# Chapter 3 Stacks and Queues

# 3.1 Templates in C++

## introduce the concept of templates

### make classes and functions more reusable

# 3.1.1 Templates Functions

## Program ~~1.3~~ 1.6: sorts an array of integers using the selection sort method

### Prog 1

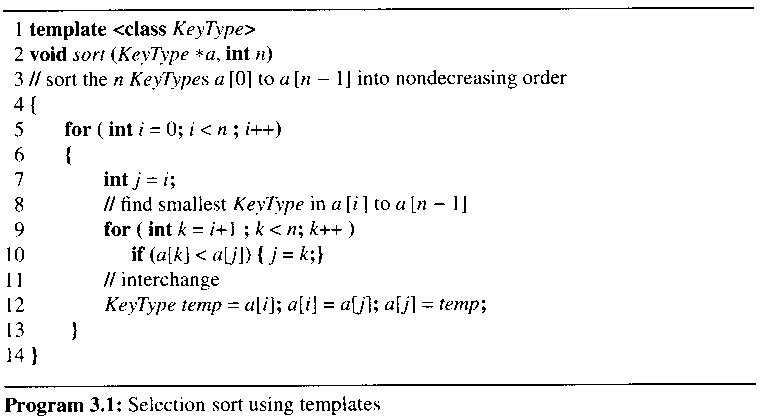
## wish to use selection sort to sort an array of floating point numbers

### achieve this by changing the program 1.6

## templates

### parameterized types

### a variable that can be instantiated to any data type



### the copy constructor in line 12

#### distinguish an initialization from an assignment by the presence of a type specifier

#### KeyType temp = a[i];

##### the constructor function may be overloaded or default implementation

##### define temp to be a variable of type KeyType and initialize it to a[i]

##### invoked when an object is initialized with another object

#### prog 3

# 3.1.2 Using Templates to Represent Container Classes

## a container class

### a class that represents a data structure that contains or stores a number of data objects

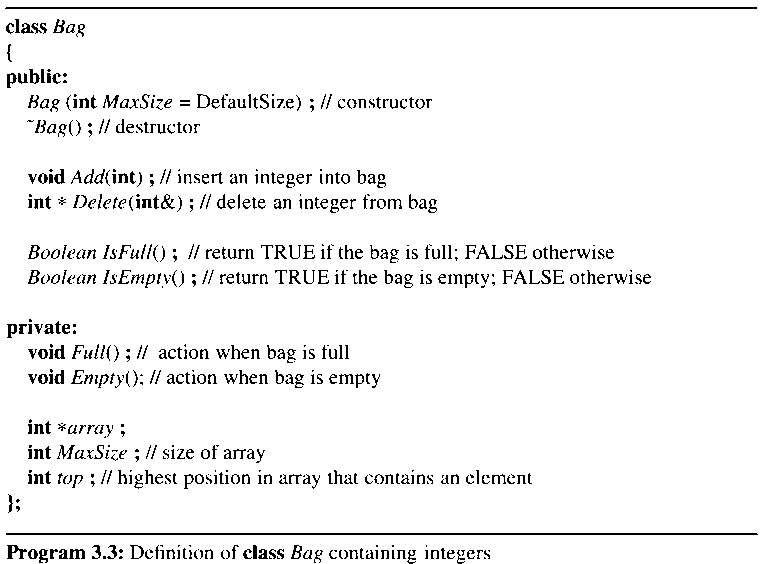
### examples of a container class : array, bag

