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

#define MAX 10

struct stack
{
        int arr[MAX] ;
        int top ;
} ;

void initstack ( struct stack * ) ;
void push ( struct stack *, int item ) ;
```

```
int pop ( struct stack * ) ;

int main( )
{
    struct stack s ;
    int i ;

    system ( "cls" ) ;

    initstack ( &s ) ;

    push ( &s, 11 ) ;
    push ( &s, 23 ) ;
    push ( &s, -8 ) ;
    push ( &s, 16 ) ;
    push ( &s, 27 ) ;
    push ( &s, 14 ) ;
    push ( &s, 20 ) ;
    push ( &s, 39 ) ;
    push ( &s, 2 ) ;
    push ( &s, 15 ) ;
    push ( &s, 7 ) ;

    i = pop ( &s ) ;
    if ( i != NULL )
        printf ( "Item popped: %d\n", i ) ;

    i = pop ( &s ) ;
    if ( i != NULL )
        printf ( "Item popped: %d\n", i ) ;

    i = pop ( &s ) ;
    if ( i != NULL )
        printf ( "Item popped: %d\n", i ) ;

    i = pop ( &s ) ;
    if ( i != NULL )
        printf ( "Item popped: %d\n", i ) ;

    i = pop ( &s ) ;
    if ( i != NULL )
        printf ( "Item popped: %d\n", i ) ;
```

```
    i = pop ( &s ) ;
    if ( i != NULL )
        printf ( "Item popped: %d\n", i ) ;

    return 0 ;
}

/* intializes the stack */
void initstack ( struct stack *s )
{
    s -> top = -1 ;
}

/* adds an element to the stack */
void push ( struct stack *s, int item )
{
    if ( s -> top == MAX - 1 )
    {
        printf ( "Stack is full.\n" ) ;
        return ;
    }
    s -> top++ ;
    s -> arr[s ->top] = item ;
}

/* removes an element from the stack */
int pop ( struct stack *s )
{
    int data ;
    if ( s -> top == -1 )
    {
        printf ( "Stack is empty.\n" ) ;
        return NULL ;
    }
    data = s -> arr[s -> top] ;
    s -> top-- ;
    return data ;
```

```
}
```

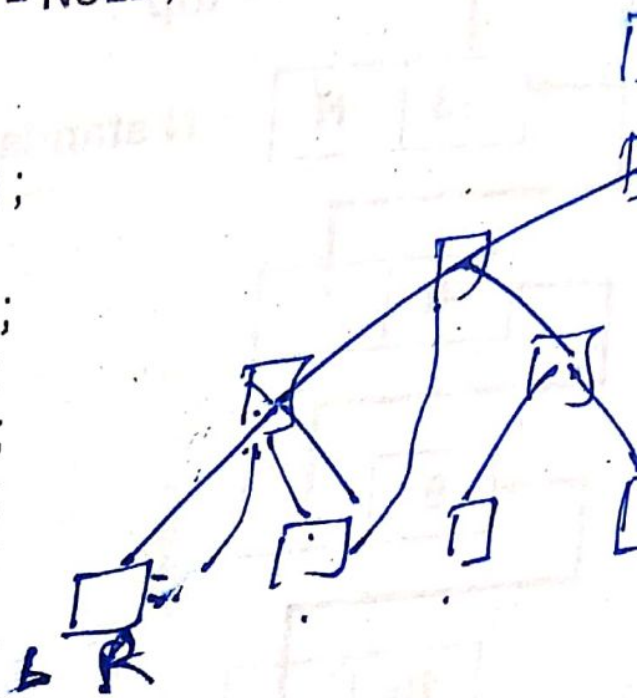
Output:

```
Stack is full.
Item popped: 15
Item popped: 2
Item popped: 39
Item popped: 20
Item popped: 14
```

Let us now see a program that implements stack as a linked list.

