Chapter 3
Stacks and Queues

3.1 Templates in C++

The mechanism of templates provided by C++ makes classes and functions more reusable.

3.1.1 Templates Functions

A template (or parameterized type) may be viewed as a variable that can be instantiated to any data type.

```
1 template <class T>
2 void  SelectionSort (T *a, int n)
3 //sort a[0] to a[n-1] into nondecreasing order
4 {    for (int  i=0;i<n;i++)
5       {
6          int  j=i;
7          // find the smallest KeyType in a[i] to a[n-1]
8          for (int  k=i +1;k<n;k++)
9              if  (a[k]<a[j]) j=k;
10          swap(a[i], a[j]);
11       }
12 }
```

Function SelectionSort can be used quite easily to sort an array of integers or floats as shown in the following:

```
float farray[100];
int   intarray[250];
…
//assume that the arrays are initialized at this point
sort(farray,100);
sort(intarray,250);
```

Function SelectionSort is instantiated to the type of the array argument supplied to it. e.g., SelectionSort(farray,100) is to sort an array of floats because farray is an array of floats.

For user defined data type T, the operator < should be overloaded in a manner consistent with its usage. e.g., sort is based on the assumption that < is transitive.

Suppose we want to sort an array of Rectangles in non-decreasing order of their areas, operator< should be overloaded with the semantic of area comparing.

The template function ChangeSize1D changes the size of a 1-Dimensional array of type T from oldSize to newSize:

```
template <class T>
void ChangeSize1D(T* a, const int oldSize, const int newSize)
{
   if (newSize < 0) throw “New length must be >= 0”;
   T* temp = new T[newSize];
   int number = min(oldSize, newSize);
   copy(a, a + number, temp);
   delete [] a;
   a = temp;
}
```

3.1.2 Using Templates to Represent Container Classes

A container class---a data structure for containing or storing a number of data objects.

Bag---a data structure into which objects can be inserted and from which objects can be deleted.

A Bag can have multiple occurrences of the same element, but we don't care the position of an element nor do we care which element to delete.

```
template <class T>
class Bag
 {
public:
    Bag (int BagCapacity = 10); // constructor
    ~Bag () ; // destructor

    int Size() const;
    bool isEmpty() const;
    T& Element() const;
    void Push(const T&);
    void Pop();

private:
    T *array;
    int capacity;
    int top;
};
```

```
template <class T>
Bag<T>::Bag(int bagCapacity):capacity(bagCapacity)  {
    if (capacity<1) throw “Capacity must be > 0”;
    array = new T[capacity];
    top = -1;
}

template <class T>
Bag<T>::~Bag()  {delete [ ] array;}

template <class T>
Inline T& Bag<T>::Element() const {
    if (IsEmpty() throw “Bag is empty”;
    return array[0];
}
```

```
template <class T>
void Bag<T>::Push(const T& x)  {
    if (capacity==top+1)
    {
        ChangSize1D(array, capacity, 2*capacity);
        capacity*=2;
     }
     array[++top] = x;
}
```