### bag

#### have multiple occurrences of the same element

#### don't care about the position of an element

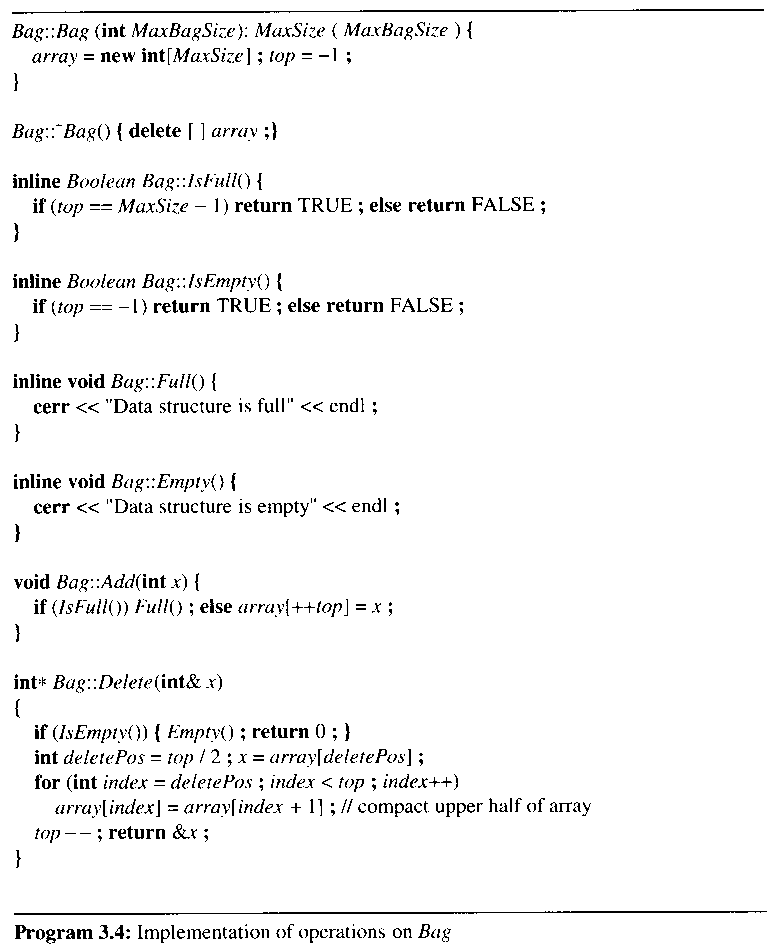
#### don't care which element is removed when a delete operation is performed



#### a data structure into which objects can be inserted and from which objects can be deleted

#### bag의 behaviors를 어떻게 구현하나?

## the implementation of operations of Bag



#### insert: store the element into the first available position in the array

#### delete: delete the element in the middle position of the array

##### midddle이 아닌 임의 장소의 객체를 삭제하는 것으로 구현 가능

## member function Delete returns a pointer to an integer

### suppose Delete returns the integer

#### if the bag is empty, then return (-1) to signal to the calling function that the bag is empty

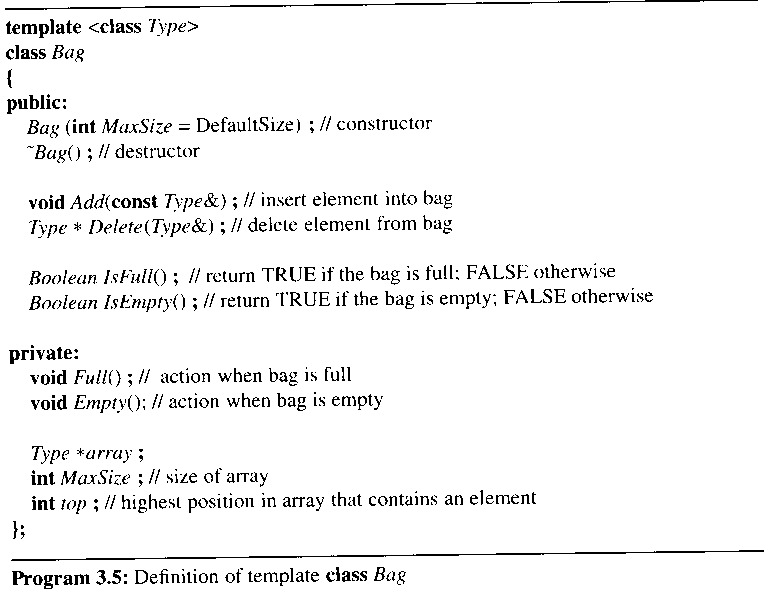
#### how does the calling function know whether return (-1) or empty bag

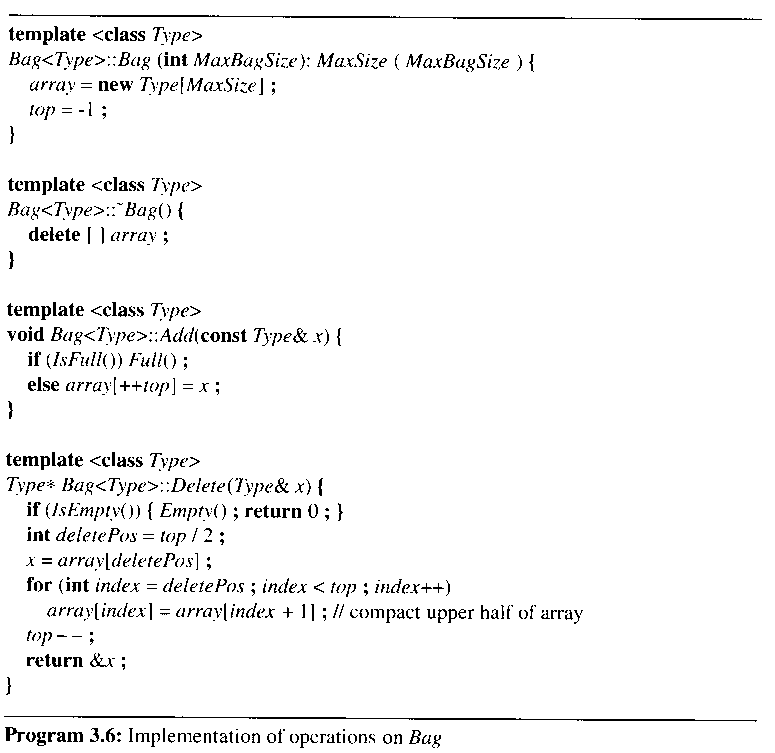
### the Delete ( ) function returns a pointer to an integer that is deleted from the bag

