4.2. Algebraic Expressions:

An algebraic expression is a legal combination of operators and operands. Operand is the quantity on which a mathematical operation is performed. Operand may be a variable like x, y, z or a constant like 5, 4, 6 etc. Operator is a symbol which signifies a mathematical or logical operation between the operands. Examples of familiar operators include +, -, *, /, ^ etc.

An algebraic expression can be represented using three different notations. They are infix, postfix and prefix notations:

Infix: It is the form of an arithmetic expression in which we fix (place) the arithmetic operator in between the two operands.

Example: (A + B) * (C - D)

Prefix: It is the form of an arithmetic notation in which we fix (place) the arithmetic operator before (pre) its two operands. The prefix notation is called as

polish notation (due to the polish mathematician Jan Lukasiewicz in the year 1920).

Example: * + A B – C D

Postfix: It is the form of an arithmetic expression in which we fix (place) the arithmetic operator after (post) its two operands. The postfix notation is called as *suffix notation* and is also referred to *reverse polish notation*.

Example: A B + C D - *

The three important features of postfix expression are:

1. The operands maintain the same order as in the equivalent infix expression.
2. The parentheses are not needed to designate the expression un-ambiguously.
3. While evaluating the postfix expression the priority of the operators is no longer relevant.

We consider five binary operations: +, -, *, / and $ or ↑ (exponentiation). For these binary operations, the following in the order of precedence (highest to lowest):

OPERATOR	PRECEDENCE	VALUE
Exponentiation ($ or ↑ or ^)	Highest	3
*, /	Next highest	2
+, -	Lowest	1

4.3. Converting expressions using Stack:

Let us convert the expressions from one type to another. These can be done as follows:

1. Infix to postfix
2. Infix to prefix
3. Postfix to infix
4. Postfix to prefix
5. Prefix to infix
6. Prefix to postfix

4.3.1. Conversion from infix to postfix:

Procedure to convert from infix expression to postfix expression is as follows:

1. Scan the infix expression from left to right.
2. a) If the scanned symbol is left parenthesis, push it onto the stack.

   b) If the scanned symbol is an operand, then place directly in the postfix expression (output).

c) If the symbol scanned is a right parenthesis, then go on popping all the items from the stack and place them in the postfix expression till we get the matching left parenthesis.

d) If the scanned symbol is an operator, then go on removing all the operators from the stack and place them in the postfix expression, if and only if the precedence of the operator which is on the top of the stack is greater than (*or greater than or equal*) to the precedence of the scanned operator and push the scanned operator onto the stack otherwise, push the scanned operator onto the stack.

Example 1:

Convert ((A – (B + C)) * D) ↑ (E + F) infix expression to postfix form:

SYMBOL	POSTFIX STRING	STACK	REMARKS
(		(	
(		((	
A	A	((	
-	A	((-	
(	A	((- (	
B	A B	((- (	
+	A B	((- (+	
C	A B C	((- (+	
)	A B C +	((-	
)	A B C + -	(	
*	A B C + -	(*	
D	A B C + - D	(*	
)	A B C + - D *		
↑	A B C + - D *	↑	
(	A B C + - D *	↑ (	
E	A B C + - D * E	↑ (	
+	A B C + - D * E	↑ (+	
F	A B C + - D * E F	↑ (+	
)	A B C + - D * E F +	↑	
End of string	A B C + - D * E F + ↑	The input is now empty. Pop the output symbols from the stack until it is empty.	

Example 2:

Convert a + b * c + (d * e + f) * g the infix expression into postfix form.

SYMBOL	POSTFIX STRING	STACK	REMARKS
a	a		
+	a	+	
b	a b	+	

*	a b	+ *	
c	a b c	+ *	
+	a b c * +	+	
(	a b c * +	+ (	
d	a b c * + d	+ (	
*	a b c * + d	+ (*	
e	a b c * + d e	+ (*	
+	a b c * + d e *	+ (+	
f	a b c * + d e * f	+ (+	
)	a b c * + d e * f +	+	
*	a b c * + d e * f +	+ *	
g	a b c * + d e * f + g	+ *	
End of string	a b c * + d e * f + g * +	The input is now empty. Pop the output symbols from the stack until it is empty.	

