```
    {
        if (expr[i] == '(')
            Push(&s, i);
        else if (expr[i] == ')')
            {
                j = Pop(&s);
                if (j != -1)
                    printf("%d  %d\n",j+1,i+1);
                else
                    printf("No match for right
                      parenthesis at: %d\n", i+1);
            }
    }

    while (!Empty(&s))
    {
        j = Pop(&s);
        printf("No match for left parenthesis
                              at: %d\n", j+1);
    }
}
```

Whenever a left parenthesis is encountered in the input expression `expr`, push its position (`i`) in the stack and whenever a right parenthesis is read, the matching left parenthesis position is popped (`j`) and displayed. When you get an extra ')', a pop is initiated from an empty stack which returns –1, hence flag an error. Similarly, when you get an extra left parenthesis '(', it would have been stored in stack. The *while* loop pops all extra left parentheses and flags the error "*No match for left parenthesis*". A sample run of the above function is shown below:

Example Run 1

```
Enter an expression
((a+b)*c)
2  6
1  9
```

Example Run 2

```
Enter an expression
((a+b)
2  6
No match for left parenthesis at: 1
```

Example Run 3

```
Enter an expression
(a+b))
1 5
No match for right parenthesis at: 6
```

3.4.2 Evaluation of Postfix Expression

An expression in computer field is not a new concept; in fact, arithmetic expressions are common in mathematics. Before we attempt to design an algorithm for the evaluation of a **postfix** expression, it is necessary to understand the basic concepts and examples of **infix**, **postfix** and **prefix** expressions.

Basic definitions and Examples – Infix Notation

The expression what we use in mathematics is called as the **infix expression**. Consider the following simple arithmetic expression,

A + B

In this expression, operands of the operator + are A and B. When the operator is fixed between the operands, then we say such an expression as infix expression. The infix expressions may also contain parenthesis to show the order of evaluation – i.e., **precedence**. For example, in the below infix expression

A + B * C

B * C is evaluated first and then the result of this sub expression becomes the second operand for + and of course the first operand is A. You can change this predefined order of evaluation by enclosing the expression(s) within brackets.

(A + B) * C

Now, (A + B) is evaluated first and then the result is multiplied with C. We are comfortable in writing infix expressions because of early knowledge of mathematics. However, for compilers it is not an easy job to evaluate infix expressions with one single left-to-right scan. To evaluate such expressions in one single scan is possible if it is written in postfix notation.

Postfix Notation

When the operator is placed after the two operands, it is called as **postfix expression** or suffix expression. This type of notation is known as *Kukasiewicz notation* (due to the Polish logician Jan Kuksiewicz). The suffix expression is some times called as **Reverse Polish** notation. Examples for postfix expression are given in Table 3.1.

Notice that suffix expressions do not have parentheses unlike infix expressions. Postfix expressions have the precedence of operators being attached to the expressions from left to right. Take for instance the (iii) expression, where AB+ is done first as indicated in the infix notation (A + B). Then this result is used as the first operand for *.

Table 3.1

	Infix	Postfix (reverse Polish)
(i)	A + B	A B +
(ii)	A + B * C	A B C * +
(iii)	(A + B) * C	A B + C *

Prefix notation

If the operator precedes the tow operands, such an expression is called as **prefix notation** or **Polish notation**. Consider the same infix expression A + B and its equivalent prefix expression is + A B.

We will rewrite the expressions shown in Table 3.1 to show all three types of notations and it appears in Table 3.2

Table 3.2

	Infix	Suffix	Prefix
(i)	A + B	A B +	+ A B
(ii)	A + B * C	A B C * +	+ A * B C
(iii)	(A + B) * C	A B + C *	* + A B C

Converting infix to other notations

As evaluation of suffix expressions is easier and efficient than its equivalent expressions, we shall study algorithms to convert from one notation to the other.

Example 1 *Infix to Postfix*

A + B * C
A + [B C*]
A B C * + ← Answer

In converting from infix to postfix, follow stepwise conversion. Considering the above example, first convert B * C to B C *, because * has higher precedence than +. Then, consider A + B C * whose final postfix is A B C * +.