#### in the bag example, return a pointer to the appropriate array element

## to implement Bag using templates to store objects of any data type

### container classes are suitable for implementation using templates





### template <class Type>

### Bag <int> a;

### Bag <Rectangle> r;

# 3.2 The Stack Abstract Data Type

## A = a0, a1, …, an-1 is an ordered list of n >= 0 elements

## A stack is an ordered list in which insertions and deletions are made at one end

### fig 3

#### LIFO

#### An element of the stack describes an activation record or a stack frame

##### Local variables, previous frame pointer, return address

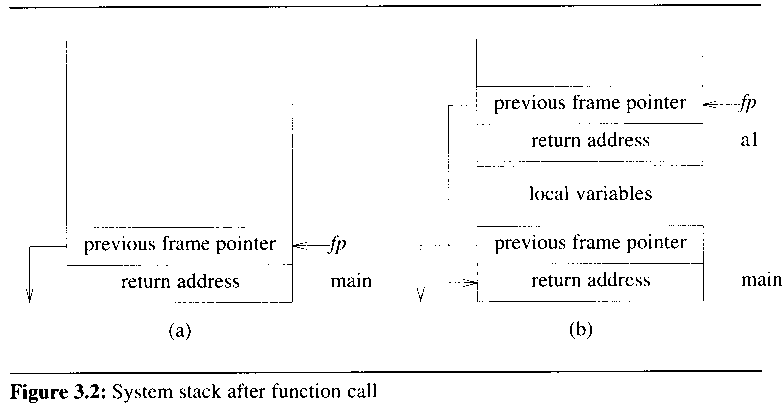
# Example 3.1 [system stack]

## place an activation record or a stack frame on top of the system stack

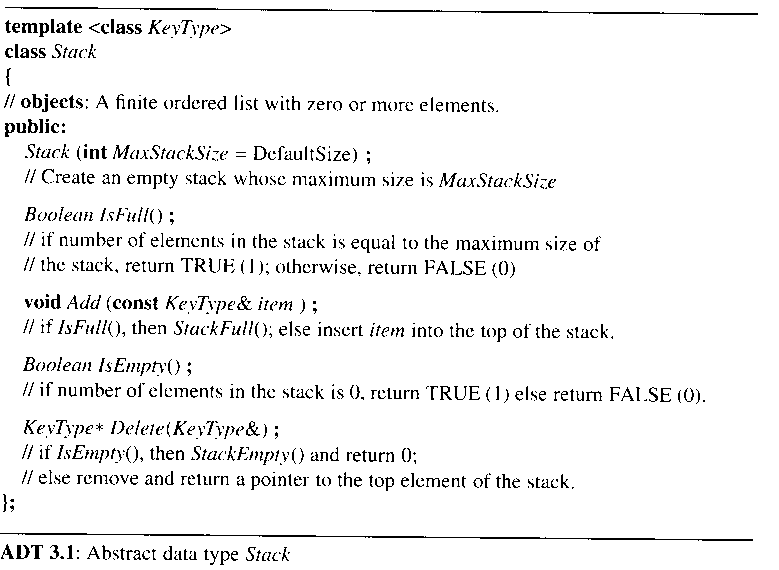
### the activation record for the invoked function