Example 3:

Convert the following infix expression A + B * C – D / E * H into its equivalent postfix expression.

SYMBOL	POSTFIX STRING	STACK	REMARKS
A	A		
+	A	+	
B	A B	+	
*	A B	+ *	
C	A B C	+ *	
-	A B C * +	-	
D	A B C * + D	-	
/	A B C * + D	- /	
E	A B C * + D E	- /	
*	A B C * + D E /	- *	
H	A B C * + D E / H	- *	
End of string	A B C * + D E / H * -	The input is now empty. Pop the output symbols from the stack until it is empty.	

Example 4:

Convert the following infix expression A + (B * C – (D / E ↑ F) * G) * H into its equivalent postfix expression.

SYMBOL	POSTFIX STRING	STACK	REMARKS
A	A		
+	A	+	

(	A	+ (	
B	A B	+ (	
*	A B	+ (*	
C	A B C	+ (*	
-	A B C *	+ (-	
(	A B C *	+ (- (	
D	A B C * D	+ (- (	
/	A B C * D	+ (- (/	
E	A B C * D E	+ (- (/	
↑	A B C * D E	+ (- (/ ↑	
F	A B C * D E F	+ (- (/ ↑	
)	A B C * D E F ↑ /	+ (-	
*	A B C * D E F ↑ /	+ (- *	
G	A B C * D E F ↑ / G	+ (- *	
)	A B C * D E F ↑ / G * -	+	
*	A B C * D E F ↑ / G * -	+ *	
H	A B C * D E F ↑ / G * - H	+ *	
End of string	A B C * D E F ↑ / G * - H * +	The input is now empty. Pop the output symbols from the stack until it is empty.	

4.3.2. Program to convert an infix to postfix expression:

```
# include <string.h>

char postfix[50];
char infix[50];
char opstack[50];                    /* operator stack */
int i, j, top = 0;

int lesspriority(char op, char op_at_stack)
{
        int k;
        int pv1;                    /* priority value of op  */
        int pv2;                    /* priority value of op_at_stack */
        char operators[] = {'+', '-', '*', '/', '%', '^', '(' };
        int priority_value[] = {0,0,1,1,2,3,4};
        if( op_at_stack == '(' )
                return 0;
        for(k = 0; k < 6; k ++)
        {
                if(op == operators[k])
                        pv1 = priority_value[k];
        }
        for(k = 0; k < 6; k ++)
        {
                if(op_at_stack == operators[k])
                        pv2 = priority_value[k];
        }
        if(pv1 < pv2)
                return 1;
        else
                return 0;
}
```

```
void push(char op)        /* op - operator  */
{
        if(top == 0)
        {
                opstack[top] = op;
                top++;
        }
        else
        {
                if(op != '(' )
                {
                        while(lesspriority(op, opstack[top-1]) == 1 && top > 0)
                        {
                                        postfix[j] = opstack[--top];
                                        j++;
                        }
                }
                opstack[top] = op;      /* pushing onto stack */
                top++;
        }
}

pop()
{
        while(opstack[--top] != '(' )              /* pop until '(' comes  */
        {
                postfix[j] = opstack[top];
                j++;
        }
}

void main()
{
        char ch;
        clrscr();
        printf("\n Enter Infix Expression : ");
        gets(infix);
        while( (ch=infix[i++]) != '\0')
        {
                switch(ch)
                {
                        case ' ' : break;
                        case '(' :
                        case '+' :
                        case '-' :
                        case '*' :
                        case '/' :
                        case '^' :
                        case '%' :
                                push(ch);                /* check priority and push */
                                break;
                        case ')' :
                                pop();
                                break;
                        default :
                                postfix[j] = ch;
                                j++;
                }
        }
        while(top >= 0)
        {
                postfix[j] = opstack[--top];
                j++;
```

/* before pushing the operator 'op' into the stack check priority of op with top of opstack if less then pop the operator from stack then push into postfix string else push op onto stack itself */

```
        }
        postfix[j] = '\0';
        printf("\n Infix  Expression    : %s ", infix);
        printf("\n Postfix Expression : %s ", postfix);
        getch();
}
```

4.3.3. Conversion from infix to prefix:

The precedence rules for converting an expression from infix to prefix are identical. The only change from postfix conversion is that traverse the expression from right to left and the operator is placed before the operands rather than after them. The prefix form of a complex expression is not the mirror image of the postfix form.