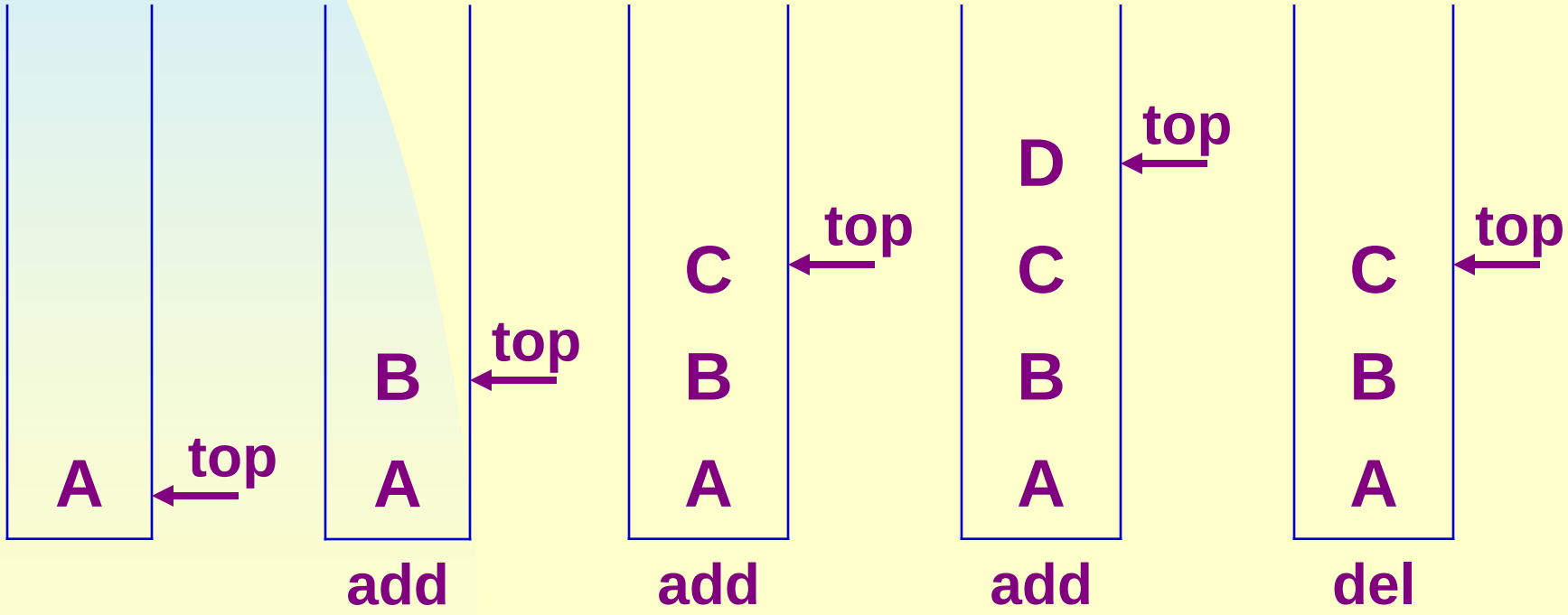
```
template <class T>
void Bag<T>::Pop()  {
    if (isEmpty()) throw “Bag is empty, cannot delete”;
    int  deletePos = top/2; // note we don’t care which to delete
    copy(array+deletePos+1, array+top+1, array+deletePos);
    array[top--].~T(); // destructor for T
}

Bag<int> a;
Bag<Rectangle> r;
```

So a is a Bag of integers and r of Rectangles.

3.2 The Stack Abstract Data type

A Stack---an ordered list in which all insertions and deletions are made at one end called top (LIFO).

Inserting and deleting elements in a stack:



ADT 3.1 Stack

```
template <class T>
class Stack
 { // A finite ordered list with zero or more elements.
 public:
     Stack (int stackCapacity = 10);
    //Creates an empty stack with initial capacity of stackCapacity

     bool IsEmpty() const;
     //If number of elements in the stack is 0, true else false

     T& Top() const;
     // Return the top element of stack

     void Push(const T& item);
     // Insert item into the top of the stack
```

```
    void Pop();
    // Delete the top element of the stack.
};
```

To implement this ADT, we can use an array and a variable top. Initially top is set to –1.

So we have the following data members of Stack:

```
private:
    T* stack;
    int top;
    int capacity;
```

```
template <class T>
Stack<T>::Stack(int stackCapacity): capacity(stackCapacity)
{
    if (capacity < 1) throw “Stack capacity must be > 0”;
    stack = new T[capacity];
    top = -1;
}

template <class T>
Inline bool Stack<T>::IsEmpty() const { return(top == -1);}
```