#### contain a pointer to the previous stack frame and a return address

#### on top of the system stack : one function executed at any given time



## ADT specification of stack



### a way to implement ADT

#### to use a one-dimensional array, stack[MaxSize]

### data member declarations and constructor definition of stack

#### private:

#### int top;

#### KeyType \*stack;

#### int MaxSize;

#### template <class KeyType>

#### // a member initialization list

#### stack<KeyType>::stack (int MaxStackSize): MaxSize(MaxStackSize)

#### {

#### stack = new KeyType[MaxSize];

#### top = -1;

#### }

### the member functions IsFull() and IsEmpty()

#### template <class KeyType>

#### inline Boolean stack<KeyType>::IsFull()

#### {

#### if (top == MaxSize -1 ) return TRUE;

#### else return FALSE;

#### }

#### template <class KeyType>

#### inline Boolean stack<KeyType>::IsEmpty()

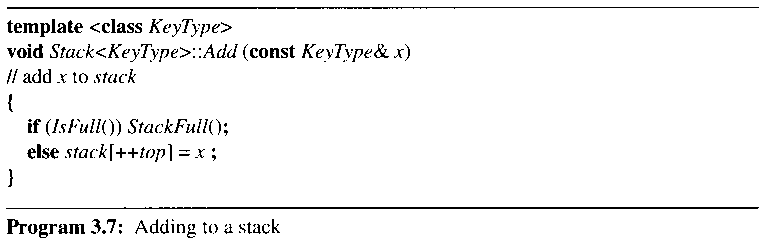
#### {

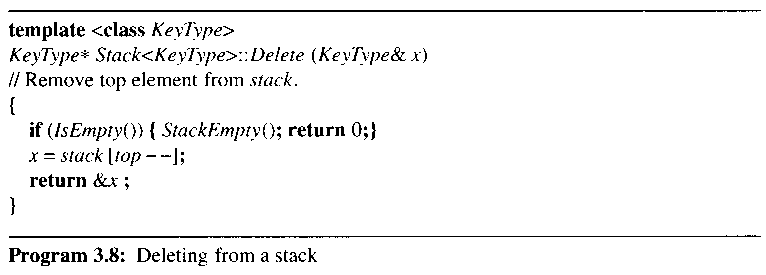
#### if (top == -1) return TRUE;

#### else return FALSE;

#### }

### the Add and Delete operations



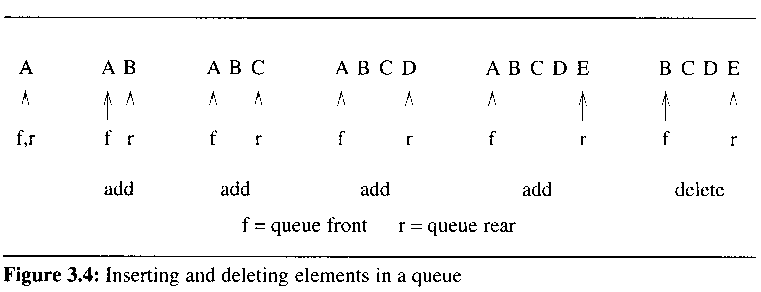


* ADT Bag, ADT Stack은 별 차이가 없다. 그리고 representation도 array 를 사용하고 top 변수를 쓰는 것은 같다. Bag와 Stack의 성질 차이는 어떻게 구현하나?
* ADT 타입이 bag와 stack의 semantics을 imply하는 것으로 생각함
* int, float 타입을 사용하는 것과 같음, boolean type에 대한 and, or 연산을 사용하지 곱셈 연산을 사용하지 않음
* ADT Set?
* Set, Bag, Stack, Queue 간의 class hierarchies?