Example 1:

Convert the infix expression A + B - C into prefix expression.

SYMBOL	PREFIX STRING	STACK	REMARKS
C	C		
-	C	-	
B	B C	-	
+	B C	- +	
A	A B C	- +	
End of string	- + A B C	The input is now empty. Pop the output symbols from the stack until it is empty.	

Example 2:

Convert the infix expression (A + B) * (C - D) into prefix expression.

SYMBOL	PREFIX STRING	STACK	REMARKS
)		)	
D	D	)	
-	D	) -	
C	C D	) -	
(	- C D		
*	- C D	*	
)	- C D	*)	
B	B - C D	*)	
+	B - C D	*) +	
A	A B - C D	*) +	
(	+ A B – C D	*	
End of string	* + A B – C D	The input is now empty. Pop the output symbols from the stack until it is empty.	

Example 3:

Convert the infix expression A ↑ B * C – D + E / F / (G + H) into prefix expression.

SYMBOL	PREFIX STRING	STACK	REMARKS
)		)	
H	H	)	
+	H	) +	
G	G H	) +	
(	+ G H		
/	+ G H	/	
F	F + G H	/	
/	F + G H	/ /	
E	E F + G H	/ /	
+	/ / E F + G H	+	
D	D / / E F + G H	+	
-	D / / E F + G H	+ -	
C	C D / / E F + G H	+ -	
*	C D / / E F + G H	+ - *	
B	B C D / / E F + G H	+ - *	
↑	B C D / / E F + G H	+ - * ↑	
A	A B C D / / E F + G H	+ - * ↑	
End of string	+ - * ↑ A B C D / / E F + G H	The input is now empty. Pop the output symbols from the stack until it is empty.	

4.3.4. Program to convert an infix to prefix expression:

```
# include <conio.h>
# include <string.h>

char prefix[50];
char infix[50];
char opstack[50];                    /* operator stack */
int j, top = 0;

void insert_beg(char ch)
{
        int k;
        if(j == 0)
                prefix[0] =  ch;
        else
        {
                for(k = j + 1; k > 0; k--)
                        prefix[k] = prefix[k - 1];
                prefix[0] = ch;
        }
        j++;
}
```

```
int lesspriority(char op, char op_at_stack)
{
	int k;
	int pv1;			/* priority value of op  */
	int pv2;			/* priority value of op_at_stack */
	char operators[] = {'+', '-', '*', '/', '%', '^', ')'};
	int priority_value[] = {0, 0, 1, 1, 2, 3, 4};
	if(op_at_stack == ')' )
		return 0;
	for(k = 0; k < 6; k ++)
	{
		if(op == operators[k])
			pv1 = priority_value[k];
	}
	for(k = 0; k < 6; k ++)
	{
		if( op_at_stack == operators[k] )
			pv2 = priority_value[k];
	}
	if(pv1 < pv2)
		return 1;
	else
		return 0;
}


void push(char op)		/* op – operator */
{
	if(top == 0)
	{
		opstack[top] = op;
		top++;
	}
	else
	{
		if(op != ')')
		{
			/* before pushing the operator 'op' into the stack check priority of op with
		top of operator stack if less pop the operator from stack then push into postfix
		string else push op onto stack itself  */

			while(lesspriority(op, opstack[top-1]) == 1 && top > 0)
			{
					insert_beg(opstack[--top]);
			}
		}
		opstack[top] = op;			/* pushing onto stack */
		top++;
	}
}

void pop()
{
	while(opstack[--top] != ')')			/* pop until ')' comes; */
		insert_beg(opstack[top]);
}


void main()
{
	char ch;
	int l, i = 0;
	clrscr();
	printf("\n Enter Infix Expression : ");
```

```
        gets(infix);
        l = strlen(infix);
        while(l > 0)
        {
                ch = infix[--l];
                switch(ch)
                {
                        case ' ' : break;
                        case ')' :
                        case '+' :
                        case '-' :
                        case '*' :
                        case '/' :
                        case '^' :
                        case '%' :
                                push(ch);               /* check priority and push */
                                break;
                        case '(' :
                                pop();
                                break;
                        default :
                                insert_beg(ch);
                }
        }
        while( top > 0 )
        {
                insert_beg( opstack[--top] );
                j++;
        }
        prefix[j] = '\0';
        printf("\n Infix Expression    : %s ", infix);
        printf("\n Prefix Expression : %s ", prefix);
        getch();
}
```