```
#include <stdio.h>
#include <conio.h>
#include <malloc.h>
#include <windows.h>

/* structure containing data part and link part */
struct node
{
    int data ;
    struct node *link ;
} ;

void push ( struct node **, int ) ;
int pop ( struct node ** ) ;
void delstack ( struct node ** ) ;

int main( )
```

```
{
    struct node *s = NULL ;
    int i ;

    system ( "cls" ) ;

    push ( &s, 14 ) ;
    push ( &s, -3 ) ;
    push ( &s, 18 ) ;
    push ( &s, 29 ) ;
    push ( &s, 31 ) ;
    push ( &s, 16 ) ;

    i = pop ( &s ) ;
    if ( i != NULL )
        printf ( "Item popped: %d\n", i ) ;

    i = pop ( &s ) ;
    if ( i != NULL )
        printf ( "Item popped: %d\n", i ) ;

    i = pop ( &s ) ;
    if ( i != NULL )
        printf ( "Item popped: %d\n", i ) ;

    delstack ( &s ) ;

    return 0 ;
}

/* adds a new node to the stack as linked list */
void push ( struct node **top, int item )
{
    struct node *temp ;
    temp = ( struct node * ) malloc ( sizeof ( struct node ) ) ;

    if ( temp == NULL )
        printf ( "Stack is full.\n" ) ;
```



```
    temp -> data = item ;
    temp -> link = *top ;
    *top = temp ;
}

/* pops an element from the stack */
int pop ( struct node **top )
{
    struct node *temp ;
    int item ;

    if ( *top == NULL )
    {
        printf ( "Stack is empty.\n" ) ;
        return NULL ;
    }

    temp = *top ;
    item = temp -> data ;
    *top = ( *top ) -> link ;

    free ( temp ) ;
    return item ;
}

/* deallocates memory */
void delstack ( struct node **top )
{
    struct node *temp ;

    if ( *top == NULL )
        return ;

    while ( *top != NULL )
    {
        temp = *top ;
        *top = ( *top ) -> link ;
```

```
            free ( temp ) ;
        }
}
```

Output:

```
Item popped: 16
Item popped: 31
Item popped: 29
```

```
(A+B)*C
A*(B-C)
(A+B)/(C-D)
```

The expressions within a pair of parentheses are always eva earlier than other operations.

In prefix notation the operator comes before the operand example, consider an arithmetic expression expressed in notation as shown below:

```
A+B
```

This expression in prefix form would be represented as follow

```
+AB
```

The same expression in postfix form would be represent follows:

```
AB+
```

In postfix notation, the operator follows the two operands.

The prefix and postfix expressions have three features:

- The operands maintain the same order as in the equi infix expression

Infix To Prefix Conversion

Let us now see a program that would accept an expression in inf form and convert it to a prefix form.

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

#define MAX 50

struct infix
{
    char target[MAX] ;
    char stack[MAX] ;
    char *s, *t ;
    int top, l ;
};

void initinfix ( struct infix * ) ;
void setexpr ( struct infix *, char * ) ;
void push ( struct infix *, char ) ;
char pop ( struct infix * ) ;
void convert ( struct infix * ) ;
int priority ( char c ) ;
void show ( struct infix ) ;

int main( )
{
    struct infix q ;
```

```
    char expr[MAX] ;

    system ( "cls" ) ;

    initinfix ( &q ) ;

    printf ( "Enter an expression in infix form:\n" ) ;
    gets ( expr ) ;

    setexpr ( &q, expr ) ;
    convert ( &q ) ;

    printf ( "The Prefix expression is:\n" ) ;
    show ( q ) ;

    return 0 ;
}

/* initializes elements of structure variable */
void initinfix ( struct infix *pq )
{
    pq -> top = -1 ;
    strcpy ( pq -> target, "" ) ;
    strcpy ( pq -> stack, "" ) ;
    pq -> l = 0 ;
}

/* reverses the given expression */
void setexpr ( struct infix *pq, char *str )
{
    pq -> s = str ;
    strrev ( pq -> s ) ;
    pq -> l = strlen ( pq -> s ) ;
    *( pq -> target + pq -> l ) = '\0' ;
    pq -> t = pq -> target + ( pq -> l - 1 ) ;
}

/* adds operator to the stack */
```

```
void push ( struct infix *pq, char c )
{
    if ( pq -> top == MAX - 1 )
        printf ( "Stack is full.\n" ) ;
    else
    {
        pq -> top++ ;
        pq -> stack[pq -> top] = c ;
    }
}

/* pops an operator from the stack */
char pop ( struct infix *pq )
{
    if ( pq -> top == -1 )
    {
        printf ( "Stack is empty.\n" ) ;
        return -1 ;
    }
    else
    {
        char item = pq -> stack[pq -> top] ;
        pq -> top-- ;
        return item ;
    }
}

/* converts the infix expr. to prefix form */
void convert ( struct infix *pq )
{
    char opr ;

    while ( *( pq -> s ) )
    {
        if ( *( pq -> s ) == ' ' || *( pq -> s ) == '\t' )
        {
            pq -> s++  ;
            continue ;
```

```
}

if ( isdigit ( *( pq -> s ) ) || isalpha ( *( pq -> s ) ) )
{
    while ( isdigit ( *( pq -> s ) ) || isalpha ( *( pq -> s ) ) )
    {
        *( pq -> t ) = *( pq -> s ) ;
        pq -> s++  ;
        pq -> t-- ;
    }
}

if ( *( pq -> s ) == ')' )
{
    push ( pq, *( pq -> s ) ) ;
    pq -> s++  ;
}

if ( *( pq -> s ) == '*' || *( pq -> s ) == '+' ||
  *( pq -> s ) == '/' || *( pq -> s ) == '%' ||
  *( pq -> s ) == '-' || *( pq -> s ) == '$' )
{
    if ( pq -> top != -1 )
    {
        opr = pop ( pq ) ;

        while ( priority ( opr ) > priority ( *( pq -> s ) ) )
        {
            *( pq -> t ) = opr ;
            pq -> t-- ;
            opr = pop ( pq ) ;
        }
        push ( pq, opr ) ;
        push ( pq, *( pq -> s ) ) ;
    }
    else
        push ( pq, *( pq -> s ) ) ;
    pq -> s++  ;
```

```
        }

        if ( *( pq -> s ) == '(' )
        {
            opr = pop ( pq ) ;
            while ( opr != ')' )
            {
                *( pq -> t ) = opr ;
                pq -> t-- ;
                opr = pop ( pq ) ;
            }
            pq -> s++   ;
        }
    }

    while ( pq -> top != -1 )
    {
        opr = pop ( pq ) ;
        *( pq -> t ) = opr ;
        pq -> t-- ;
    }
    pq -> t++ ;
}

/* returns the priority of the operator */
int priority ( char c )
{
    if ( c == '$' )
        return 3 ;
    if ( c == '*' || c == '/' || c == '%' )
        return 2 ;
    else
    {
        if ( c == '+' || c == '-' )
            return 1 ;
        else
            return 0 ;
    }
```

```
}

/* displays the prefix form of given expr. */
void show ( struct infix pq )
{
        while ( *( pq.t ) )
        {
                printf ( " %c", *( pq.t ) ) ;
                pq.t++ ;
        }
}
```

Output:

```
Enter an expression in infix form:
4 $ 2 * 3 - 3 + 8 / 4 / ( 1 + 1 )
Stack is empty.
The Prefix expression is:
  + - * $ 4 2 3 3 / / 8 4 + 1 1
```

Infix To Postfix Conversion

Let us now see a program that converts an arithmetic expres
given in an infix form to a postfix form.