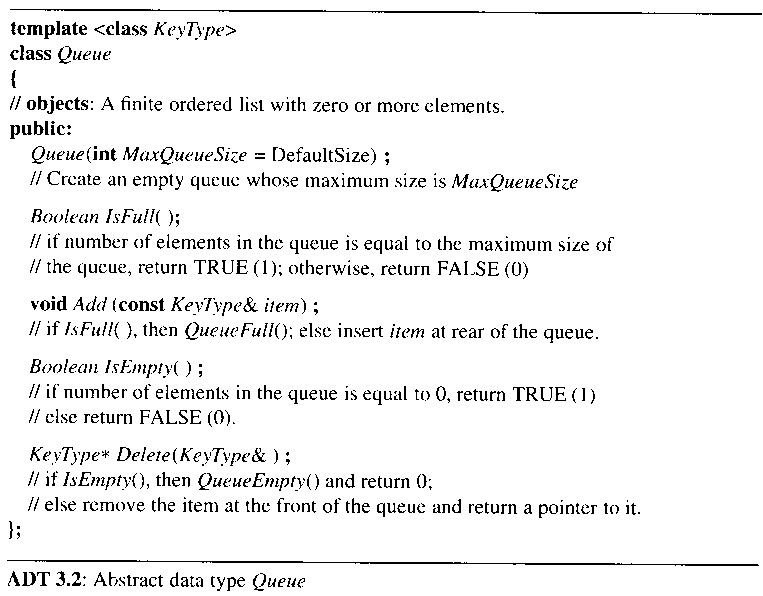
# 3.3 The Queue Abstract Data Type

## queue

### an ordered list in which all insertions take place at one end and all deletions take place at the opposite end



#### First-In-First-Out(FIFO) lists



### ADT에 FIFO를 직접적으로 표현하지 않음

### FIFO는 Add( ), Delete( )의 주석으로 기술함

#### FIFO 구현은 Add, Delete 함수의 body에 표현

#### Add, Delete signature로 FIFO 시맨틱이 추상화됨

## the representation of a queue

### employ a one-dimensional array and two variables, front and rear

#### front < rear

### data member declarations and constructor definition of Queue

#### private:

#### int front, rear;

#### KeyType \*queue;

#### int MaxSize;

#### template <class KeyType>

#### Queue<KeyType>::Queue (int MaxQueueSize):MaxSize (MaxQueueSize)

#### {

#### queue = new KeyType[MaxSize];

#### front = rear = -1;

#### }

## the member functions IsFull() and IsEmpty()

#### template <class KeyType>

#### inline Boolean Queue<KeyType>::IsFull()

#### {

#### if (rear ==MaxSize-1) return TRUE;

#### else return FALSE;

#### }

#### template <class KeyType>

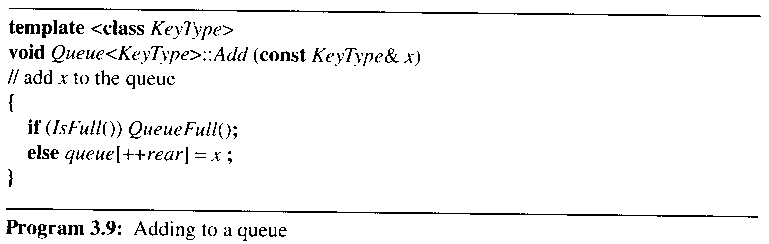
#### inline Boolean Queue<KeyType>::IsEmpty()

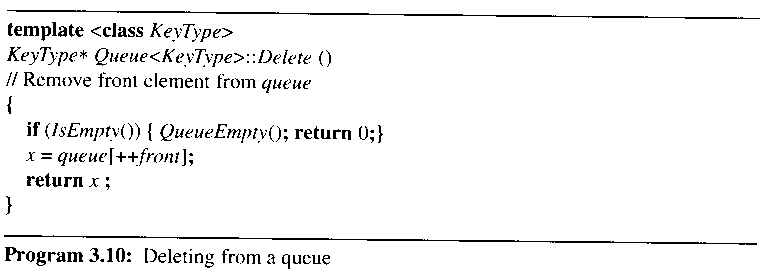
#### {

#### if (front == rear) return TRUE;

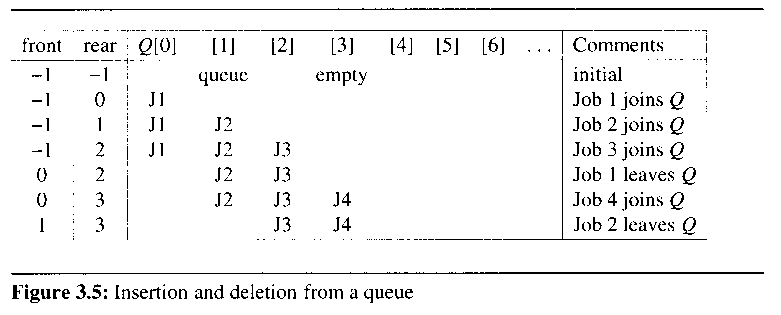
#### else return FALSE;