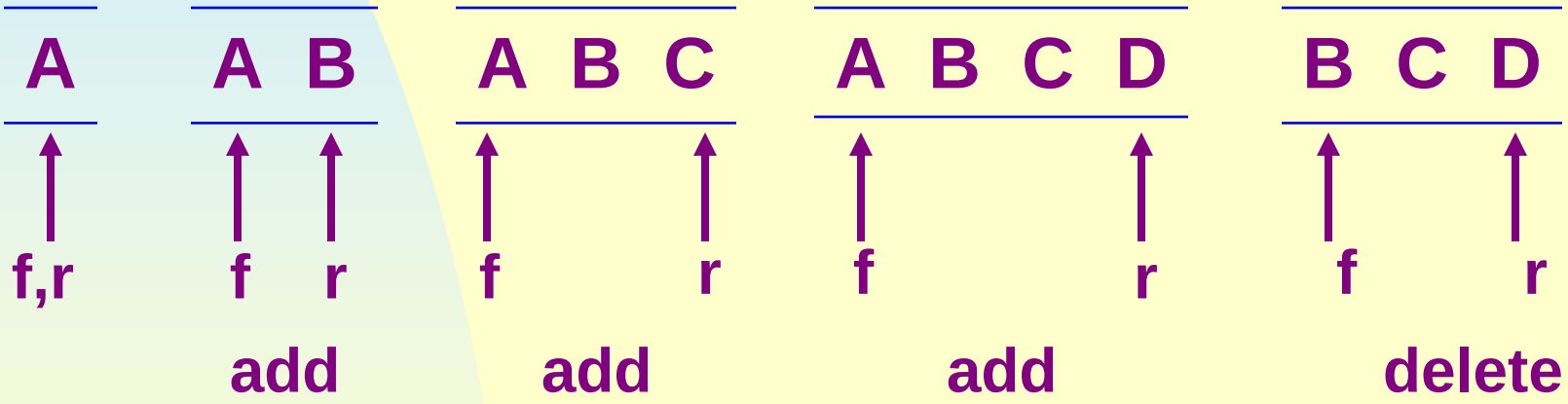
```
template <class T>
inline T& Stack<T>::Top() const
{
    if (IsEmpty()) throw “Stack is Empty”;
    return stack[top];
}

template <class T>
void Stack<T>::Push(const T& x)
{
    if (top == capacity - 1)
    {
       ChangeSize1D(stack, capacity, 2*capacity);
       capacity *= 2;
     }
     stack[++top] = x;
}
```

```
template <class T>
void Stack<T>::Pop()
{ // Delete top element of stack.
    if (IsEmpty()) throw “Stack is empty, cannot delete.”;
    stack[top--].~T();  // destructor for T
}
```

Exercises: P138-1, 2

3.3 The Queue Abstract Data Type

A Queue--- an ordered list in which all insertions take place at one end and all deletions take place at the opposite end (FIFO).



f = queue front r = queue rear

ADT 3.2 Queue

```
template <class T>
class Queue
 { // A finite ordered list with zero or more elements.
 public:
    Queue (int queueCapacity = 10);
    // Creates an empty queue with initial capacity of
    // queueCapacity

    bool IsEmpty() const;

    T& Front() const; //Return the front element of the queue.

    T& Rear() const; //Return the rear element of the queue.

    void Push(const T& item);
     //Insert item at the rear of the queue.
```

```
    void Pop();
    // Delete the front element of the queue.
};
```

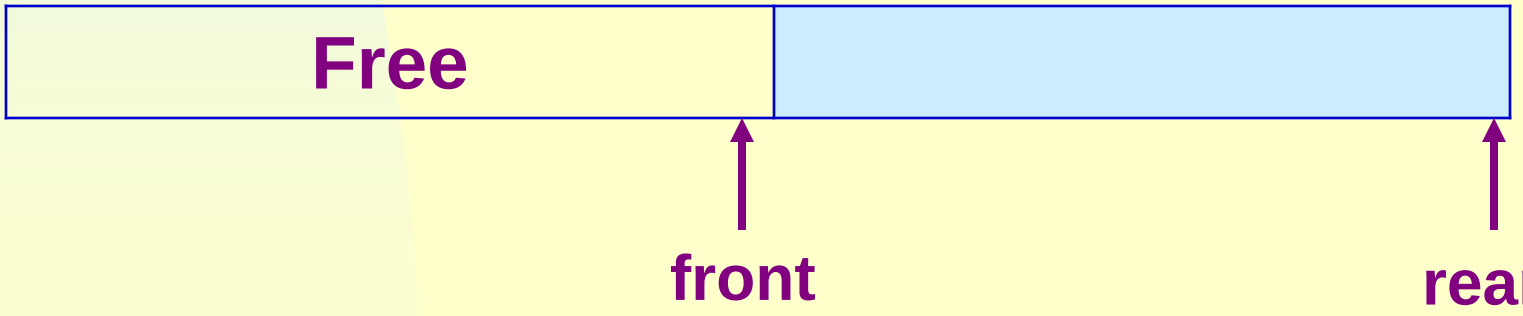
To implement this ADT, we can use an array and two variable front and rear with front being one less than the position of the first element. So we have the following data members of Queue:

```
private:
    T* queue;
    int front,
        rear,
        capacity;
```