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

```
#include <string.h>
#include <ctype.h>
#include <windows.h>

#define MAX 50

struct infix
{
    char target[MAX] ;
    char stack[MAX] ;
    char *s, *t ;
    int top ;
} ;

void initinfix ( struct infix * ) ;
void setexpr ( struct infix *, char * ) ;
void push ( struct infix *, char ) ;
char pop ( struct infix * ) ;
void convert ( struct infix * ) ;
int priority ( char ) ;
void show ( struct infix ) ;

int main( )
{
    struct infix p ;
    char expr[MAX] ;

    initinfix ( &p ) ;

    system ( "cls" ) ;

    printf ( "Enter an expression in infix form:\n" ) ;
    gets ( expr ) ;

    setexpr ( &p, expr ) ;
    convert ( &p ) ;

    printf ( "The postfix expression is:\n" ) ;
```

```
    show ( p ) ;

    return 0 ;
}

/* initializes structure elements */
void initinfix ( struct infix *p )
{
    p -> top = -1 ;
    strcpy ( p -> target, "" ) ;
    strcpy ( p -> stack, "" ) ;
    p -> t = p -> target ;
    p -> s = "" ;
}

/* sets s to point to given expr. */
void setexpr ( struct infix *p, char *str )
{
    p -> s = str ;
}

/* adds an operator to the stack */
void push ( struct infix *p, char c )
{
    if ( p -> top == MAX )
        printf ( "Stack is full.\n" ) ;
    else
    {
        p -> top++ ;
        p -> stack[p -> top] = c ;
    }
}

/* pops an operator from the stack */
char pop ( struct infix *p )
{
    if ( p -> top == -1 )
    {
```

```
        printf ( "Stack is empty.\n" ) ;
        return -1 ;
    }
    else
    {
        char item = p -> stack[p -> top] ;
        p -> top-- ;
        return item ;
    }
}

/* converts the given expr. from infix to postfix form */
void convert ( struct infix *p )
{
    char opr ;

    while ( *( p -> s ) )
    {
        if ( *( p -> s ) == ' ' || *( p -> s ) == '\t' )
        {
            p -> s++ ;
            continue ;
        }
        if ( isdigit ( *( p -> s ) ) || isalpha ( *( p -> s ) ) )
        {
            while ( isdigit ( *( p -> s ) ) || isalpha ( *( p -> s ) ) )
            {
                *( p -> t ) = *( p -> s ) ;
                p -> s++ ;
                p -> t++ ;
            }
        }
        if ( *( p -> s ) == '(' )
        {
            push ( p, *( p -> s ) ) ;
            p -> s++ ;
        }
```

```
if ( *( p -> s ) == '*' || *( p -> s ) == '+' || *( p -> s ) == '/' ||
  *( p -> s ) == '%' || *( p -> s ) == '-' || *( p -> s ) == '$' )
{
    if ( p -> top != -1 )
    {
        opr = pop ( p ) ;
        while ( priority ( opr ) >= priority ( *( p -> s ) ) )
        {
            *( p -> t ) = opr ;
            p -> t++ ;
            opr = pop ( p ) ;
        }
        push ( p, opr ) ;
        push ( p, *( p -> s ) ) ;
    }
    else
        push ( p, *( p -> s ) ) ;
    p -> s++ ;
}

if ( *( p -> s ) == ')' )
{
    opr = pop ( p ) ;
    while ( ( opr ) != '(' )
    {
        *( p -> t ) = opr ;
        p -> t++ ;
        opr = pop ( p ) ;
    }
    p -> s++ ;
}
}

while ( p -> top != -1 )
{
    char opr = pop ( p ) ;
    *( p -> t ) = opr ;
    p -> t++ ;
```

```
    }

    *( p -> t ) = '\0' ;
}

/* returns the priority of an operator */
int priority ( char c )
{
    if ( c == '$' )
        return 3 ;
    if ( c == '*' || c == '/' || c == '%' )
        return 2 ;
    else
    {
        if ( c == '+' || c == '-' )
            return 1 ;
        else
            return 0 ;
    }
}

/* displays the postfix form of given expr. */
void show ( struct infix p )
{
   printf ( "%s", p.target ) ;
}
```

Output:

```
Enter an expression in infix form:
4 $ 2 * 3 - 3 + 8 / 4 / ( 1 + 1 )
Stack is empty.
The postfix expression is:
```

Postfix To Prefix Conversion

Let us now see a program that converts an exp
form to a prefix form.