Example 2

(A + B) * C
[A B +] * C
A B + C * ← Answer

Example 3

(A + B) * (C – D) $ E * F ; $ is used for power
[A B +] * [C D -] $ [E] * [F]

[AB+] * [CD-E$] * [F] ; * left to right

[AB+CD-E$*] * [F]

AB+CD−E$*F* ← Answer

Example 4 ***Infix to Prefix***

A + B * C

A + [*BC]

+ A * B C ← Answer

Example 5

(A + B) * C

[+AB] * C

* + A B C ← Answer

Example 6

(A + B) * (C − D) $ E * F

[+AB] * [−CD] $ [E] * [F]

[+AB] * [$-CDE] * [F]

[*+AB$-CDE] * [F]

* * + A B $ - C D E F ← Answer

Example 7 ***Prefix to infix***

+ - A B C

+ [A-B] C

(A − B) + C or A − B + C ← Answer

Example 8

+ + A - * $ B C D / + E F * G H I

+ + A - * [B$C] D / [E+F] [G*H] I ← Step 1

+ + A - [(B$C)*D] / (E + F) (G * H) I

+ + A - [(B$C)*D] [(E+F)/(G*H)] I

+ + A [((B$C)*D)-((E+F)/(G*H))] I

+ [A+((B$C)*D)- ((E+F)/(G*H))] I

A + ((B $ C) * D) − ((E + F) / (G * H)) + I ← Answer

First look for an operator that is followed by two operands (A, B, C, …, Z). Convert them into infix form. In Step 1, you see three such sub expressions (B $ C), (E + F) and (G * H). Continue this process until you process the leftmost operator. The grouping is shown in boxes.

Example 9 Postfix to Infix

A B + C –
[A + B] C –
(A + B) – C or A + B – C ← Answer

Converting postfix to infix is much easier than converting prefix to infix. The reason is that the precedence of the operators is in the same order as it appears in the suffix expression.

Example 10

A B – C + D E F - + $
[A - B] C + D E F - + $
[(A - B) + C] D E F - + $
[(A - B) + C] D [(E - F)] + $
[(A - B) + C] [D + (E - F)] $
((A – B) + C) $ (D + (E – F)) ← Answer

Now we are in a position to devise a method to evaluate a given postfix expression.

Problem statement for Postfix evaluation

The objective of this problem is to show how a stack is useful for the evaluation of a postfix expression. Secondly, to develop an algorithm and C program to evaluate a given postfix expression. We will assume the following points:

- The input is a valid postfix expression (string of digits and operators).
- The expression contains only digits from 0, 1, 2, …,9.
- No blank space is allowed in between characters.
- The allowed operators are +, -, *, / and $.
- The final evaluated result is returned to the calling routine.
- Only binary operators are allowed.

For example, for the postfix expression 2 3 4 * +, we must get the result as 14.

The method

It is easy to think that the property of stack (LIFO) can be used to evaluate a postfix expression because the operator always succeeds the operands. Consider a simple example – B C *, to know what operation to be performed on this expression you must

wait until * is read. Since only one scanning is allowed (from left to right), we are forced to remember the operands B and C. To do so, B and C are pushed into a stack as we scan from left to right. When * is read, the operands can be popped from stack and evaluated.

This process is true even for a complicated expression, which contains a sub expression, say A B C * + (A + B * C is its infix equivalent). The symbols A, B and C are all stored in the stack in the same order as they appear, i.e., C will be at the top of the stack. When * is scanned, the last inserted two operands are popped (LIFO) and (B * C) is calculated and pushed back into stack. Now, top of the stack contains B * C and the bottom most is the symbol A. When + is read, again the value of (B * C) and A are popped and added and again pushed into stack. Since the end of string is reached, the final result is obtained by one pop. The entire steps can now be summarized as,

Step 1: Initialize the Stack.
Step 2: Repeat thru Step 4 until end of string is reached.
    Step 3: Get the symbol.
    Step 4: If symbol is a digit
        then Push it into stack
        else if it is an operator
            then Pop twice and perform the required operation
                Push the sub expression result
            else error – invalid char
Step 5: Pop the final answer and return the result.
Step 6: End.