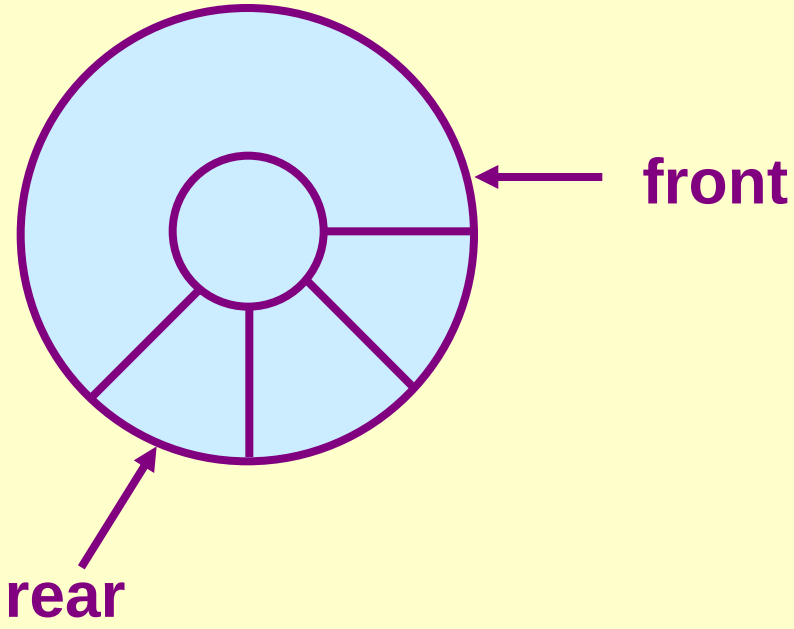
At the beginning, we can set front = rear = -1, and front == rear means that the queue is empty.

As we push and pop the queue, front may be > 0, and rear may be = capacity – 1.

In that case, we cannot add an element to the queue without shifting all elements to the left, as shown below:



Hence, circular representation. To distinguish between queue full and queue empty, we shall increase the capacity of a queue just before it becomes full.

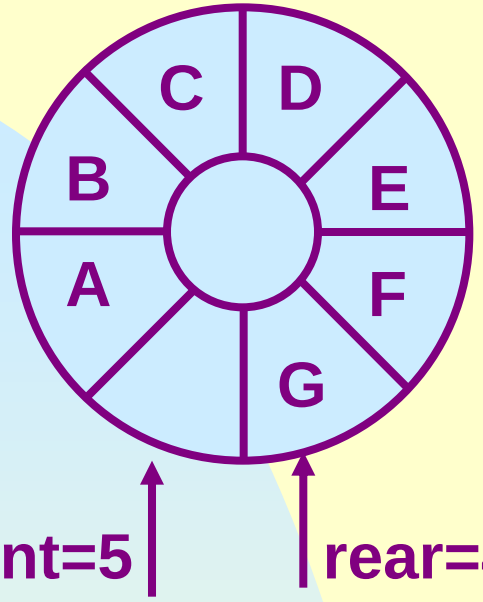


```
template <class T>
Queue<Type>::Queue(int queueCapacity):
                                    capacity(queueCapacity)
{
   if (capacity < 1) throw “Queue capacity must > 0”;
   queue = new T[capacity];
   front = rear = 0;
}
```

```
template <class T>
Inline bool Queue<T>::IsEmpty() const {return front==rear };

template <class T>
inline T& Queue<T>::Front() const
{
   if (IsEmpty()) throw “Queue is empty. No front element”;
   return queue[(front+1)%capacity];
}

template <class T>
inline T& Queue<T>::Rear() const
{
   if (IsEmpty()) throw “Queue is empty. No rear element”;
   return queue[rear];
}
```

```
template <class T>
void Queue<T>::Push(const T& x)
{ // add x at rear of queue
   if ((rear+1)%capacity == front)
   { // queue full, double capacity
     // code to double queue capacity comes here
   }
   rear = (rear+1)%capacity;
   queue[rear] = x;
}
```

