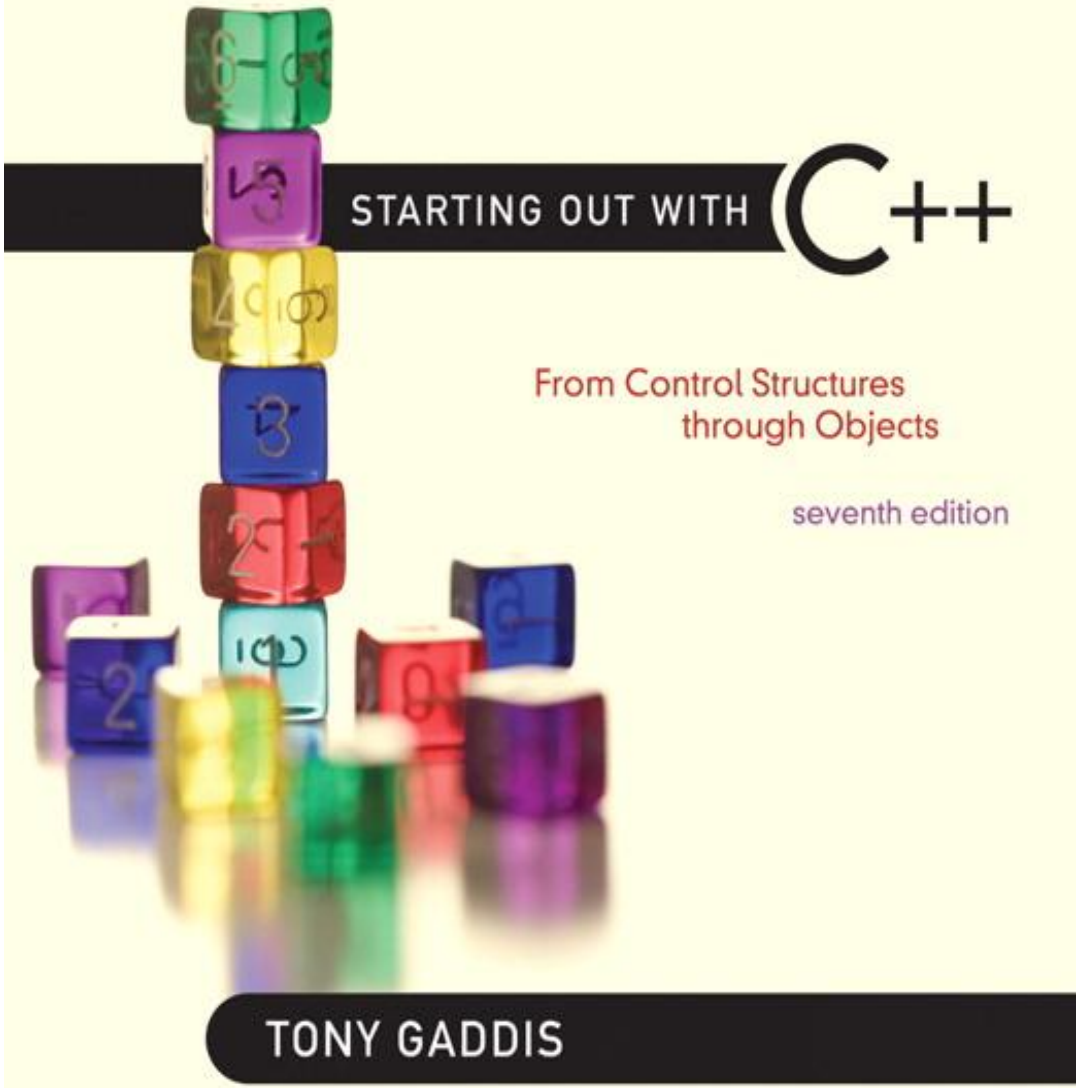
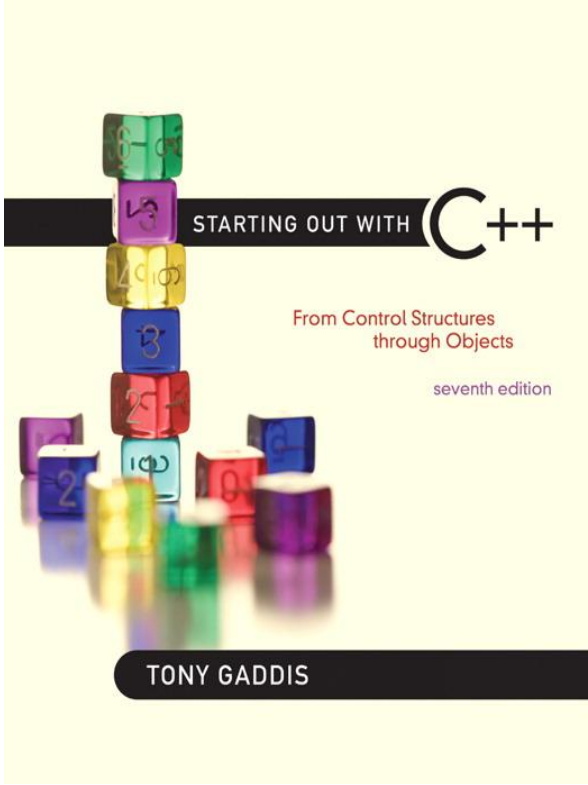
Chapter 18:

Stacks And Queues



Addison-Wesley is an imprint of PEARSON



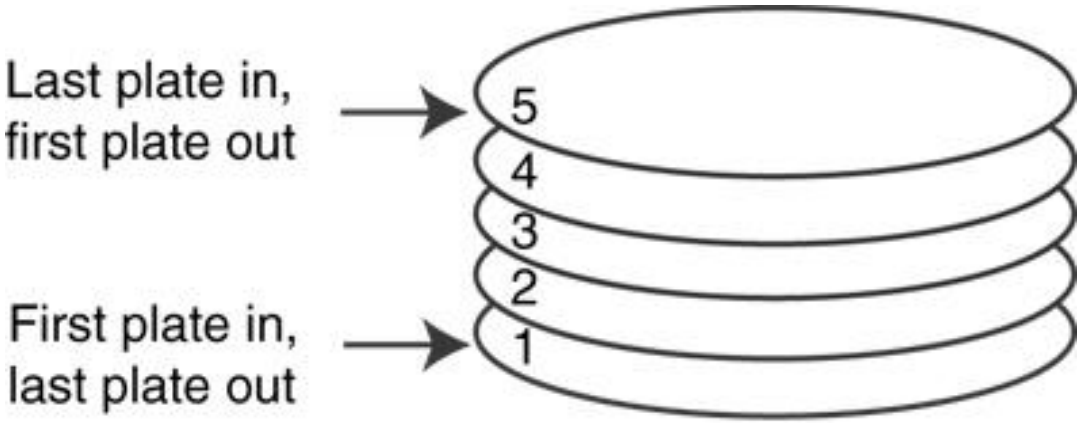
18.1

Introduction to the Stack ADT

Introduction to the Stack ADT

- Stack
  - a LIFO (last in, first out) data structure
- Examples
  - plates in a cafeteria
  - return addresses for function calls
- Implementation
  - static
    - fixed size, implemented as an array
  - dynamic
    - variable size, implemented as a linked list

