Chapter 18:

Stacks And Queues
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 array
  - dynamic: variable size, implemented as linked list
A LIFO Structure

Last plate in, first plate out

First plate in, last plate out
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
A stack that can hold char values:

push('E');

push('K');

push('G');
Stack Operations - Example

- A stack that can hold `char` values:

```
pop();  // remove G

pop();  // remove K

pop();  // remove E
```
Contents of `IntStack.h`

```cpp
// Specification file for the IntStack class
#ifndef INTSTACK_H
#define INTSTACK_H

class IntStack
{
  private:
    int *stackArray; // Pointer to the stack array
    int stackSize;  // The stack size
    int top;        // Indicates the top of the stack

  public:
    // Constructor
    IntStack(int);

    // Copy constructor
    IntStack(const IntStack &);

    // Destructor
    ~IntStack();

    // Stack operations
    void push(int);
    void pop(int &);
    bool isFull() const;
    bool isEmpty() const;
};
#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
The STL stack container
The STL stack container

- Stack template can be implemented as a **vector**, a **linked list**, or a **deque**
- Implements **push**, **pop**, and **empty** member functions
- Implements other member functions:
  - **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:
  \[
  \text{stack}< \text{char}, \text{vector}<\text{char}> > \text{cstack};
  \]
- implemented using a list:
  \[
  \text{stack}< \text{char}, \text{list}<\text{char}> > \text{cstack};
  \]
- implemented using a deque:
  \[
  \text{stack}< \text{char} > \text{cstack};
  \]
- Spaces are required between consecutive >>, << symbols
18.4

Introduction to the Queue ADT
Introduction to the Queue ADT

- **Queue**: a FIFO (first in, first out) data structure.
- **Examples**:
  - people in line at the theatre box office
  - print jobs sent to a printer
- **Implementation**:
  - static: fixed size, implemented as array
  - dynamic: variable size, implemented as 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');

Copyright © 2009 Pearson Education, Inc. Publishing as Pearson Addison-Wesley
Queue Operations - Example

• `enqueue ('K');`

```
   front
   E  K
```

• `enqueue ('G');`

```
   front
   E  K  G
```
Queue Operations - Example

• dequeue(); // remove E

<Diagram>

• dequeue(); // remove K

<Diagram>
**dequeue Issue, Solutions**

- 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)

```cpp
public:
// Constructor
IntQueue(int);

// Copy constructor
IntQueue(const IntQueue &);

// Destructor
~IntQueue();

// Queue operations
void enqueue(int);
void dequeue(int &);
bool isEmpty() const;
bool isFull() const;
void clear();
```

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

- **deque**: a double-ended queue. Has member functions to enqueue (push_back) and dequeue (pop_front).
- **queue**: container ADT that can be used to provide queue as a vector, list, or deque. Has member functions to enqueue (push) and dequeue (pop).
• Defining a queue of `chars`, named `cQueue`, implemented using a deque:
  ```cpp
deque<char> cQueue;
```  
• implemented using a queue:
  ```cpp
queue<char> cQueue;
```  
• implemented using a `list`:
  ```cpp
queue< char, list<char> > cQueue;
```  
• Spaces are required between consecutive `>>`, `<<` symbols