We can double the capacity of queue in the way as shown in the next slide:



queue

0	1	2	3	4	5	6	7
C	D	E	F	G		A	B

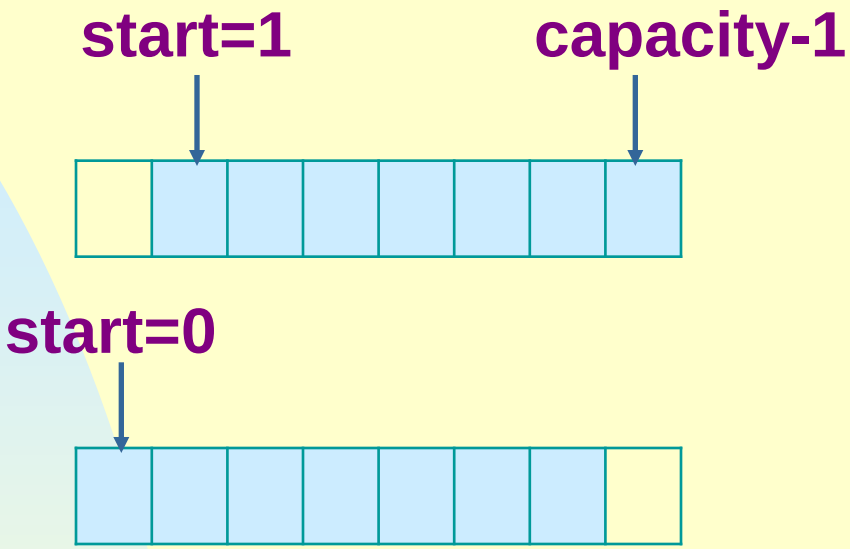
front=5, rear=4

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	B	C	D	E	F	G									

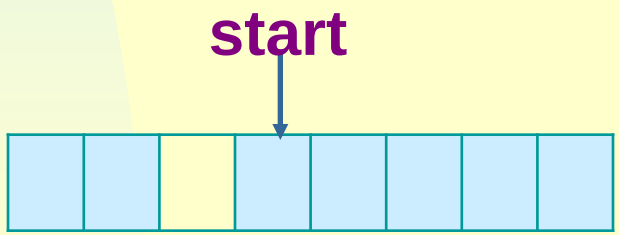
front=15, rear=6

Let start = (front+1)%capacity

Case 1, start < 2:



Case2, start >= 2:



Thus, the previous configuration may be obtained as follows:

(1) Create a new array newQueue of twice the capacity.

(2) Copy the second segment (if any) to positions in newQueue beginning at 0.

(3) Copy the first segment (if any) to positions in newQueue beginning at (capacity-start)%capacity.

The code is in the next slide:

```
// allocate an array with twice the capacity
T*  newQueue = new T[2*capacity];

// copy from queue to newQueue
int start = (front+1)%capacity;
if (start < 2)
    // no wrap around
    copy(queue+start, queue+start+capacity-1, newQueue);
else
{ // queue wraps around
    copy(queue+start, queue+capacity, newQueue);
    copy(queue, queue+rear+1, newQueue+capacity-start);
}

// switch to newQueue
front = 2*capacity-1;  rear =  capacity-2; capacity *= 2;
delete [] queue;
queue = newQueue;
```

```
template <class T>
void Queue<T>::Pop()
{  // Delete front elemnet from queue
   if (IsEmpty()) throw “Queue is empty. Cannot delete”;
   front = (front+1)%capacity;
   queue[front].~T;
}
```

For the circular representation, the worst-case add and delete times (assuming no array resizing is needed) are O(1).

Exercises: P147-1, 3.

3.5 A Mazing Problem

Problem: find a path from the entrance to the exit of a maze.