4.3.5. Conversion from postfix to infix:

Procedure to convert postfix expression to infix expression is as follows:

1. Scan the postfix expression from left to right.
2. If the scanned symbol is an operand, then push it onto the stack.
3. If the scanned symbol is an operator, pop two symbols from the stack and create it as a string by placing the operator in between the operands and push it onto the stack.
4. Repeat steps 2 and 3 till the end of the expression.

Example:

Convert the following postfix expression A B C * D E F ^ / G * - H * + into its equivalent infix expression.

Symbol	Stack	Remarks
A	A	Push A
B	A B	Push B
C	A B C	Push C
*	A (B*C)	Pop two operands and place the operator in between the operands and push the string.
D	A (B*C) D	Push D
E	A (B*C) D E	Push E
F	A (B*C) D E F	Push F
^	A (B*C) D (E^F)	Pop two operands and place the operator in between the operands and push the string.
/	A (B*C) (D/(E^F))	Pop two operands and place the operator in between the operands and push the string.
G	A (B*C) (D/(E^F)) G	Push G
*	A (B*C) ((D/(E^F))*G)	Pop two operands and place the operator in between the operands and push the string.
-	A ((B*C) – ((D/(E^F))*G))	Pop two operands and place the operator in between the operands and push the string.
H	A ((B*C) – ((D/(E^F))*G)) H	Push H
*	A (((B*C) – ((D/(E^F))*G)) * H)	Pop two operands and place the operator in between the operands and push the string.
+	(A + (((B*C) – ((D/(E^F))*G)) * H))	
End of string	The input is now empty. The string formed is infix.	

4.3.6. Program to convert postfix to infix expression:

```
# include <stdio.h>
# include <conio.h>
# include <string.h>
# define MAX 100

void pop (char*);
void push(char*);

char stack[MAX] [MAX];
int top = -1;
```

```
void main()
{
        char s[MAX], str1[MAX], str2[MAX], str[MAX];
        char s1[2],temp[2];
        int i=0;
        clrscr( ) ;
        printf("\Enter the postfix expression; ");
        gets(s);
        while (s[i]!='\0')
        {
                if(s[i] == ' ' )                        /*skip whitespace, if any*/
                        i++;
                if (s[i] == '^' || s[i] == '*'|| s[i] == '-' || s[i] == '+' || s[i] == '/')
                {
                        pop(str1);
                        pop(str2);
                        temp[0] ='(';
                        temp[1] ='\0';
                        strcpy(str, temp);
                        strcat(str, str2);
                        temp[0] = s[i];
                        temp[1] = '\0';
                        strcat(str,temp);
                        strcat(str, str1);
                        temp[0] =')';
                        temp[1] ='\0';
                        strcat(str,temp);
                        push(str);
                }
                else
                {
                        temp[0]=s[i];
                        temp[1]='\0';
                        strcpy(s1, temp);
                        push(s1);
                }
                i++;
        }

        printf("\nThe Infix expression is: %s", stack[0]);

}

void pop(char *a1)
{
        strcpy(a1,stack[top]);
        top--;
}

void push (char*str)
{
        if(top == MAX - 1)
                printf("\nstack is full");
        else
        {
                top++;
                strcpy(stack[top], str);
        }
}
```

4.3.7. Conversion from postfix to prefix:

Procedure to convert postfix expression to prefix expression is as follows:

1. Scan the postfix expression from left to right.
2. If the scanned symbol is an operand, then push it onto the stack.
3. If the scanned symbol is an operator, pop two symbols from the stack and create it as a string by placing the operator in front of the operands and push it onto the stack.
5. Repeat steps 2 and 3 till the end of the expression.

Example:

Convert the following postfix expression A B C * D E F ^ / G * - H * + into its equivalent prefix expression.