A LIFO Structure

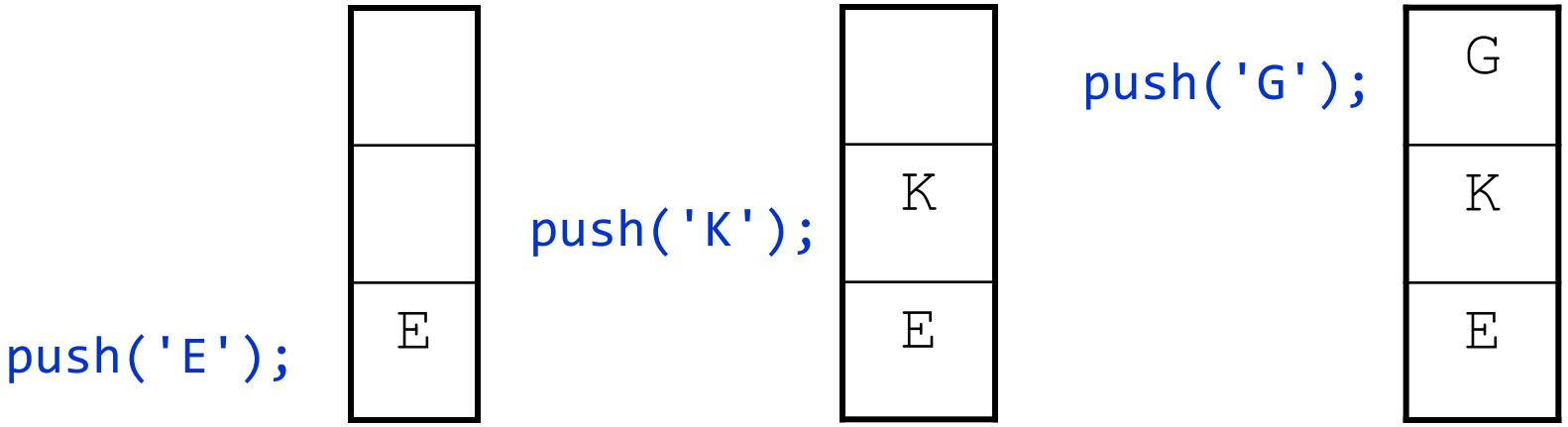


Stack Operations and Functions

- Operations
  - push add a value onto the top of the stack
  - pop remove a value from the top of the stack
- Functions
  - isFull
    - `true` if the stack is currently full, *i.e.*, has no more space to hold additional elements
  - isEmpty
    - `true` if the stack currently contains no elements

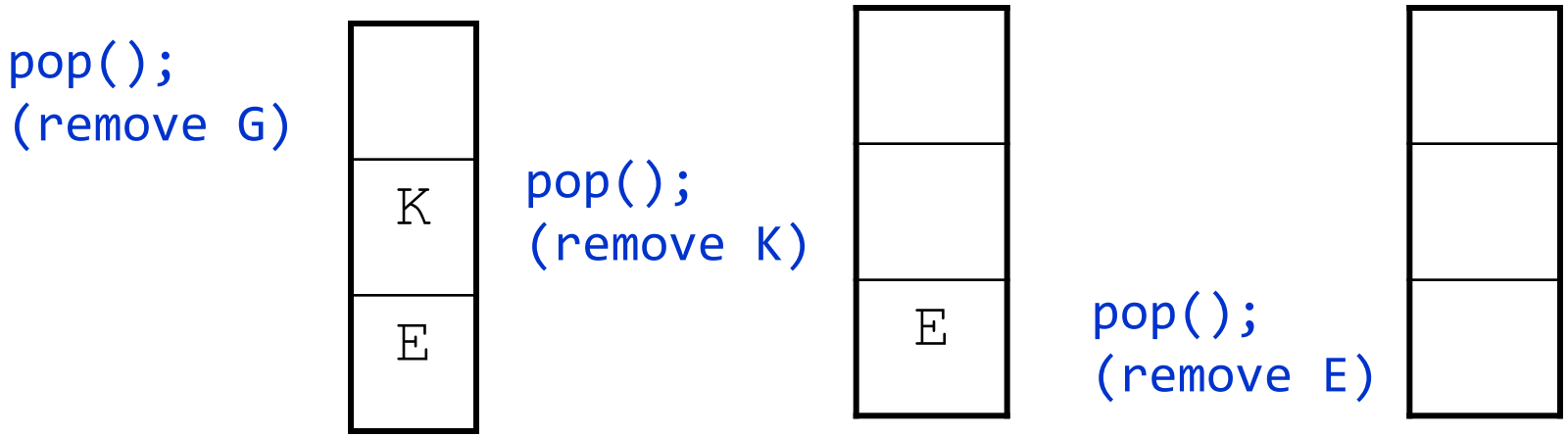
Stack Operations – Example

- A stack that can hold `char` values:



Stack Operations – Example

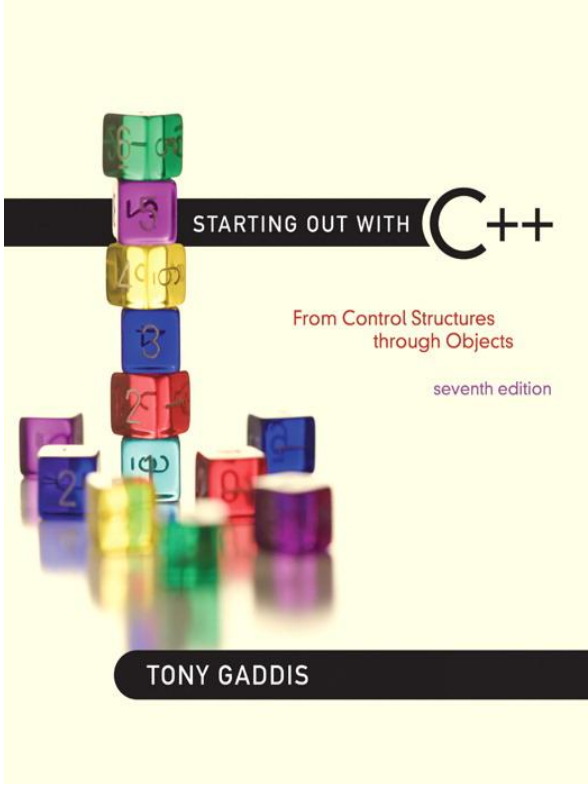
- A stack that can hold `char` values:



Contents of `IntStack.h`

```
1  // Specification file for the IntStack class
2  #ifndef INTSTACK_H
3  #define INTSTACK_H
4
5  class IntStack
6  {
7  private:
8     int *stackArray;  // Pointer to the stack array
9     int stackSize;    // The stack size
10    int top;          // Indicates the top of the stack
11
12 public:
13    // Constructor
14    IntStack(int);
15
16    // Copy constructor
17    IntStack(const IntStack &);
18
19    // Destructor
20    ~IntStack();
21
22    // Stack operations
23    void push(int);
24    void pop(int &);
25    bool isFull() const;
26    bool isEmpty() const;
27 };
28 #endif
```

(See IntStack.cpp for the implementation.)



18.2

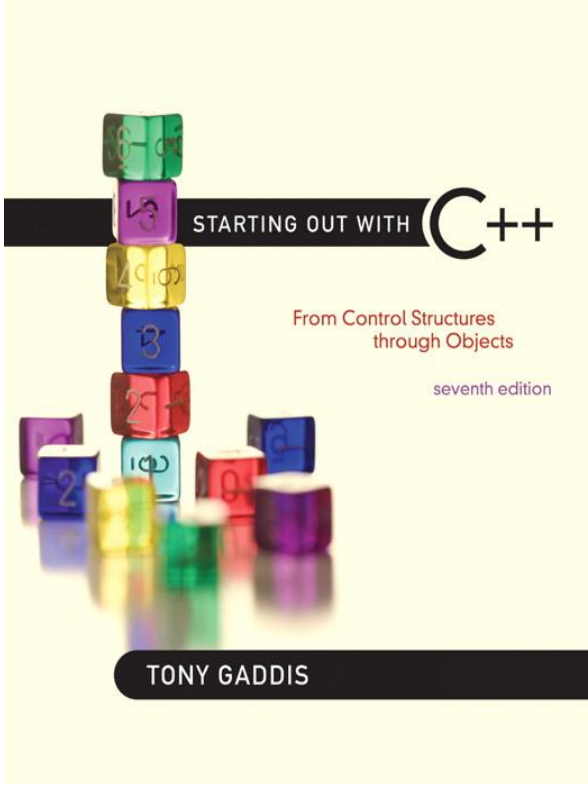
Dynamic Stacks

Dynamic Stacks

- Grow and shrink as necessary
- Can't ever be full as long as memory is available
- Implemented as a linked list