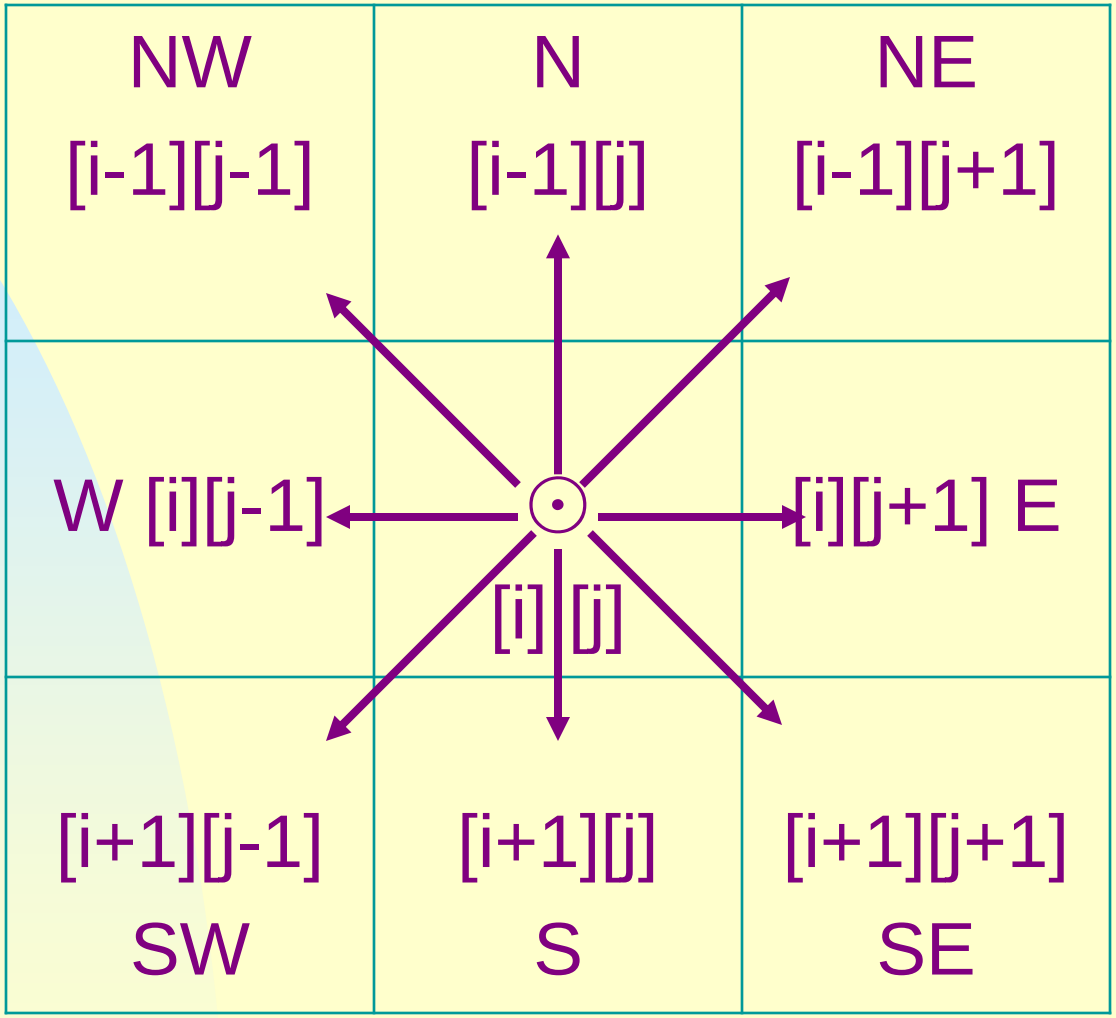
entrance

0	1	0	0	1	1	0	1	1
1	**0**	**0**	1	0	0	1	1	1
0	1	1	**0**	1	1	1	0	1
1	1	0	**0**	1	**0**	**0**	1	0
1	0	0	1	**0**	1	1	**0**	1
0	0	1	1	0	1	**0**	1	1
0	1	0	0	1	1	**0**	**0**	**0**

exit

Representation:

- maze[i][j], $1 \le i \le m$, $1 \le j \le p$.
- 1--- blocked, 0 --- open.
- the entrance: maze[1][1], the exit: maze[m][p].
- current point: [i][j].
- boarder of 1's, so a maze[m+2][p+2].
- 8 possible moves: N, NE, E, SE, S, SW, W and NW.