#### }





# Example 3.2[Job Scheduling]



## the queue gradually shifts to the right

## the queue is full when rear = MaxSize -1

### should move the entire queue to the left

FIG 3.6 Queue example

## a more efficient queue representation : circular queue

### circular queue는 새로운 ADT가 아니고 different representation임

### if rear == MaxSize-1 then rear = 0

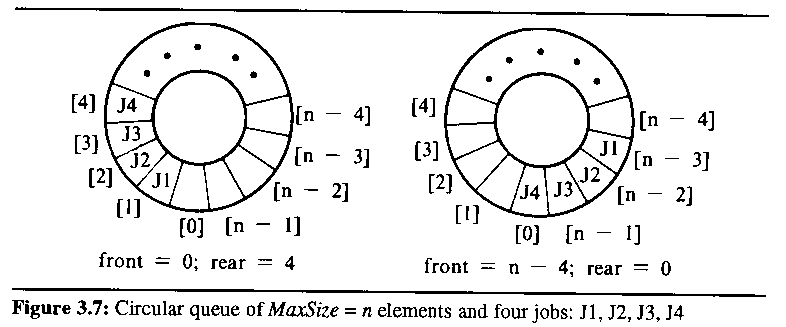
### if front == rear then empty queue

### if front == rear then queue is full

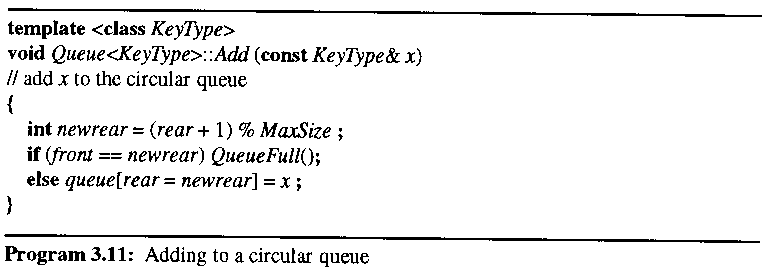
##### the queue contains Maxsize elements