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

#define MAX 50

struct postfix
{
    char stack[MAX][MAX], target[MAX] ;
    char temp1[2], temp2[2] ;
    char str1[MAX], str2[MAX], str3[MAX] ;
    int i, top ;
} ;

void initpostfix ( struct postfix * ) ;
void setexpr ( struct postfix *, char * ) ;
void push ( struct postfix *, char * ) ;
void pop ( struct postfix *, char * ) ;
void convert ( struct postfix * ) ;
void show ( struct postfix ) ;

int main( )
{
    struct postfix q ;
    char expr[MAX] ;

    system ( "cls" ) ;

    initpostfix ( &q ) ;

    printf ( "Enter an expression in postfix form:\n" ) ;
    gets ( expr ) ;

    setexpr ( &q, expr ) ;
    convert ( &q ) ;

    printf ( "The Prefix expression is:\n" ) ;
```

```
    show ( q ) ;
    return 0 ;
}

/* initializes the elements of the structure */
void initpostfix ( struct postfix *p )
{
    p -> i = 0 ;
    p -> top = -1 ;
    strcpy ( p -> target, "" ) ;
}

/* copies given expr. to target string */
void setexpr ( struct postfix *p, char *c )
{
    strcpy ( p -> target, c ) ;
}

/* adds an operator to the stack */
void push ( struct postfix *p, char *str )
{
    if ( p -> top == MAX - 1 )
        printf ( "Stack is full.\n" ) ;
    else
    {
        p -> top++ ;
        strcpy ( p -> stack[p -> top], str ) ;
    }
}

/* pops an element from the stack */
void pop ( struct postfix *p, char *a )
{
    if ( p -> top == -1 )
    {
        printf ( "Stack  is empty.\n" ) ;
        return NULL ;
    }
```

```
    else
    {
        strcpy ( a, p -> stack[p -> top] ) ;
        p -> top-- ;
    }
}

/* converts given expr. to prefix form */
void convert ( struct postfix *p )
{
    while ( p -> target[p -> i] != '\0' )
    {
        /* skip whitespace, if any */
        if ( p -> target[p -> i] == ' ')
            p -> i++ ;
        if( p -> target[p -> i] == '%' || p -> target[p -> i] == '*' ||
            p -> target[p -> i] == '-' || p -> target[p -> i] == '+' ||
            p -> target[p -> i] == '/' || p -> target[p -> i] == '$' )
        {
            pop ( p, p -> str2 ) ;
            pop ( p, p -> str3 ) ;
            p -> temp1[0] = p -> target[ p -> i] ;
            p -> temp1[1] = '\0' ;
            strcpy ( p -> str1, p -> temp1 ) ;
            strcat ( p -> str1, p -> str3 ) ;
            strcat ( p -> str1, p -> str2 ) ;
            push ( p, p -> str1 ) ;
        }
        else
        {
            p -> temp1[0] = p -> target[p -> i] ;
            p -> temp1[1] = '\0' ;
            strcpy ( p -> temp2, p -> temp1 ) ;
            push ( p, p -> temp2 ) ;
        }
        p -> i++ ;
    }
}
```