Symbol	Stack	Remarks
A	A	Push A
B	A B	Push B
C	A B C	Push C
*	A *BC	Pop two operands and place the operator in front the operands and push the string.
D	A *BC D	Push D
E	A *BC D E	Push E
F	A *BC D E F	Push F
^	A *BC D ^EF	Pop two operands and place the operator in front the operands and push the string.
/	A *BC /D^EF	Pop two operands and place the operator in front the operands and push the string.
G	A *BC /D^EF G	Push G
*	A *BC */D^EFG	Pop two operands and place the operator in front the operands and push the string.
-	A - *BC*/D^EFG	Pop two operands and place the operator in front the operands and push the string.
H	A - *BC*/D^EFG H	Push H
*	A *- *BC*/D^EFGH	Pop two operands and place the operator in front the operands and push the string.
+	+A*-*BC*/D^EFGH	
End of string	The input is now empty. The string formed is prefix.	

4.3.8. Program to convert postfix to prefix expression:

```
# include <conio.h>
# include <string.h>

#define MAX 100
void pop (char *a1);
void push(char *str);
char stack[MAX][MAX];
int top =-1;

main()
{
        char s[MAX], str1[MAX], str2[MAX], str[MAX];
        char s1[2], temp[2];
        int i = 0;
        clrscr();
        printf("Enter the postfix expression; ");
        gets (s);
        while(s[i]!='\0')
        {
        /*skip whitespace, if any */
                if (s[i] == ' ')
                        i++;
                if(s[i] == '^' || s[i] == '*' || s[i] == '-' || s[i]== '+' || s[i] == '/')
                {
                        pop (str1);
                        pop (str2);
                        temp[0] = s[i];
                        temp[1] = '\0';
                        strcpy (str, temp);
                        strcat(str, str2);
                        strcat(str, str1);
                        push(str);
                }
                else
                {
                        temp[0] = s[i];
                        temp[1] = '\0';
                        strcpy (s1, temp);
                        push (s1);
                }
                i++;
        }

        printf("\n The prefix expression is: %s", stack[0]);
}

void pop(char*a1)
{
        if(top == -1)
        {
                printf("\nStack is empty");
                return ;
        }
        else
        {
                strcpy (a1, stack[top]);
                top--;
        }
}
```

```
void push (char *str)
{
        if(top == MAX - 1)
                printf("\nstack is full");
        else
        {
                top++;
                strcpy(stack[top], str);
        }
}
```

4.3.9. Conversion from prefix to infix:

Procedure to convert prefix expression to infix expression is as follows:

1. Scan the prefix expression from right to left (reverse order).
2. If the scanned symbol is an operand, then push it onto the stack.
3. If the scanned symbol is an operator, pop two symbols from the stack and create it as a string by placing the operator in between the operands and push it onto the stack.
4. Repeat steps 2 and 3 till the end of the expression.

Example:

Convert the following prefix expression + A * - * B C * / D ^ E F G H into its equivalent infix expression.

Symbol	Stack	Remarks
H	H	Push H
G	H | G	Push G
F	H | G | F	Push F
E	H | G | F | E	Push E
^	H | G | (E^F)	Pop two operands and place the operator in between the operands and push the string.
D	H | G | (E^F) | D	Push D
/	H | G | (D/(E^F))	Pop two operands and place the operator in between the operands and push the string.
*	H | ((D/(E^F))*G)	Pop two operands and place the operator in between the operands and push the string.
C	H | ((D/(E^F))*G) | C	Push C
B	H | ((D/(E^F))*G) | C | B	Push B
*	H | ((D/(E^F))*G) | (B*C)	Pop two operands and place the operator in front the operands and push the string.
-	H | ((B*C)-((D/(E^F))*G))	Pop two operands and place the operator in front the operands and push the

		string.
*	(((B*C)-((D/(E^F))*G))*H)	Pop two operands and place the operator in front the operands and push the string.
A	(((B*C)-((D/(E^F))*G))*H) | A	Push A
+	(A+(((B*C)-((D/(E^F))*G))*H))	Pop two operands and place the operator in front the operands and push the string.
End of string	The input is now empty. The string formed is infix.	