Implementing a Stack

- Programmers can program their own routines to implement stack functions
- See `DynIntStack` class in the book for an example.
- Can also use the implementation of stack available in the STL



18.3

The STL `stack` Container

The STL `stack` container

- Stack template can be implemented as a
  - vector, linked list, or deque
- Implements member functions:
  - push / pop
  - empty
  - size: number of elements on the stack
  - top: reference to element on top of the stack

Defining a `stack`

- Defining a stack of `char` named `cstack`
  - implemented using a **`vector`**:

    ```
    stack< char, vector<char> > cstack;
    ```

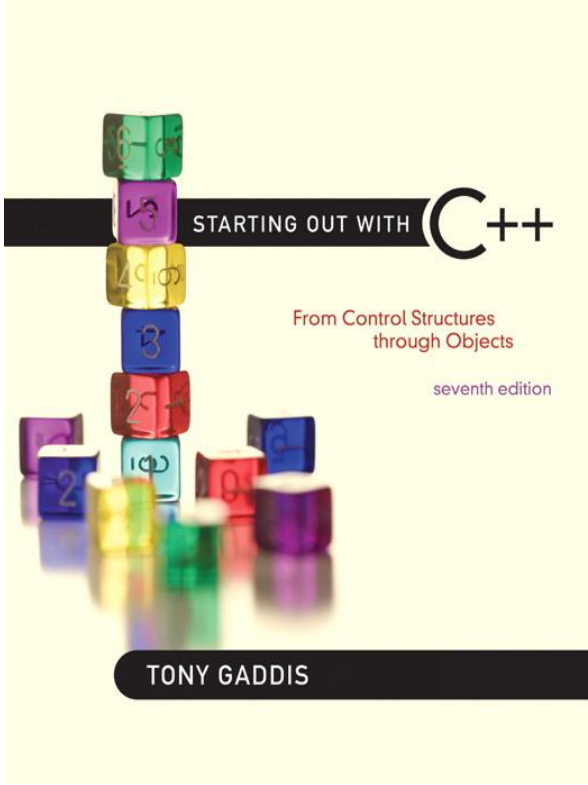
  - implemented using a **`list`**:

    ```
    stack< char, list<char> > cstack;
    ```

  - implemented using a **`deque`**:

    ```
    stack<char> cstack; //default implementation uses a deque
    ```

- Spaces are required between consecutive `>>` and `<<` symbols



18.4

Introduction to the Queue ADT

Introduction to the Queue ADT

- Queue
  - a FIFO (first in, first out) data structure.
  - commonly used in computer operating systems
- Examples
  - people in line at the theatre box office
  - print jobs sent to a printer
- Implementation
  - static: fixed size, implemented as an array
  - dynamic: variable size, implemented as a linked list

Queue Locations and Operations

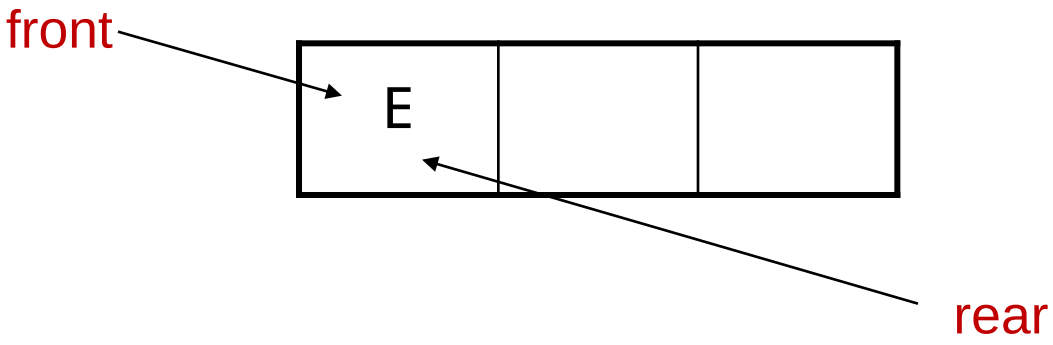
- rear
  - position where elements are added
- front
  - position from which elements are removed