```
/* displays the prefix form of expr. */
void show ( struct postfix p )
{
    char *temp = p.stack[0] ;
    while ( *temp )
    {
        printf ( "%c ", *temp ) ;
        temp++ ;
    }
}
```

Postfix To Infix Conversion

Let us now see a program to convert an expression in postfix form to an infix form.

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

#define MAX 50

struct postfix
{
    char stack[MAX][MAX], target[MAX] ;
    char temp1[2], temp2[2] ;
    char str1[MAX], str2[MAX], str3[MAX] ;
    int i, top ;
};

void initpostfix ( struct postfix * ) ;
void setexpr ( struct postfix *, char * ) ;
void push ( struct postfix *, char * ) ;
void pop ( struct postfix *, char * ) ;
void convert ( struct postfix * ) ;
void show ( struct postfix ) ;

int main( )
{
    struct postfix q ;
    char expr[MAX] ;

    system ( "cls" ) ;

    initpostfix ( &q ) ;

    printf ( "Enter an expression in postfix form:\n" ) ;
```

```
    gets ( expr ) ;

    setexpr ( &q, expr ) ;
    convert ( &q ) ;

    printf ( "The infix expression is:\n" ) ;
    show ( q ) ;
    return 0 ;
}

/* initializes structure elements */
void initpostfix ( struct postfix *p )
{
    p -> i = 0 ;
    p -> top = -1 ;
    strcpy ( p -> target, "" ) ;
}

/* copies given expression to target string */
void setexpr ( struct postfix *p, char *c )
{
    strcpy ( p -> target, c ) ;
}

/* adds an expr. to the stack */
void push ( struct postfix *p, char *str )
{
    if ( p -> top == MAX - 1 )
        printf ( "Stack is full.\n" ) ;
    else
    {
        p -> top++ ;
        strcpy ( p -> stack[p -> top], str ) ;
    }
}

/* pops an expr. from the stack */
void pop ( struct postfix *p, char *a )
```

```
{
    if ( p -> top == -1 )
    {
        printf ( "Stack  is empty.\n" ) ;
        return NULL ;
    }
    else
    {
        strcpy ( a, p -> stack[p -> top] ) ;
        p -> top-- ;
    }
}

/* converts given expr. to infix form */
void convert ( struct postfix *p )
{
    while ( p -> target[p -> i] )
    {
        /* skip whitespace, if any */
        if( p -> target[p -> i] == ' ' )
            p -> i++ ;
        if ( p -> target[p -> i] == '%' || p -> target[p -> i] == '*' ||
             p -> target[p -> i] == '-' || p -> target[p -> i] == '+' ||
             p -> target[p -> i] == '/' || p -> target[p -> i] == '$' )
        {
            pop ( p, p -> str2 ) ;
            pop ( p, p -> str3 ) ;
            p -> temp1[0] = p -> target[p -> i] ;
            p -> temp1[1] = '\0' ;
            strcpy ( p -> str1, p -> str3 ) ;
            strcat ( p -> str1, p -> temp1 ) ;
            strcat ( p -> str1, p -> str2 ) ;
            push ( p, p -> str1 ) ;
        }
        else
        {
            p -> temp1[0] = p -> target[p -> i] ;
            p -> temp1[1] = '\0' ;
```

```
                strcpy ( p -> temp2, p -> temp1 ) ;
                push ( p, p -> temp2 ) ;
            }
            p -> i++ ;
        }
}

/* displays the expression */
void show ( struct postfix p )
{
        char *t ;
        t = p.stack[0] ;
        while ( *t )
        {
            printf ( "%c ", *t ) ;
            t++ ;
        }
}
```

Output:

```
Enter an expression in postfix form:
4 2 $ 3 * 3 - 8 4 / 1 1 + / +
The infix expression is:
4 $ 2 * 3 - 3 + 8 / 4 / 1 + 1
```

Postfix Expression: 4 2 $ 3 * 3 - 8 4 / 1 1 + / +

Char. Scanned	Stack Contents
4	4
2	4, 2
$	4 $ 2
3	4 $ 2, 3
*	4 $ 2 * 3
3	4 $ 2 * 3, 3
-	4 $ 2 * 3 – 3
8	4 $ 2 * 3 – 3, 8
4	4 $ 2 * 3 – 3, 8, 4
/	4 $ 2 * 3 – 3, 8 / 4
1	4 $ 2 * 3 – 3, 8 / 4, 1
1	4 $ 2 * 3 – 3, 8 / 4, 1, 1
+	4 $ 2 * 3 – 3, 8 / 4 , 1 + 1
/	4 $ 2 * 3 – 3, 8 / 4 / 1 + 1
+	4 $ 2 * 3 – 3 + 8 / 4 / 1 + 1