4.3.10. Program to convert prefix to infix expression:

```
# include <string.h>
# define MAX 100

void pop (char*);
void push(char*);
char stack[MAX] [MAX];
int top = -1;

void main()
{
        char s[MAX], str1[MAX], str2[MAX], str[MAX];
        char s1[2],temp[2];
        int i=0;
        clrscr( ) ;
        printf("\Enter the prefix expression; ");
        gets(s);
        strrev(s);
        while (s[i]!='\0')
        {
                /*skip whitespace, if any*/
                if(s[i] == ' ' )
                        i++;
                if (s[i] == '^' || s[i] == '*'|| s[i] == '-' || s[i] == '+' || s[i] == '/')
                {
                        pop(str1);
                        pop(str2);
                        temp[0] ='(';
                        temp[1] ='\0';
                        strcpy(str, temp);
                        strcat(str, str1);
                        temp[0] = s[i];
                        temp[1] = '\0';
                        strcat(str,temp);
                        strcat(str, str2);
                        temp[0] =')';
                        temp[1] ='\0';
                        strcat(str,temp);
                        push(str);
                }
                else
                {
                        temp[0]=s[i];
                        temp[1]='\0';
                        strcpy(s1, temp);
                        push(s1);
```

```
                }
                i++;
        }
        printf("\nThe infix expression is: %s", stack[0]);
}

void pop(char *a1)
{
        strcpy(a1,stack[top]);
        top--;
}

void push (char*str)
{
        if(top == MAX - 1)
                printf("\nstack is full");
        else
        {
                top++;
                strcpy(stack[top], str);
        }
}
```

4.3.11. Conversion from prefix to postfix:

Procedure to convert prefix expression to postfix expression is as follows:

1. Scan the prefix expression from right to left (reverse order).
2. If the scanned symbol is an operand, then push it onto the stack.
3. If the scanned symbol is an operator, pop two symbols from the stack and create it as a string by placing the operator after the operands and push it onto the stack.
4. Repeat steps 2 and 3 till the end of the expression.

Example:

Convert the following prefix expression + A * - * B C * / D ^ E F G H into its equivalent postfix expression.

Symbol	Stack	Remarks
H	H | | | | |	Push H
G	H | G | | | |	Push G
F	H | G | F | | |	Push F
E	H | G | F | E | |	Push E
^	H | G | EF^ | |	Pop two operands and place the operator after the operands and push the string.
D	H | G | EF^ | D | |	Push D

/	H | G | DEF^/ |	Pop two operands and place the operator after the operands and push the string.
*	H | DEF^/G* |	Pop two operands and place the operator after the operands and push the string.
C	H | DEF^/G* | C |	Push C
B	H | DEF^/G* | C | B |	Push B
*	H | DEF^/G* | BC* |	Pop two operands and place the operator after the operands and push the string.
-	H | BC*DEF^/G*- |	Pop two operands and place the operator after the operands and push the string.
*	BC*DEF^/G*-H* |	Pop two operands and place the operator after the operands and push the string.
A	BC*DEF^/G*-H* | A |	Push A
+	ABC*DEF^/G*-H*+ |	Pop two operands and place the operator after the operands and push the string.
End of string	The input is now empty. The string formed is postfix.	

4.3.12. Program to convert prefix to postfix expression:

```
# include <stdio.h>
# include <conio.h>
# include <string.h>

#define MAX 100

void pop (char *a1);
void push(char *str);
char stack[MAX][MAX];
int top =-1;

void main()
{
        char s[MAX], str1[MAX], str2[MAX], str[MAX];
        char s1[2], temp[2];
        int i = 0;
        clrscr();
        printf("Enter the prefix expression; ");
        gets (s);
        strrev(s);
        while(s[i]!='\0')
        {
                if (s[i] == ' ')                        /*skip whitespace, if any */
                        i++;
                if(s[i] == '^' || s[i] == '*' || s[i] == '-' || s[i]== '+' || s[i] == '/')
                {
                        pop (str1);
                        pop (str2);
                        temp[0] = s[i];
                        temp[1] = '\0';
                        strcat(str1,str2);
                        strcat (str1, temp);
                        strcpy(str, str1);
                        push(str);
                }
```

```
                else
                {
                        temp[0] = s[i];
                        temp[1] = '\0';
                        strcpy (s1, temp);
                        push (s1);
                }
                i++;
        }

        printf("\nThe postfix expression is: %s", stack[0]);
}
void pop(char*a1)
{
        if(top == -1)
        {
                printf("\nStack is empty");
                return ;
        }
        else
        {
                strcpy (a1, stack[top]);
                top--;
        }
}

void push (char *str)
{
        if(top == MAX - 1)
                printf("\nstack is full");
        else
        {
                top++;
                strcpy(stack[top], str);
        }
}
```

