Exercise-1: Develop the code for converting the given infix expression to postfix expression.

Aim: To convert the infix expression into postfix expression.

Theory:

Postfix expression is one of the intermediate codes to be used in the translation of source code into machine code. For example, if an expression is like “a+b*c”, it is converted into “abc*+”. In postfix expression, the operators are moved to the position that is next to the required operands. In order to convert the infix to postfix expression, three data structure are used: (i) a stack, (ii) input buffer, and (iii) output buffer. During the evaluation of the expression some portion of the expressions are evaluated first and the other portions are evaluated next, depending on the priority of the operator.

Steps:

1. Create buffer (array) to hold both infix and postfix expression.
2. Create an operator stack.
3. Fix priorities for the operators in the infix string and stack.
4. For every operand or operator read from the infix expression string do step 5.
5. If it is operand place it in postfix string

   Else either push the operator on to stack or move the contents of stack to postfix string depending on the priority of the operator compared with the priority of the stack operator.

Modules:

- Stack related operations
  - Creation and disposal of stack
  - Checking the empty and full status of the stack for pop and push operations
  - Push, Pop, Peep at top operations

- Determining the priority of an operator
- Deciding whether the element is operator or operand
- Conversion of infix to postfix expression

Required data structure:

- Buffer for infix and postfix expression
- Operator array
- Infix and stack priority of the operators

Algorithm:

The following algorithm outlines the main procedure of converting the infix to postfix expression.

```
Algorithm infix2postfix()
//Input: Infix expression
//Output: Postfix expression
{
      defineOperatorPriorities();
      x=readInfix();
      if(isOperator(x))
      {
            ip=findInfixPriority()
            sp=findStackPriority()
            if(ip > sp)
                  push(stack,x);
            else
            {
```

```
                while(sp > ip)
                      //Keep moving operator y on stack to postfix string
                      add(postfix,y) //y should not be any parenthesis
                push(s,x);
          }
    }
    else
          add(postfix,x)

}
```

The following C program code list consists of:

- Includes and macro definitions
- Data declarations
- Function prototype declarations
- Definitions of the functions
- Main function

Following the code list, the outputs for the various test cases are presented.

```
/* Program listing */
```

//Include and macro defintions

```
#include<stdio.h>
#include<stdlib.h>
#include<malloc.h>
```

```
#include<string.h>
#define EMPTYSTACK -1
#define MAXSIZE 50
```

//Data Declarations

```
typedef char ElementType;
typedef struct StackRecord
{
      int capacity;
      int top;
      ElementType *array;
}StackRecord;

typedef StackRecord *Stack;

ElementType op[]={'*','+','&','|','(',')','#'};
int sp[]=        {5,   4,  3,  2,  1,  -1,-2};
int ip[]=        {5,   4,  3,  2,  5,  0, -1};
```

//Function prototype declarations

```
void push(Stack s,ElementType x);
ElementType top(Stack s);
void pop(Stack s);
ElementType topAndPop(Stack s);
Stack createStack(int mexLength);
```

```
void disposeStack(Stack s);
int checkEmpty(Stack s);
int checkFull(Stack s);

int prior(ElementType x,int y);
int isOperator(ElementType x);
ElementType* in2post(ElementType *infix);
```

//Function Definitions

```
Stack createStack(int maxElements)
{
      Stack s;
      s=(struct StackRecord*)malloc(sizeof(struct StackRecord));
      s->capacity=maxElements;
      s->array=(ElementType *)malloc(MAXSIZE*sizeof(ElementType));
      s->top=0;
      s->array[0]='#';
      return s;
}

void push(Stack s,ElementType x)
{
      s->top=s->top+1;
      s->array[s->top]=x;
}

ElementType top(Stack s)
```

```
{
      ElementType x;
      x=s->array[s->top];
      return x;
}

void pop(Stack s)
{
      s->top=s->top-1;
}

ElementType topAndPop(Stack s)
{
      ElementType x;
      x=s->array[s->top];
      s->top=s->top-1;
      return x;
}

int checkEmpty(Stack s)
{
      if(s->top==0||s->top==0)
      {
            return -1;
                                                                  }
      return 1;
}

int checkFull(Stack s)
```

```
{
      if(s->top==s->capacity)
      {
            return 1;
      }
    return 0;
}

int prior(ElementType x,int y)
{
      int pt;
      int ii;

      for(ii=0;ii<7;ii++)
      {
            if(x==op[ii])
            {
                  if(y==0)
                  {
                        pt=sp[ii];
                  }
                  else
                  {
                        pt=ip[ii];
                  }
            }
      }
      return pt;
}
```

```
int isOperator(ElementType x)
{

      int ck=0,ii;
      for(ii=0;ii<7;ii++)
      {
            if(x==op[ii])
            {
                  ck=1;
                  return ck;
            }
      }
      return ck;
}

ElementType* in2post(ElementType *infix)
{
    int i,j,maxElements;
    ElementType *postfix;
    ElementType x,y,ip,sp,h;
    Stack s;
    maxElements=strlen(infix);
    postfix = (ElementType*)malloc(maxElements*sizeof(ElementType));
    if(postfix == NULL)
    {
             printf("Memory allocation problem\n");
             exit(-1);
    }
```

```
s=createStack(maxElements);
for(i=0;i<maxElements;i++)
{
      x=infix[i];
      if(isOperator(x))
      {
            y=top(s);
            ip=prior(x,1);   //Infix priority
            sp=prior(y,0);   //stack priority
            if(ip > sp)      //ip > SP
            {
                  push(s,x);
            }
            else
            {
                  while(sp > ip)
                  {
                        y=top(s);
                        pop(s);
                        sp=prior(y,0);
                        if(y!=')'&& y!='(')
                        {
                              postfix[j++]=y;
                        }
                        h=checkEmpty(s);
                        if(h==-1)
                                    break;
                  }
                  push(s,x);
```

```
                  }
            }
            else
            {
                  postfix[j++]=x;
            }
      }
      postfix[j]='\0';
      return postfix;

}
main()
{
      ElementType infix[MAXSIZE],*postfix;
      printf("\nEnter the infix expression\t");
      scanf("%s",infix);
      postfix=in2post(infix);
      printf("infix: %s is converted into postfix:%s\n",infix,postfix);
}
```

Output

Test case1:

```
Enter the infix expression      a+b*c#
infix: a+b*c# is converted into postfix:abc*+
```

Test case2:

```
Enter the infix expression      a*b+c#
infix: a*b+c# is converted into postfix:ab*c+
```

Test case3:

```
Enter the infix expression      a+b*(c+d)#
infix: a+b*(c+d)# is converted into postfix:abcd+*+
```

Test case4:

```
Enter the infix expression      (a+c)*b+d#
infix: (a+c)*b+d is converted into postfix:ac+b*d+
```

Additional Exercises to students

The above demonstration assumes only character type for the operators and operands. Extend it to operands of type strings.