Evaluation Of Postfix Expression

The virtue of postfix notation is that it enables easy evaluation of expressions. To begin with, the need for parentheses is eliminated. Secondly, the priority of the operators is no longer relevant. The expression can be evaluated by making a left to right scan, stacking operands, and evaluating operators using as operands the correct elements from the stack and finally placing the result onto the

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <math.h>
#include <ctype.h>
#include <windows.h>

#define MAX 50

struct postfix
{
    int stack[MAX] ;
    int top, nn ;
    char *s ;
};

void initpostfix ( struct postfix * ) ;
void setexpr ( struct postfix *, char * ) ;
void push ( struct postfix *, int ) ;
int pop ( struct postfix * ) ;
void calculate ( struct postfix * ) ;
void show ( struct postfix ) ;

int main( )
{
    struct postfix q ;
    char expr[MAX] ;

    system ( "cls" ) ;

    initpostfix ( &q ) ;

    printf ( "Enter postfix expression to be evaluated:\n" ) ;
```

```
    gets ( expr ) ;

    setexpr ( &q, expr ) ;
    calculate ( &q ) ;
    show ( q ) ;

    return 0 ;
}

/* initializes structure elements */
void initpostfix ( struct postfix *p )
{
    p -> top = -1 ;
}

/* sets s to point to the given expr. */
void setexpr ( struct postfix *p, char *str )
{
    p -> s = str ;
}

/* adds digit to the stack */
void push ( struct postfix *p, int item )
{
    if ( p -> top == MAX - 1 )
        printf ( "Stack is full.\n" ) ;
    else
    {
        p -> top++ ;
        p -> stack[p -> top] = item ;
    }
}

/* pops digit from the stack */
int pop ( struct postfix *p )
{
    int data ;
    if ( p -> top == -1 )
```

```
    {
        printf ( "Stack is empty.\n" ) ;
        return NULL ;
    }

    data = p -> stack[p -> top] ;
    p -> top-- ;

    return data ;
}

/* evaluates the postfix expression */
void calculate( struct postfix *p )
{
    int n1, n2, n3 ;
    while ( *( p -> s ) )
    {
        /* skip whitespace, if any */
        if ( *( p -> s ) == ' ' || *( p -> s ) == '\t' )
        {
            p -> s++ ;
            continue ;
        }

        /* if digit is encountered */
        if ( isdigit ( *( p -> s ) ) )
        {
            p -> nn = *( p -> s ) - '0' ;
            push ( p, p -> nn ) ;
        }
        else
        {
            /* if operator is encountered */
            n1 = pop ( p ) ;
            n2 = pop ( p ) ;
            switch ( *( p -> s ) )
            {
                case '+' :
```

```
                    n3 = n2 + n1 ;
                    break ;

                case '-' :
                    n3 = n2 - n1 ;
                    break ;

                case '/' :
                    n3 = n2 / n1 ;
                    break ;

                case '*' :
                    n3 = n2 * n1 ;
                    break ;

                case '%' :
                    n3 = n2 % n1 ;
                    break ;

                case '$' :
                    n3 = ( int ) pow ( n2 , n1 ) ;
                    break ;

                default :
                    printf ( "Unknown operator\n" ) ;
                    exit ( 1 ) ;
            }

            push ( p, n3 ) ;
        }
        p -> s++ ;
    }
}

/* displays the result */
void show ( struct postfix p )
{
    p.nn = pop ( &p ) ;
```

```
    printf ( "Result is: %d", p.nn ) ;
}
```

Output:

```
Enter postfix expression to be evaluated:
4 2 $ 3 * 3 - 8 4 / 1 1 + / +
Result is: 46
```