4.4. Evaluation of postfix expression:

The postfix expression is evaluated easily by the use of a stack. When a number is seen, it is pushed onto the stack; when an operator is seen, the operator is applied to the two numbers that are popped from the stack and the result is pushed onto the stack. When an expression is given in postfix notation, there is no need to know any precedence rules; this is our obvious advantage.

Example 1:

Evaluate the postfix expression: 6 5 2 3 + 8 * + 3 + *

SYMBOL	OPERAND 1	OPERAND 2	VALUE	STACK	REMARKS
6				6	
5				6, 5	
2				6, 5, 2	
3				6, 5, 2, 3	The first four symbols are placed on the stack.
+	2	3	5	6, 5, 5	Next a '+' is read, so 3 and 2 are popped from the stack and their sum 5, is pushed

8	2	3	5	6, 5, 5, 8	Next 8 is pushed
*	5	8	40	6, 5, 40	Now a '*' is seen, so 8 and 5 are popped as 8 * 5 = 40 is pushed
+	5	40	45	6, 45	Next, a '+' is seen, so 40 and 5 are popped and 40 + 5 = 45 is pushed
3	5	40	45	6, 45, 3	Now, 3 is pushed
+	45	3	48	6, 48	Next, '+' pops 3 and 45 and pushes 45 + 3 = 48 is pushed
*	6	48	288	288	Finally, a '*' is seen and 48 and 6 are popped, the result 6 * 48 = 288 is pushed

Example 2:

Evaluate the following postfix expression: 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

SYMBOL	OPERAND 1	OPERAND 2	VALUE	STACK
6				6
2				6, 2
3				6, 2, 3
+	2	3	5	6, 5
-	6	5	1	1
3	6	5	1	1, 3
8	6	5	1	1, 3, 8
2	6	5	1	1, 3, 8, 2
/	8	2	4	1, 3, 4
+	3	4	7	1, 7
*	1	7	7	7
2	1	7	7	7, 2
↑	7	2	49	49
3	7	2	49	49, 3
+	49	3	52	52

4.4.1. Program to evaluate a postfix expression:

```
# include <conio.h>
# include <math.h>
# define MAX 20

int isoperator(char ch)
{
        if(ch == '+' || ch == '-' || ch == '*' || ch == '/' || ch == '^')
                return 1;
        else
                return 0;
}
```

```
void main(void)
{
	char postfix[MAX];
	int val;
	char ch;
	int i = 0, top = 0;
	float val_stack[MAX], val1, val2, res;
	clrscr();
	printf("\n Enter a postfix expression: ");
	scanf("%s", postfix);
	while((ch = postfix[i]) != '\0')
	{
		if(isoperator(ch) == 1)
		{
			val2 = val_stack[--top];
			val1 = val_stack[--top];
			switch(ch)
			{
				case '+':
					res = val1 + val2;
					break;
				case '-':
					res = val1 - val2;
					break;
				case '*':
					res = val1 * val2;
					break;
				case '/':
					res = val1 / val2;
					break;
				case '^':
					res = pow(val1, val2);
					break;
			}
			val_stack[top] = res;
		}
		else
			val_stack[top] = ch-48; /*convert character digit to integer digit */
		top++;
		i++;
	}
```

```
        printf("\n Values of %s is : %f ",postfix,
        val_stack[0] ); getch();
}
```

4.5. Applications of stacks:

1. Stack is used by compilers to check for balancing of parentheses, brackets and braces.

2. Stack is used to evaluate a postfix expression.

3. Stack is used to convert an infix expression into postfix/prefix form.

4. In recursion, all intermediate arguments and return values are stored on the processor's stack.

5. During a function call the return address and arguments are pushed onto a stack and on return they are popped off.