NW
[i-1][j-1]
N
[i-1][j]
NE
[i-1][j+1]
W [i][j-1]
[i][j]
[i][j+1] E
[i+1][j-1]
SW
[i+1][j]
S
[i+1][j+1]
SE

To predefine the 8 moves:

```
struct  offsets
{
    int a,b;
};
enum directions {N, NE, E, SE, S, SW, W, NW};
offsets move[8];
```

q	move[q].a	move[q].b
N	-1	0
NE	-1	1
E	0	1
SE	1	1
S	1	0
SW	1	-1
W	0	-1
NW	-1	-1

Table of moves

Thus, from [i][j] to [g][h] in SW direction:

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

The basic idea:

Given current position [i][j] and 8 directions to go, we pick one direction d, get the new position [g][h].

If [g][h] is the goal, success.

If [g][h] is a legal position, save [i][j] and d+1 in a stack in case we take a false path and need to try another direction, and [g][h] becomes the new current position.

Repeat until either success or every possibility is tried.

In order to prevent us from going down the same path twice, use another array mark[m+2][p+2], which is initially 0.

Mark[i][j] is set to 1 once the position is visited.

First pass:

Initialize stack to the maze entrance coordinates and direction east;

while (stack is not empty)

{

(i, j, dir)=coordinates and direction from top of stack;

pop the stack;

while (there are more moves from (i, j))

{

(g, h)= coordinates of next move ;

if ((g==m) **&&** (h==p)) success;

```
        if ((!maze[g][h]) && (!mark[g][h])) // legal and not visited
        {
            mark[g][h]=1;
            dir=next direction to try;
            push (i, j, dir) to stack;
            (i, j, dir) = (g, h, N);
         }
    }
}
cout << “No path in maze.”<< endl;
```

We need a stack of items:

```
struct Items {
        int x, y, dir;
 };
```

Also, to avoid doubling array capacity during stack pushing, we can set the size of stack to m*p.

Now a precise maze algorithm.

```
void path(const int m, const int p)
{ //Output a path (if any) in the maze; maze[0][i] = maze[m+1][i]
  // = maze[j][0] = maze[j][p+1] = 1, 0 ≤ i ≤ p+1, 0 ≤ j ≤ m+1.
     // start at (1,1)
     mark[1][1]=1;
     Stack<Items> stack(m*p);
     Items temp(1, 1, E);
     stack.Push(temp);
     while (!stack.IsEmpty())
     {
           temp= stack.Top();
          Stack.Pop();
          int i=temp.x; int j=temp.y; int d=temp.dir;
```

```
while (d<8)
{
    int g=i+move[d].a; int h=j+move[d].b;
    if ((g==m) && (h==p)) { // reached exit
       // output path
       cout <<stack;
       cout << i<<" "<< j<<" "<<d<< endl; // last two
       cout << m<<" "<< p<< endl;          // points
       return;
     }
```

```
            if ((!maze[g][h]) && (!mark[g][h])) { //new position
                mark[g][h]=1;
                temp.x=i; temp.y=j; temp.dir=d+1;
                stack.Push(temp);
                i=g ; j=h ; d=N;   // move to (g, h)
             }
             else d++;  // try next direction
        }
    }
 cout << "No path in maze."<< endl;
}
```

The operator << is overloaded for both Stack and Items as:

```
template <class T>
ostream& operator<<(ostream& os, Stack<T>& s)
{
    os << "top="<<s.top<< endl;
    for (int i=0;i<=s.top;i++);
        os<<i<<":"<<s.stack[i]<< endl;
    return os;
}
```

We assume << can access the private data member of Stack through the friend declaration.

```
ostream& operator<<(ostream& os,Items& item)
{
    return os<<item.x<<“,”<<item.y<<“,”<<item.dir-1;
    // note item.dir is the next direction to go so the current
    // direction is item.dir-1.
}
```

Since no position is visited twice, the worst case computing time is O(m*p).