- enqueue
  - add an element to the rear of the queue
- dequeue
  - remove an element from the front of a queue

Queue Operations - Example

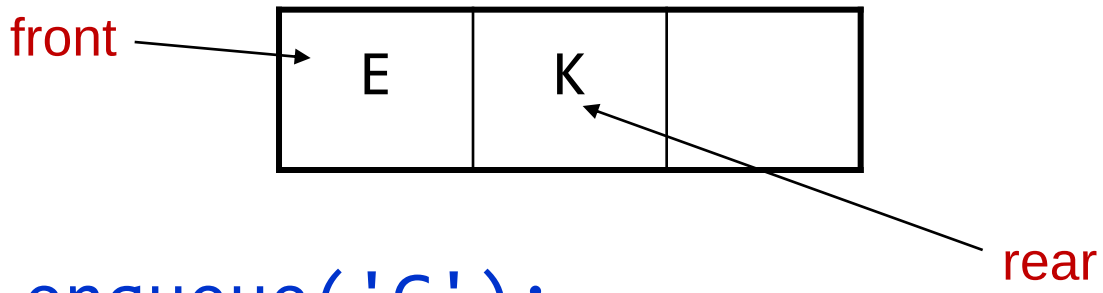
- A currently empty queue that can hold `char` values:

- `enqueue('E');`

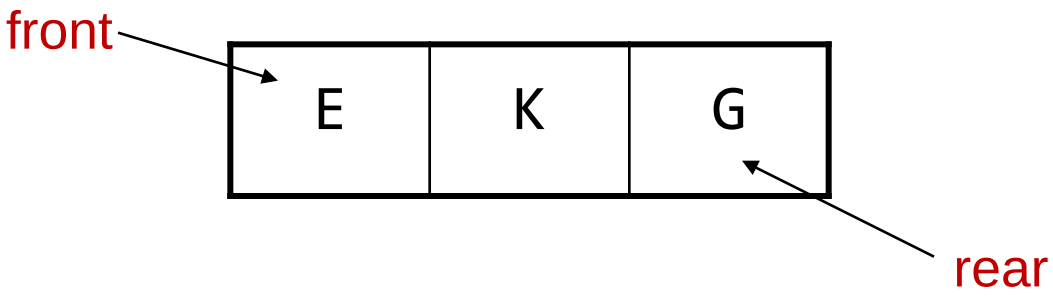


Queue Operations - Example

- `enqueue('K');`

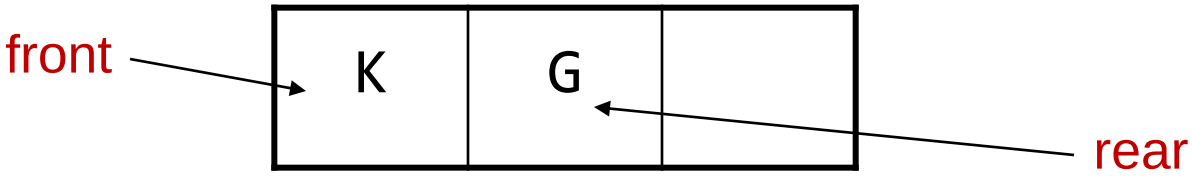


- `enqueue('G');`

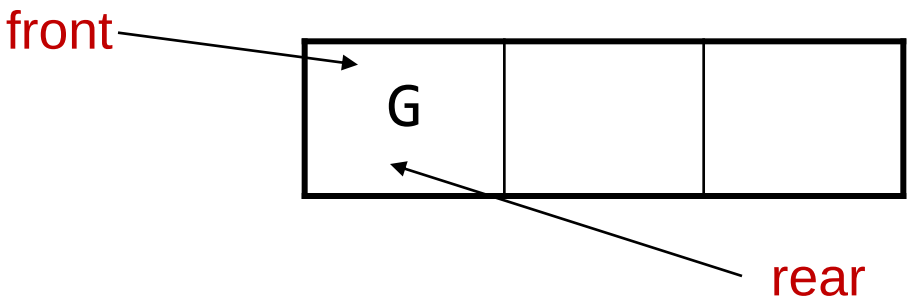


Queue Operations - Example

- `dequeue(); // remove E`



- `dequeue(); // remove K`



dequeue Issue, Solutions

- Issue:
  - When removing an element from a queue, remaining elements must shift to front
- Solutions:
  - Let front index move as elements are removed (works as long as rear index is not at end of array)
  - Use above solution, and also let rear index "wrap around" to front of array, treating array as circular instead of linear (more complex enqueue, dequeue code)