Implementation in C

The input postfix expression is stored in a character array (not an integer array!). The character array is used because the expression normally contains digits and operators – which is a character. To convert an ASCII digit to its numeric data (decimal), simply adopt the below given logic.

'5' – '0' = 53 – 48 = 5

Hence, any character, `c`, read will be first converted to its equivalent integer (or float) by using the formula `(float)c - '0'`. Before we explain the designing of the functions, it is appropriate to show with a diagram that our logic really works (shown in Figure 3.6).

The function Eval()

The C code for evaluation of a suffix expression is shown in Program 3.5 – `Eval()`. The main `for` loop iterates by picking the first character pointed by `pos` and up to the end of the string ('\0').

Example

Infix expression: "3 + 4 * 5 \0"
Postfix expression: "3 4 5 * + \0"

Input symbol	Stack contents	Remarks
-	Empty Stack	-
3	Top→ 3	3 is a digit, hence push it.
4	Top→ 4, 3	4 is also pushed.
5	Top→ 5, 4, 3	5 is a digit, push it.
*	Top→ 20, 3	Operator, hence pop 5 and 4 and do 5 * 4 = 20 and push it again.
+	Top→ 23	Operator, hence pop 20 and 3 and do 20 + 3 = 23 and push it again.
\0	Pop(23) and return the result.	

Fig. 3.6 Snapshot of evaluation of "3 4 5 * +"

Program 3.5
Suffix Evaluation

```
float Eval (char expr[])
{
     int c,pos;
     float opnd1,opnd2,value;
     struct stk opndstack;
     opndstack.top = -1;
```

```
    for (pos = 0; (c = expr[pos]) != '\0'; pos++)
         if (isdigit(c))
         /* returns nonzero value, if c is 0 to 9 */
              Push (&opndstack, (float) (c - '0'));
         else
         {
              opnd2 = Pop(&opndstack);
              opnd1 = Pop(&opndstack);
              value = oper(c,opnd1,opnd2);
              Push(&opndstack,value);
         }
         return(Pop(&opndstack));
}

/* Returns float after appropriate operation */
float oper (char symb, float op1, float op2)
{
    switch (symb)
    {
         case '+' : return(op1 + op2);
         case '-' : return(op1 - op2);
         case '*' : return(op1 * op2);
         case '/' : return(op1 / op2);
         case '$' : return(pow(op1, op2));
         default  : printf("%s","Error:Illegal
                                operator");
                    exit(1);
    }
}
```

If `c` is a digit (`isdigit()` finds this) it is pushed into stack after converting it into a *float* number. Otherwise pop the operands and do the required operation by calling `oper()` function. The final result is in stack and is returned to the caller by one pop operation.

3.4.3 Infix to Suffix conversion

The need for converting an infix arithmetic expression to suffix is that a number of compilers require such conversions for evaluation purpose. Section 3.4.2 demonstrated how simple is to evaluate a suffix or reverse Polish notation in one scan. However, it is too much to ask the users to write their expressions in suffix form. Generally, for a long time we are used to infix form and a program can be designed for the conversion. This is the exact objective of this section. For example, if we are given an infix expression

*(a + b) * c*, then the expected result is *a b + c **. Similarly, *a + b * c* should yield *a b c * +*. The assumptions that we considered for this problem are listed below:

- The infix expression contains only single character operands i.e., *a*, *b*, ..., *z*.
- The valid operators are +, -, *, /, $ or ^.
- The expression may or may not contain parentheses.
- The given infix expression is assumed to be valid.

The Method

The backbone of the algorithm is the precedence value assigned to various types of symbols in the expression – operands, operators, left parenthesis, right parenthesis, etc. The precedence of the symbols decides whether the symbol will go into the stack or be padded with the output postfix string.

We must maintain two types of precedence for the symbols – the first is the incoming symbol precedence and the other one is the stack precedence. An incoming symbol with a precedence value greater than that of the top element of the stack will be pushed into stack. Because, the higher precedence operators should appear first in the suffix string compared to other operators on the stack. Table 3.3 shows the precedence of various symbols. Notice that the precedence values are written in an ascending order, assigning left parenthesis the highest.