Exercises: P157-2, 3

3.6 Evaluation of Expressions

3.6.1 Expressions

A expression is made of operands, operators, and delimiters. For instance,

X = A / B – C + D * E – A * C

- **the order in which the operations are carried out must be uniquely defined.**
- **to fix the order, each operator is assigned a priority.**

- within any pair of parentheses, operators with highest priority will be evaluated first.
- evaluation of operators of the same priority will proceed left to right.
- Innermost parenthesized expression will be evaluated first.

The next slide shows a set of sample priorities from C++.

priority	operator
1	unary minus, !
2	*, /, %
3	+, -
4	<, <=, >=, >
5	==, !=
6	&&
7	||

Figure 3.15

Problem: how to evaluate an expression?

Solution:

1.Translate from infix to post fix;

2.Evaluate the postfix.

3.6.2 Postfix Notation

Infix: operators come in-between operands (unary operators precede their operand).

Postfix: each operator appears after its operands.

e.g.

infix: **A / B – C + D * E – A * C**

postfix: **A B / C – D E * + A C * –**

Every time we compute a value, we store it in the temporary location T_i, $i \geq 1$. Read the postfix left to right to evaluate it:

A B / C – D E * + A C * –

operation	postfix
T_1=A/B	T_1 C – D E * + A C * –
T_2=T_1 - C	T_2 D E * + A C * –
T_3=D*E	T_2 T_3 + A C * –
T_4=T_2+ T_3	T_4 A C * –
T_5=A*C	T_4 T_5–
T_6= T_4–T_5	T_6

T_6 is the result.

Virtues of postfix:

- **no need for parentheses**
- **the priority of the operators is no longer relevant**

These make the evaluation of postfix easy: make a left to right scan, stack operands, and evaluate operators using the correct number of operands from the stack and place the result onto the stack.

```
void  Eval(Expression e)
{ // evaluate the postfix expression e. It is assumed that the
  // last token in e is '#'. A function NextToken is used to get
  // the next token from e. Use stack.
     Stack<Token> stack;  //initialize stack
     for  (Token x = NextToken(e); x!='#'; x=NextToken(e))
        if (x is an operand) stack.Push(x);
        else {  // operator
            remove the correct number of operands for operator x
            from stack; perform the operation x and store the result
            (if any) onto the stack;
         }
}
```

3.6.3 Infix to Postfix

Idea: note the order of the operands in both infix and postfix is the same, we can form the postfix by immediately passing any operands to the output, then store the operators in a stack until just the right time.

e.g.

A*(B+C)*D → ABC+*D*

The algorithm behaves as:

Next token	stack	output
A	empty	A
*	*	A
(	*(	A
B	*(	AB
+	*(+	AB
C	*(+	ABC
)	*	ABC+
*	*	ABC+ *
D	*	ABC+ *D
done	empty	ABC+ *D*

Attention

From the example, we can see the left parenthesis behaves as an operator with high priority when its not in the stack, whereas once it get in, it behaves as one with low priority.

- isp (in-stack priority)
- icp (in-coming priority)
- the isp and icp of all operators in Fig. 3.15 remain unchanged
- isp(‘(‘)=8, icp(‘(‘)=0, isp(‘#’)=8

Hence the rule:

Operators are taken out of stack as long as their isp is numerically less than or equal to the icp of the new operator.

```
void Postfix (Expression e)
{ // output the postfix of the infix expression e. It is assumed
  // that the last token in e is '#'. Also, '#' is used at the bottom
  // of the stack.
    Stack<Token> stack;  //initialize stack
    stack.Push('#');
```

```
  for  (Token x=NextToken(e); x!='#'; x=NextToken(e))
    if (x is an operand) cout<<x;
    else if (x==')')
        { // unstack until '('
          for  (; stackTop()!='('; stack.Pop())
              cout<<stack.Top();
          stack.Pop();  // unstack '('
          }
    else { // x is an operator
         for (; isp(stack.Top()) <= icp(x); stack.Pop())
            cout<<stack.Top();
         stack.Push(x);
     }
  // end of expression, empty the stack
  for (; !stack.IsEmpty()); cout<<stack.Top(), stack.Pop());
  cout << endl;
}
```

Analysis:

- Computing time: one pass across the input with n tokens, O(n).
- The stack will not be deeper than 1 (‘#’) + the number of operators in e.

Exercises: P165-1,2