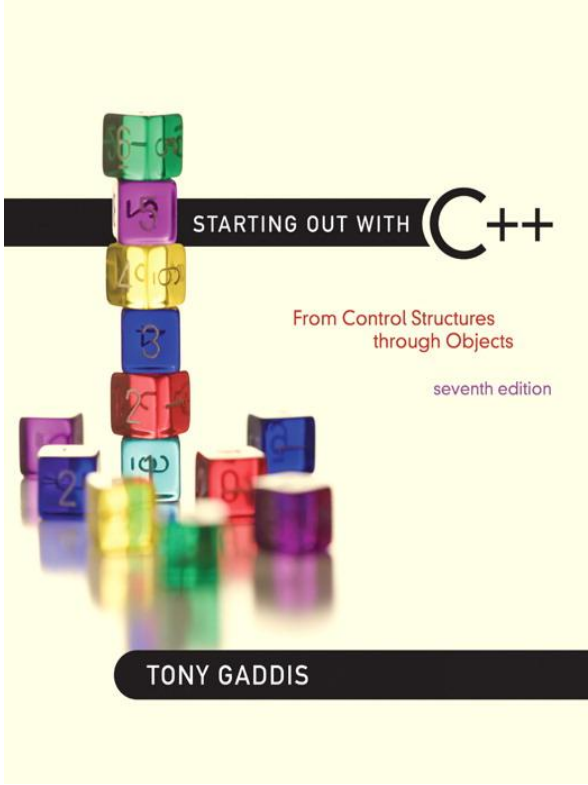
Contents of IntQueue.h

```
 1  // Specification file for the IntQueue class
 2  #ifndef INTQUEUE_H
 3  #define INTQUEUE_H
 4
 5  class IntQueue
 6  {
 7  private:
 8     int *queueArray;  // Points to the queue array
 9     int queueSize;    // The queue size
10     int front;        // Subscript of the queue front
11     int rear;         // Subscript of the queue rear
12     int numItems;     // Number of items in the queue
```

Contents of IntQueue.h (Continued)

```
13  public:
14     // Constructor
15     IntQueue(int);
16
17     // Copy constructor
18     IntQueue(const IntQueue &);
19
20     // Destructor
21     ~IntQueue();
22
23     // Queue operations
24     void enqueue(int);
25     void dequeue(int &);
26     bool isEmpty() const;
27     bool isFull() const;
28     void clear();
29  };
30  #endif
```

(See IntQueue.cpp for the implementation)