Table 3.3

Symbol	Incoming symbol Precedence (*f*)	Stack Precedence (*g*)
+ , -	1	2
* , /	3	4
$, ^	6	5
a, b, ..., z	7	8
(	9	0
)	0	-

Let us devise the method first for an unparenthesized expression and later introduce how to handle parenthesized expression. The example infix expression is A + B * C. As explained already, when the incoming symbol precedence is greater than the stack precedence, it is pushed in the stack, otherwise pop the symbol and put it in the output.

When we scan the expression, the first symbol is A and this should be put in the stack. For this to happen there should be already a symbol in the stack that has a lower precedence. We shall assume that '(' is already pushed in the stack before the main loop starts. Note that its stack precedence is assigned 0 for this purpose. Next symbol is + and since its precedence is 1, symbol A is popped and put in the output string. No more symbols will be popped as all the symbols have their precedence value greater than g('('). Note, that + is saved in stack (SUFFIX = "A").

The symbol B is pushed into stack with the same arguments as before and when * is read, B is popped and added to output string (SUFFIX = "A B"). The precedence of * is greater than + and so it is kept in stack only.

When C is scanned, it is pushed in the stack. Now the current stack contents is "(+ * C" in which C is at the top of the stack. Now to force all the symbols to come of the stack, we assume that every infix expression is attached with a ')' and assign its incoming symbol precedence (f) as 0. Since ')' has this lowest precedence value, all but '(' will be removed from stack. So, SUFFIX = "A B C * +" which is our expected result.

Let us now consider few points related to parenthesized expressions. All the left parentheses will be stored in stack. However, the right parenthesis will not be stored in the stack (see its precedence value f ('(') = 0). Because, when you encounter ')', simply pop the symbol until you get '(' and send it to SUFFIX. Note that the stack precedence of '(' and incoming symbol precedence of ')' is same (i.e., 0) to remove '(' from stack and also not to add it into SUFFIX.

Combining all these points, we can write the algorithm for converting infix expression to suffix expression and is shown in Algorithm 3.1

Algorithm 3.1 Infix to Postfix

```
Algorithm Infix-to-Postfix(infix, SUFFIX)
{
    // infix - given infix expression - char string
    // postr - suffix expression output - char string.
    // s - stack
    // top - pointer to top of stack
    // f and g are precedence functions
    // Initialize_Stack
    top = 0;
    s[top] = '(';
    cur = 0;   // cursor to scan infix
    while((symb = infix[cur]) != '\0') do
    {
         while (f(symb) < g(s[top]))
         {
              c = pop(s);
              Append c to SUFFIX;
         }
         if (f(symb) != g(s[top]))
              Push(s, symb);
         else
              Pop(s);
         cur = cur + 1;   // point to the next character
    }
}
```

Example 1 *Table 3.4*

Input infix expression: (A + B) * C

	Input symbol	Stack contents	Output (postr)	Remarks
1.	-	Top→ (	-	Stack initialization
2.	(	Top→ ((	-	f(() > g(()
3.	A	Top→ A ((	-	f(A) > g(()
4.	+	Top→ + ((	A	f(+) < g(A) Push +
5.	B	Top→ B + ((	A	f(B) > g(+)
6.	)	Top→ (	A B +	f()) < g(B) all symbols up to (are popped
7.	*	Top→ * (	A B +	f(*) > f(()
8.	C	Top→ C * (	A B +	f(c) > f(*)
9.	)	Empty Stack	A B + C *	since f()) is less that C and *, both are popped and f()) = g(() one pop will clear the stack

We shall trace Algorithm 3.1 with a sample expression and is shown in Table 3.4. As per the algorithm, the stack initially contains '(' and the expression will be padded with ')' as the last character.

The function `Postfix()` is a C code written using Algorithm 3.1 and is shown in Program 3.5.