#### permit a maximum of MaxSize-1 rather than MaxSize elements to be in the queue at any time

### initial condition: front == rear == 1

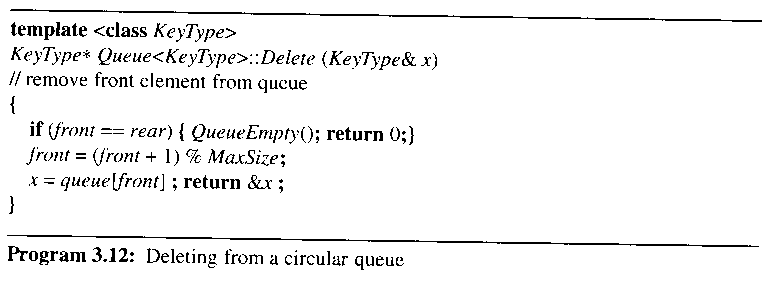


#### use the modulus operator to compute remainders: (rear+1) % MaxSize



#### if (front == newrear) QueueFull();

##### a maximum of Maxsize – 1 rather than Maxsize



### queue의 public interface는 변동이 없음

## one way to use all MaxSize positions

### to use a variable, LastOp, to record the last operation performed on the queue

#### add or delete operator

#### initialized to "delete"

#### set to "add" after each addition

#### set to "delete" following each deletion

### if front == rear and LastOp == "add" then the queue is full

# 3.4 Subtyping and Inheritance in C++

## inheritance

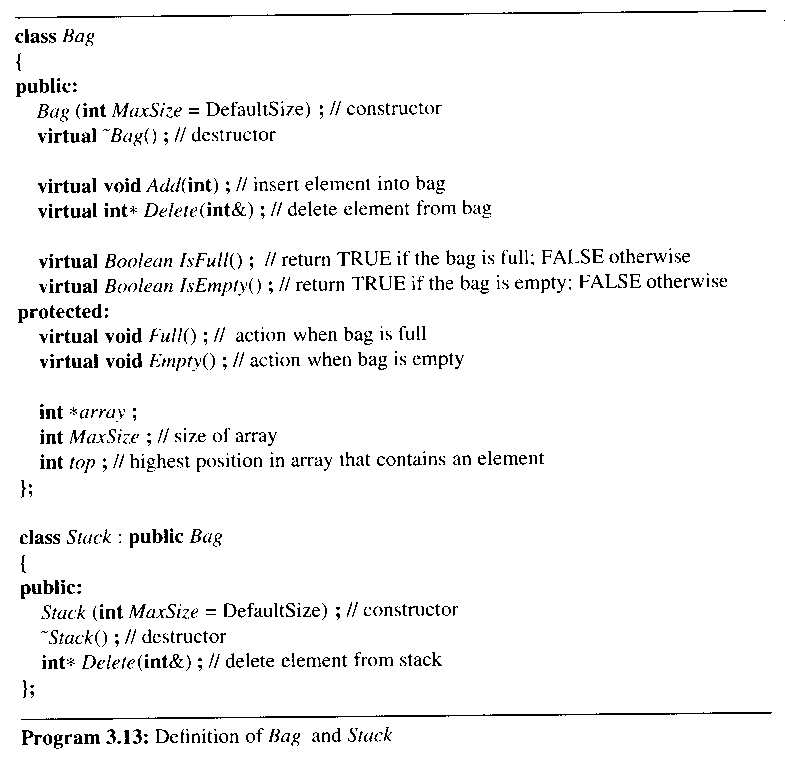
### subtype relationships between ADTs

### IS-A relationship

### if B is-a A then B is more specialized

#### stack is a bag even if their implementations are changed

## implement the IS-A relationship in C++ by using public inheritance



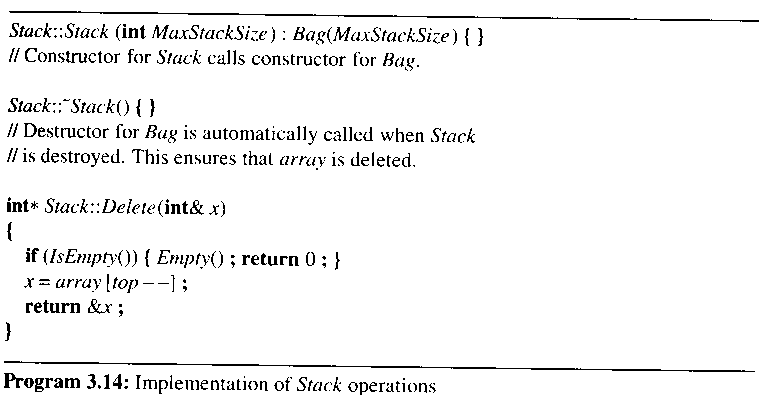
### base class: Bag

### derived class: Stack

#### inherit all the non-private members of the base class

#### inherited members have the same level of access in the derived class (under public inheritance)

#### possible to override the base class implementation



#### virtual member functions

#### class Bag

#### {public:

#### void Add(int);

#### …

#### };

##### dynamic binding?

#### Bag b(3); // use Bag constructor

#### Stack s(3); // use Stack constructor

#### b.Add(1); b.Add(2); b.Add(3);

#### // use Bag::Add

#### // Bag::Add calls functions Bag::IsFull and Bag::Full

#### s.Add(1); s.Add(2); s.Add(3);

#### // use Bag::Add, Stack::Add not defined

#### // Bag::Add calls Bag::IsFull and Bag::Full (not defined in Stack)

#### int x;

#### b.Delete(x); // use Bag::Delete

#### s.Delete(x); // use Stack::Delete

# 3.5 A Mazing Problem

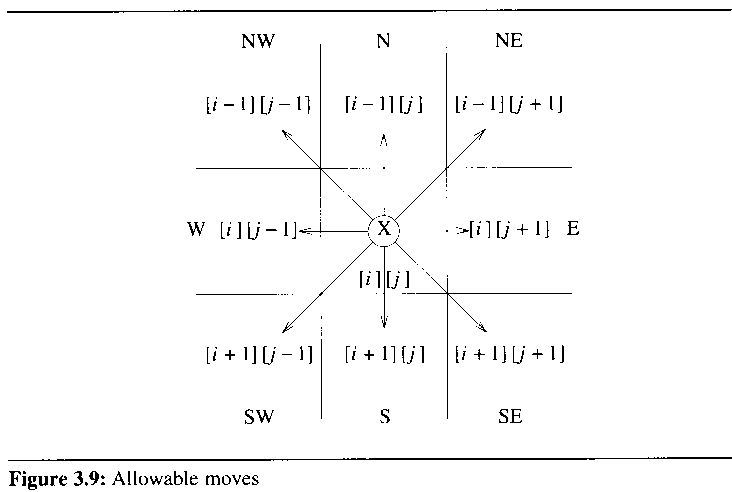
## represent the maze by a two-dimensional array, maze[i][j]

### a blocked path : 1

### can walk right on through : 0

### start at maze[1][1], exit at maze[m][p]fig 3

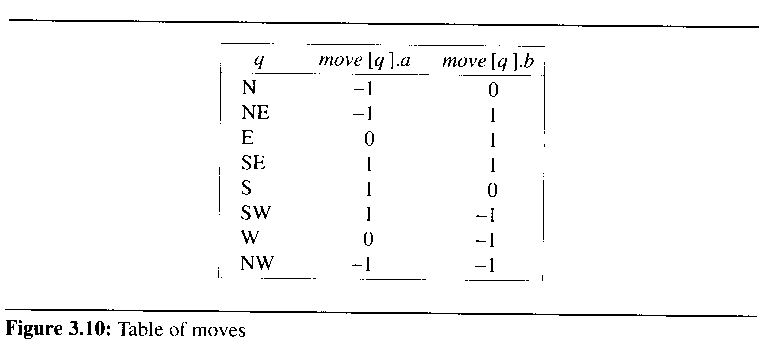
## the possible moves the rat can make from a point [i][j] in the maze



