# **Assignment 7** Dated Jan 27th, 2025

## **Problem Statement**

A program in C to convert an infix expression into a postfix, where the input is given by the user.

## **Algorithm**

### Input

createStack(), push(), infixToPostfix() takes necessary input.

### Output

peek() and infixToPostfix() displays the necessary output.

**Algorithm for createStack(cap)**

**Step 1:** Start.  
**Step 2:** Input an integer cap (capacity of the stack).  
**Step 3:** Allocate memory for a Stack structure and assign it to stack.  
**Step 4:** Set stack->cap = cap.  
**Step 5:** Set stack->top = -1 (indicating an empty stack).  
**Step 6:** Allocate memory for stack->array of size cap.  
**Step 7:** Return the pointer to the newly created stack.  
**Step 8:** Stop.  
**Step 9:** [End of function createStack defined at Step 1.]

**Algorithm for isFull(stack)**

**Step 10:** Start.  
**Step 11:** Input a pointer to stack.  
**Step 12:** If stack->top == stack->cap - 1, return 1 (stack is full).  
**Step 13:** Otherwise, return 0 (stack is not full).  
**Step 14:** Stop.  
**Step 15:** [End of function isFull defined at Step 10.]

**Algorithm for isEmpty(stack)**

**Step 16:** Start.  
**Step 17:** Input a pointer to stack.  
**Step 18:** If stack->top == -1, return 1 (stack is empty).  
**Step 19:** Otherwise, return 0 (stack is not empty).  
**Step 20:** Stop.  
**Step 21:** [End of function isEmpty defined at Step 16.]

**Algorithm for push(stack, item)**

**Step 22:** Start.  
**Step 23:** Input a pointer to stack and a character item.  
**Step 24:** If isFull(stack) == 1, return (stack is full, no push operation).  
**Step 25:** Increment stack->top by 1.  
**Step 26:** Assign item to stack->array[stack->top].  
**Step 27:** Stop.  
**Step 28:** [End of function push defined at Step 22.]

**Algorithm for pop(stack)**

**Step 29:** Start.  
**Step 30:** Input a pointer to stack.  
**Step 31:** If isEmpty(stack) == 1, return '$', indicating an underflow.  
**Step 32:** Retrieve stack->array[stack->top] and store it in popped.  
**Step 33:** Decrement stack->top by 1.  
**Step 34:** Return popped.  
**Step 35:** Stop.  
**Step 36:** [End of function pop defined at Step 29.]

**Algorithm for peek(stack)**

**Step 37:** Start.  
**Step 38:** Input a pointer to stack.  
**Step 39:** If isEmpty(stack) == 1, return '$', indicating an empty stack.  
**Step 40:** Return stack->array[stack->top] without modifying top.  
**Step 41:** Stop.  
**Step 42:** [End of function peek defined at Step 37.]

**Algorithm for isOperand(ch)**

**Step 43:** Start.  
**Step 44:** Input a character ch.  
**Step 45:** If ch is an alphanumeric character, return 1.  
**Step 46:** Otherwise, return 0.  
**Step 47:** Stop.  
**Step 48:** [End of function isOperand defined at Step 43.]

**Algorithm for precedence(ch)**

**Step 49:** Start.  
**Step 50:** Input a character ch.  
**Step 51:** If ch is '+' or '-', return 1.  
**Step 52:** If ch is '\*' or '/', return 2.  
**Step 53:** If ch is '^', return 3.  
**Step 54:** Otherwise, return -1.  
**Step 55:** Stop.  
**Step 56:** [End of function precedence defined at Step 49.]

**Algorithm for infixToPostfix(exp)**

**Step 57:** Start.  
**Step 58:** Input a character array exp.  
**Step 59:** Declare integers i and k.  
**Step 60:** Create a stack of capacity equal to strlen(exp) and store it in stack.  
**Step 61:** If memory allocation for stack fails, return -1.  
**Step 62:** Initialize k = -1 (to track output position).  
**Step 63:** Iterate over each character of exp using a loop:

* **Step 63.1:** If exp[i] is an operand, append it to exp[++k].
* **Step 63.2:** If exp[i] is '(', push it onto stack.
* **Step 63.3:** If exp[i] is ')':
  + While stack is not empty and peek(stack) != '(', append pop(stack) to exp[++k].
  + If peek(stack) == '(', pop it.
* **Step 63.4:** If exp[i] is an operator:
  + While stack is not empty and precedence(exp[i]) <= precedence(peek(stack)), append pop(stack) to exp[++k].
  + Push exp[i] onto stack.  
    **Step 64:** After the loop, pop all remaining elements from stack and append to exp[++k].  
    **Step 65:** Append '\0' at the end of exp to terminate the string.  
    **Step 66:** Print the postfix expression.  
    **Step 67:** Return 0.  
    **Step 68:** Stop.  
    **Step 69:** [End of function infixToPostfix defined at Step 57.]