Program 3.5
Infix to Postfix Conversion

```
void Postfix (char infix[], char postr[])
{
     int cur, p, len;
     int i = 0;
     char symb;
     struct stk s;
     len = strlen(infix);
     infix[len] = ')';
     infix[++len] = '\0';
     s.top = 0;
     s.items[s.top] = '(';

     for (cur = 0; (symb = infix[cur]) != '\0'; cur++)
     {
          while (f(symb) < g(s.items[s.top]))
               postr[i++] = Pop(&s);

          if (f(symb) != g(s.items[s.top]))
               Push(&s, symb);
          else
               Pop(&s);
     }
     postr[i] = '\0';
}

int Isoperand (char symb)
{
     if ((symb >= 'a') && (symb <= 'z'))
          return(1);
     else
          return(0);
}

/* Returns the incoming symbol precedence */
int f (char op1)
{
```

```
    if (Isoperand(op1))
        op1 = '#';
    switch (op1)
    {
        case '+':
        case '-':  return(1);
        case '*':
        case '/':  return(3);
        case '$':  return(6);
        case '#':  return(7);
        case '(':  return(9);
        case ')':  return(0);
        default :  printf("Error\n");
                   return (-1);
    }
}

/* Returns the Stack precedence */
int g (char op2)
{
    if (Isoperand(op2))
        op2 = '#';
    switch (op2)
    {
        case '+':
        case '-':  return(2);
        case '*':
        case '/':  return(4);
        case '$':  return(5);
        case '#':  return(8);
        case '(':  return(0);
        default :  printf("Error\n");
                   return (-1);
    }
}
```

3.4.4 Infix to Prefix conversion

LISP is a computer programming language, which uses prefix expression extensively. The objective of this section is to design an algorithm and C code to convert a given valid infix expression to prefix expression. The assumptions that has been specified in Section 3.3.3 for infix to suffix holds good for this problem also. So, we do not repeat them here.

The Method

The design is same as infix to suffix conversion, expect for the following changes:

(1) The input infix string is scanned from right to left.

(2) The precedence (stack and input symbol) table should be changed.

Since the input string is scanned from right to left, we may encounter ')' first and hence it is stored in the stack and when a matching '(' is encountered, symbols from stack are popped as long as the precedence of the incoming symbol is less than the stack top precedence.

Example 1 *Table 3.5*

Input infix string: A * B + C

Input symbol	Stack contents	Prefix string
-	Top→ %	-
C	Top→ C, %	-
+	Top→ +, %	C
B	Top→ B, +, %	C
*	Top→ *, +, %	B C
A	Top→ A, *, +, %	+ * A B C

The stack actually retains the operators and the main difference is that the symbols are added to the prefix output string from right to left. Two examples are shown in Table 3.5 and 3.6 that show the use of stack for this problem.

Let us consider another example expression that has no parentheses.

Example 2 *Table 3.6*

Input infix expression: A + B * C

Input symbol	Stack contents	Prefix string
C	Top→ %	-
*	Top→ * %	C
B	Top→ B * %	C
+	Top→ + %	* B C
A	Top→ A + %	+ A * B C

The symbol '%' is used to pop all the remaining elements from stack after the input expression is scanned and hence its precedence should be set to the lowest compared to all the other symbols (i.e., -1). The precedence table is shown in Table 3.7.

We will show (see Table 3.8) yet another example for conversion to show the behavior of the algorithm when parentheses are used.

In_Prefix() function

The C function to accomplish the infix to prefix conversion is shown in Program 3.6. The main for loop scans the infix expression from right most character using the index variable `pos = strlen(infix)-1`. The loop terminates when it reaches the first

character at position, `pos = 0`. The cursor for the `prefix` (output) string is initialized to the last position. If the infix string does not contain parenthesis, then the prefix string also will have the same length, otherwise it will have few characters less, because left and right parenthesis won't appear in prefix string.

Table 3.7

Symbol	Incoming symbol precedence (f)	Stack precedence (g)
+, -	2	1
*, /	4	3
$, ^	5	6
a, b, ...,z	7	8
(	0	-
)	9	0
%	-	-1

When the incoming symbol precedence is greater than the precedence of the stack top, the corresponding symbol is pushed in stack, else it is popped and added to `prestr` (i.e., when they differ in precedence value).