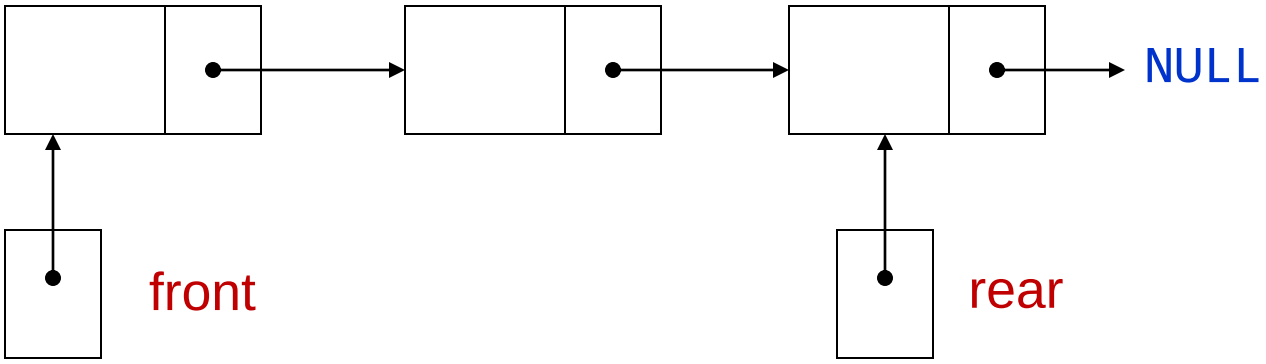


18.5

Dynamic Queues

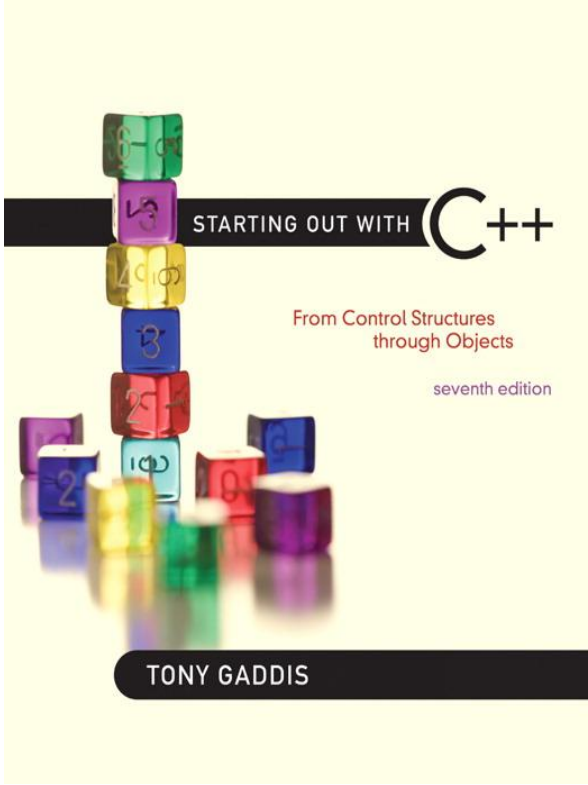
Dynamic Queues

- Like a stack, a queue can be implemented using a linked list
- Allows dynamic sizing, avoids issue of shifting elements or wrapping indices



Implementing a Queue

- Programmers can program their own routines to implement queue operations
- See the `DynIntQue` class in the book for an example of a dynamic queue
- Can also use the implementation of queue and dequeue available in the STL



18.6

The STL deque and queue Containers

The STL deque and queue Containers

Two containers that permit queue-like operations

- deque (double-ended queue. A deque is a sequence container):
  - similar to a vector except that items can inserted and removed from the front or the rear of the container.
  - Allows use of **[ ]** (linear sequence)
  - Member functions:
    - enqueue `push_front, push_back`
    - dequeue `pop_front, pop_back`
    - get value `front, back`

- queue (a FIFO container adapter - stack is a LIFO container adapter)
  - a queue that can be implemented using a `list`, or `deque`. Member functions:
    - enque `push` //always inserts at rear of queue
    - dequeue `pop` //always removes from front of queue
    - get value `front, back`

Defining a **queue**

- Defining a queue of `char` named `cQueue`
  - implemented using a deque:

```
queue< char, deque<char> > cQueue;
```

  - implemented using a list:

```
queue< char, list<char>  > cQueue;
```

- Spaces are required between consecutive `>>` and `<<` symbols

Defining a **deque**

- Defining a deque of `char` named `cDeque`

```
deque<char> cDeque;
```