**Algorithm for main()**

**Step 70:** Start.  
**Step 71:** Declare a character array exp[100].  
**Step 72:** Display a prompt: "Enter an infix expression:".  
**Step 73:** Input a string into exp.  
**Step 74:** Call infixToPostfix(exp).  
**Step 75:** Return 0 to indicate successful execution.  
**Step 76:** Stop.  
**Step 77:** [End of function main defined at Step 70.]

## **Source Code**

#include <ctype.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

// Stack structure

typedef struct {

    int top;

    size\_t cap;

    int\* array;

} Stack;

// Function to create a stack of given cap

Stack\* createStack(unsigned cap)

{

    Stack\* stack = (Stack\*)malloc(sizeof(Stack));

    stack->cap = cap;

    stack->top = -1;

    stack->array = (int\*)malloc(stack->cap \* sizeof(int));

    return stack;

}

// Stack is full when top is equal to the last index

int isFull(Stack\* stack)

{

    return stack->top == stack->cap - 1;

}

// Stack is empty when top is -1

int isEmpty(Stack\* stack)

{

    return stack->top == -1;

}

// Function to add an item to stack, increases top by 1

void push(Stack\* stack, char item)

{

    if (isFull(stack))

        return;

    stack->array[++stack->top] = item;

}

// Function to remove an item from stack, decreases top by 1

char pop(Stack\* stack)

{

    if (isEmpty(stack))

        return '$';

    return stack->array[stack->top--];

}

// Function to get the top item without removing it

char peek(Stack\* stack)

{

    if (isEmpty(stack))

        return '$';

    return stack->array[stack->top];

}

// A utility function to check if the given character is operand

int isOperand(char ch)

{

    return isalnum(ch);

}

// A utility function to return precedence of a given operator

int precedence(char ch)

{

    switch (ch) {

    case '+':

    case '-':

        return 1;

    case '\*':

    case '/':

        return 2;

    case '^':

        return 3;

    }

    return -1;

}

// The function to convert infix expression to postfix expression

int infixToPostfix(char\* exp)

{

    int i, k;

    // Create a stack of cap equal to expression length

    Stack\* stack = createStack(strlen(exp));

    if (!stack)

        return -1;

    for (i = 0, k = -1; exp[i]; ++i) {

        // If the character is an operand, add it to output

        if (isOperand(exp[i]))

            exp[++k] = exp[i];

        // If the character is '(', push it to stack

        else if (exp[i] == '(')

            push(stack, exp[i]);

        // If the character is ')', pop and output from the stack until '(' is encountered

        else if (exp[i] == ')') {

            while (!isEmpty(stack) && peek(stack) != '(')

                exp[++k] = pop(stack);

            if (!isEmpty(stack) && peek(stack) != '(')

                return -1; // Invalid expression

            else

                pop(stack);

        } else { // an operator is encountered

            while (!isEmpty(stack) && precedence(exp[i]) <= precedence(peek(stack)))

                exp[++k] = pop(stack);

            push(stack, exp[i]);

        }

    }

    // pop all the operators from the stack

    while (!isEmpty(stack))

        exp[++k] = pop(stack);

    exp[++k] = '\0';

    printf("Postfix expression: %s\n", exp);

    return 0;

}

// Driver program to test above functions

int main()

{

    char exp[100];

    printf("Enter an infix expression: ");

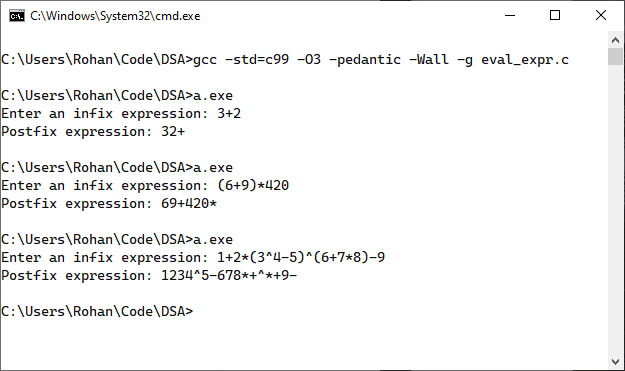
    scanf("%s", exp);

    infixToPostfix(exp);

    return 0;

}

## **Output**



### Discussion

Stack should be properly initialized before use. The precedence order must be considered carefully.

**Teacher’s signature**