## to avoid checking for border checking

### use maze[m+2][p+2]

## to predefine the possible directions to move in a table, move



#### the data types to define the possible directions

#### struct offsets

#### {

#### int a, b;

#### }

#### enum directions [N, NE, E, SE, S, SW, W, NW};

#### offsets move[8];

### to move the position [g][h] that is southwest from the position [i][j]

#### g = i + move[SW].a; h = j + move[SW].b;

## find a path through a maze

### pick one but save our current position and the direction of the last move in a list

#### if a false path, then return and try another direction

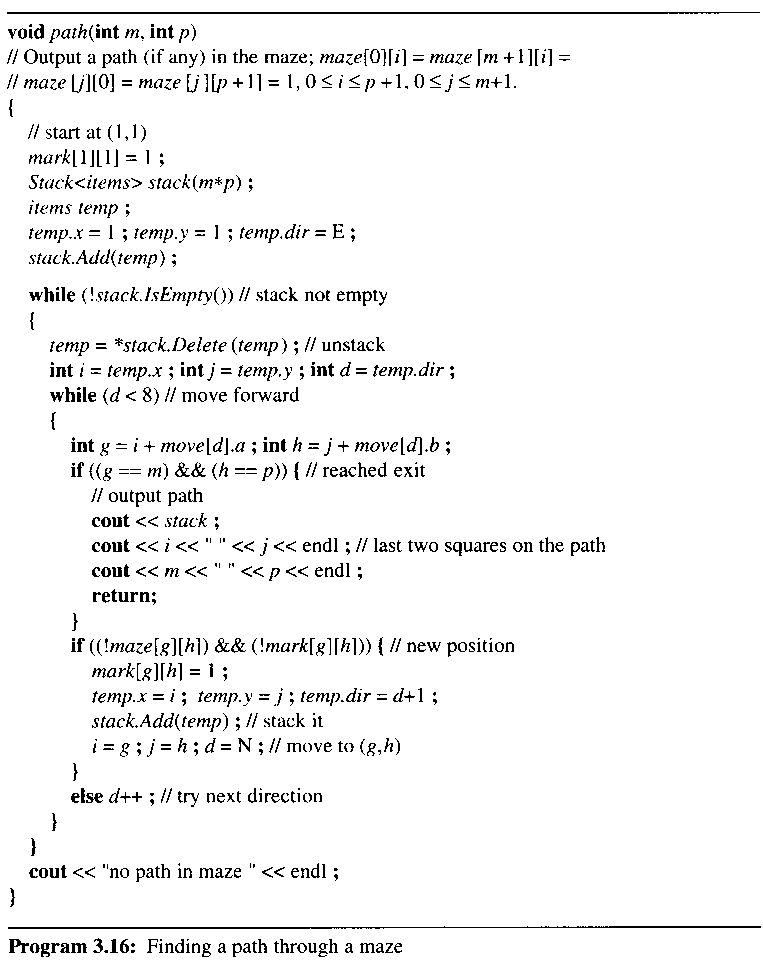
### to prevent us from going down the same path twice

#### use another array, mark[m+2][p+2] which is initialized to zero

#### set mark[i][j] to 1 once we arrive at that position

#### prog 3

#### 예제를 주었을 때 stack 변화를 보여주는 테스트



#### assume maze, mark, move to be global to path()

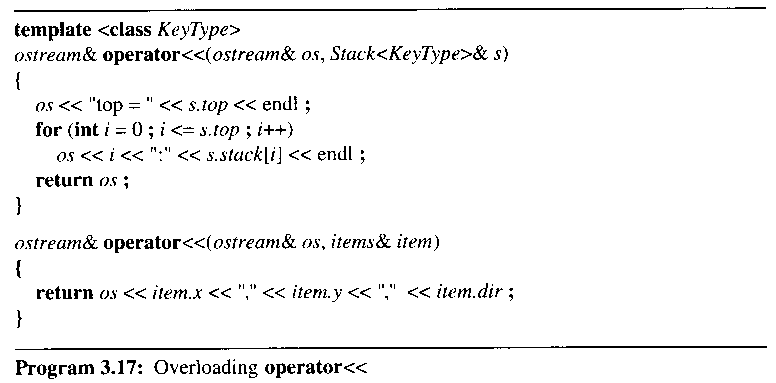
#### define stack to be a stack of items

#### struct items {

#### int x, y, dir;

#### };

#### overloaded operator << for both stack and items



#### operator << accesses private data members of class stack

##### must be made a friend of stack

#### access to the private data members of stack through the use of the friend

## difficult to make the computing time because of recursion

### eight iterations of the inner while loop for each marked position

### if # of zeros in maze = z, then at most z loops

#### the computing time O(mp)

### at most mp elements can be placed into the stack

#### z is bounded by mp

본 강의 자료의 그림 및 알고리즘 발췌

저자 : HOROWITZ

타이틀 : FUNDAMENTALS OF DATA STRUCTURES IN C++ 2nd Edition (2006)

공저 : SAHNI, MEHTA

출판사 : Silicon Press