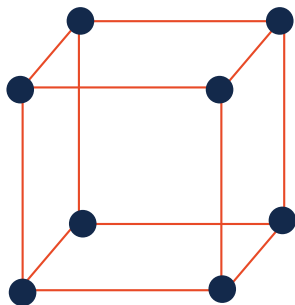
```
1 Cube c(1);  
2 myStack.push(c);  
3  
4 c.setLength(42);  
5  
6 Cube r = myStack.pop();  
7 // What is r's length?
```



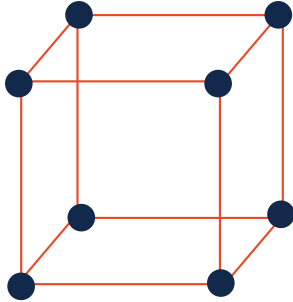
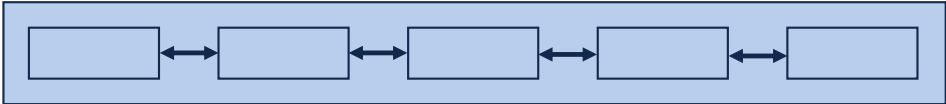
Speed

Iterators

Suppose we want to look through every element in our data structure:



Iterators encapsulated access to our data:



| Cur. Location | Cur. Data | Next |
|---------------|-----------|------|
| | | |
| | | |
| | | |



Iterators

Every class that implements an iterator has two pieces:

1. [Implementing Class]:



Iterators

Every class that implements an iterator has two pieces:

2. [Implementing Class' Iterator]:

- Must have the base class **std::iterator**
- Must implement
 - operator*
 - operator++
 - operator!=

stlList.cpp

```
1 #include <list>
2 #include <string>
3 #include <iostream>
4
5 struct Animal {
6     std::string name, food;
7     bool big;
8     Animal(std::string name = "blob", std::string food = "you", bool big = true) :
9         name(name), food(food), big(big) { /* nothing */ }
10 };
11
12 int main() {
13     Animal g("giraffe", "leaves", true), p("penguin", "fish", false), b("bear");
14     std::vector<Animal> zoo;
15
16     zoo.push_back(g);
17     zoo.push_back(p); // std::vector's insertAtEnd
18     zoo.push_back(b);
19
20     for ( std::vector<Animal>::iterator it = zoo.begin(); it != zoo.end(); it++ ) {
21         std::cout << (*it).name << " " << (*it).food << std::endl;
22     }
23
24     return 0;
25 }
```

stlList.cpp

```
1 #include <list>
2 #include <string>
3 #include <iostream>
4
5 struct Animal {
6     std::string name, food;
7     bool big;
8     Animal(std::string name = "blob", std::string food = "you", bool big = true) :
9         name(name), food(food), big(big) { /* none */ }
10 };
11
12 int main() {
13     Animal g("giraffe", "leaves", true), p("penguin", "fish", false), b("bear");
14     std::vector<Animal> zoo;
15
16     zoo.push_back(g);
17     zoo.push_back(p); // std::vector's insertAtEnd
18     zoo.push_back(b);
19
20     for ( const Animal & animal : zoo ) {
21         std::cout << animal.name << " " << animal.food << std::endl;
22     }
23
24     return 0;
25 }
```