Program 3.6
Infix to Prefix Conversion

```
void In_Prefix (char infix[], char prefix[])
{
     int  pos;
     char prestr[MAX];
     int  outpos;
     char symb;
     int  i;
     struct stk s;
     outpos = strlen(infix);
     s.top = 0;
     s.items[s.top] = '%';

     for (pos = strlen(infix) - 1; pos >= 0; pos--)
     {
          symb = infix[pos];
          while (f(symb) < g(s.items[s.top]))
               prestr[--outpos] = Pop(&s);
          if (f(symb) != g(s.items[s.top]))
               Push(&s, symb);

          else
```

```
                Pop(&s);
        }
        while (s.items[s.top] != '%')
              prestr[--outpos] = Pop(&s);

        for (i = 0; outpos+1 <= strlen(infix);)
              prefix[i++] = prestr[outpos++];
        prefix[i] = '\0';
        Pop(&s);    /* remove '%' */
}
```

The remaining characters are copied to `prefix` until the special character % is reached in stack. This is done using a `while` loop. Since, prefix expressions are parentheses free, after the conversion, it should be copied from `prestr` string to `prefix` string starting from `outpos` (cursor for `prestr`). The following code does this:

```
for(i = 0; outpos + 1 <= strlen(infix);)
      prefix[i++] = prestr[outpos++];
```

The reader is advised to carefully go through Table 3.8 to understand this concept.

Example 3 Table 3.8

Infix expression: (A + B) * C

Incoming symbol	Stack contents	prestr
-	Top→ %	-
C	Top→ C, %	-
*	Top→ *, %	C
)	Top→), *, %	C

Symbol	Stack	prestr
B	Top→ B,), *, %	C
+	Top→ +,), *, %	B C
A	Top→ A, +,), *, %	B C
(	Top→ *, %	+ A B C ')' will go into prestr
while loop will remove *		* + A B C

The length of the infix expression is 7, where as the prefix string is of length 5 (because prefix is free from parentheses). The `prefix` string after the `while` loop is executed would look like

prestr:

0	1	2	3	4	5	6
		*	+	A	B	C

Using the last `for` loop, as explained already, the characters from position 2 to 6 will be copied to `prefix` from 0 to 4.

3.5 DOUBLE STACK – A SPECIAL STACK

The stacks that have been presented so far were all single stacks. It means that the elements that are pushed go into one stack only. This section addresses a new kind of stack, which is useful in compilers and it is called as a **double stack**.

In the array implementation of stack, we put an upper limit for the stack size, say MAX = 80. This means that you can have at most 80 elements only. When only few

elements are used, there is a vast wastage of memory. Pascal compiler, for instance, uses a different approach to save this memory by using two stacks in a single array. Instead of defining two separate stack variables, we can store these elements in a single array and operate from either ends.

Look at Figure 3.7 that shows the double stack and *top* pointers – *top1* and *top2*.

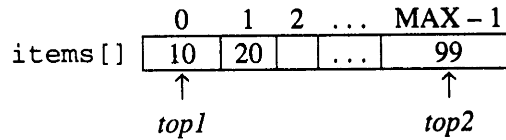

Fig 3.7 A double stack

Stack1 grows from left to right and Stack 2 grows from right to left. Since both the stacks need last inserted item to be remembered (i.e., top most element), two pointers *top1* and *top2* are required. The *top1* moves from 0 and *top2* from MAX – 1. Since, the template of the single stack can not be used for this problem, we shall show the modified one below:

```
struct double_stk
{
      int items[MAX];
      int top1;
      int top2;
};
typedef struct double_stk * Dstack;
```

The important constraint is that the elements of Stack1 and Stack2 should not get overlapped.

The function Push()

The function to accomplish a push operation to either Stack1 or Stack2 is shown in Program 3.7.

Program 3.7
Push for Double Stack

```
void Push (DStack ps, int x, int n)
{
    if (Full(ps))    /* Check for stack status */
    {
          printf("Error-overflow\n");
          return;
    }
    switch(n)
    {
```

```
            case 1: ++(ps->top1);
                    ps->items[ps->top1] = x; /* stack 1 */
                    break;
            case 2: --(ps->top2);
                    ps->items[ps->top2] = x; /* stack 2 */
                    break;
            default:printf("Invalid DStack id\n");
        }
    }
```

You can notice that there is an additional parameter `n` in addition to the usual parameters `ps` and `x`. The `Push()` function must know where `x` is to be inserted? Therefore, when you call `Push()` you must supply the stack number also. For example,

```
Push(&s, 10, 1);        // insert 10 to stack1
Push(&s, 99, 2);        // insert 99 to stack2
```

Normal stack increments `top` and then inserts element into stack. However, this cannot be followed for Stack2. The pointer `top2` should be decremented first and then inserted. Therefore, `top2` should be initialized to MAX.

That is, `top2 = MAX;`

Checking for stack overflow condition in the case of double stack is again different from ordinary stack and is explained later in this section.

The function Pop()

The function `Pop()` is shown as a C code in Program 3.8 and is almost similar to `Pop()` of single stack.

Program 3.7
Pop for Double Stack

```
int Pop (DStack ps, int n)
{
    switch (n)
    {
        case 1: if (!Empty(ps, n))
                    return (ps->items[ps->top1--]);
                break;
        case 2: if (!Empty(ps, n))
                    return (ps->items[ps->top2++]);
                break;
        default: printf("Invalid DStack id\n");
    }
```

```
    return (-1);
}
```

You may very easily observe that there is again an extra parameter n to indicate the stack number. If n = 1, the element will be popped from Stack1, else if n = 2, it will be from Stack2. After popping from Stack1, top1 is decremented (normal stack) but popping from Stack2 leads to incrementing top2. Because Stack2 shrinks towards MAX (or right side), we need to check for underflow error condition and is explained later in this section.

The function Full()

When do we say that the double stack is full? The design approach is generic; it means that we write a common Full() function applicable for both the stacks. If we keep on inserting it would go only to Stack1 and therefore top1 may reach to MAX - 1 and is full. The other possibility is that if we keep on inserting to Stack2, then top2 may reach 0, and it is again full. The third case is interesting and is explained with the help of Figure 3.8. Assume, MAX = 4.

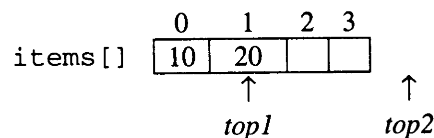

Fig. 3.8(a) After pushing 10 and 20

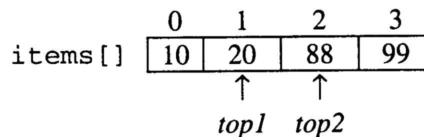

Fig. 3.8(b) After pushing 99 and 88

In Figure 3.8(b), the status of stack is full. That is, when top1 = top2-1 the stack is said to be full and returns 1, else return 0. The function Full() is shown in Program 3.9.

Program 3.9
Stack status - Full

```
int Full (DStack ps)
{ /* return 1 when full, else 0 */
    if (ps->top1 == MAX-1 || ps->top2 == 0
            || ps->top1 == ps->top2-1)
        return 1;
```

```
     else return 0;
}
```

The function Empty()

The design of `Empty()` requires little thinking and is not same as `Full()`. You may wrongly write as,

```
/* Wrong code */
if((ps->top != -1) && (ps->top2 == MAX))
            return 1;
else return 0;
```

The logical operator && is wrongly used, because the function will return *true* (1) only when both the stacks are empty. However, we must also take care of the situation where even if either Stack1 or Stack2 is empty, the error should be reported. This code cannot do that. Even if you replace the logical operator && to ||, the problem is not solved. The solution is to add a parameter `n` to check the empty status of a particular stack, say,

```
if(!Empty(ps, 1))
...
```

The correct C code is shown in Program 3.10.

Program 3.10
Stack status – Empty

```
int Empty (DStack ps, int n)
{ /* return 1 when empty, else 0 */
     switch(n)
     {
          case 1: if (ps->top1 == -1)
                    return 1; else return 0;
          case 2: if (ps->top2 == MAX)
                    return 1; else return 0;
     }
}
```