**Lab 3**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

#include<math.h>

#define MAX\_EXPR\_SIZE 100

// Structure to represent a stack

struct Stack {

int top;

unsigned capacity;

int\* array;

};

// Function to create a stack of given capacity

struct Stack\* createStack(unsigned capacity) {

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

stack->capacity = capacity;

stack->top = -1;

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

return stack;

}

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

int isFull(struct Stack\* stack) {

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

}

// Stack is empty when top is equal to -1

int isEmpty(struct Stack\* stack) {

return stack->top == -1;

}

// Function to add an item to stack

void push(struct Stack\* stack, int item) {

if (isFull(stack))

return;

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

}

// Function to remove an item from stack

int pop(struct Stack\* stack) {

if (isEmpty(stack))

return '\0';

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

}

// Function to get the top item from stack

int peek(struct Stack\* stack) {

if (isEmpty(stack))

return '\0';

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

}

// Function to check if the given character is an operator

int isOperator(char ch) {

return (ch == '+' || ch == '-' || ch == '\*' || ch == '/' || ch == '^');

}

// Function to return the precedence of operators

int precedence(char ch) {

if (ch == '+' || ch == '-')

return 1;

else if (ch == '\*' || ch == '/')

return 2;

else if (ch == '^')

return 3;

return -1;

}

// Function to evaluate postfix expression

int evaluatePostfix(char\* postfix) {

struct Stack\* stack = createStack(strlen(postfix));

int i;

for (i = 0; postfix[i]; ++i) {

if (isdigit(postfix[i])) {

push(stack, postfix[i] - '0');

} else {

int val1 = pop(stack);

int val2 = pop(stack);

switch (postfix[i]) {

case '+': push(stack, val2 + val1); break;

case '-': push(stack, val2 - val1); break;

case '\*': push(stack, val2 \* val1); break;

case '/': push(stack, val2 / val1); break;

case '^': push(stack, (int)pow(val2, val1)); break;

}

}

}

return pop(stack);

}

// Function to convert infix expression to postfix

char\* infixToPostfix(char\* infix) {

int i, j;

int len = strlen(infix);

char\* postfix = (char\*)malloc(sizeof(char) \* (len + 2));

char stack[MAX\_EXPR\_SIZE];

int top = -1;

for (i = 0, j = 0; i < len; i++) {

if (infix[i] == ' ' || infix[i] == '\t')

continue;

if (isalnum(infix[i])) {

postfix[j++] = infix[i];

} else if (infix[i] == '(') {

stack[++top] = infix[i];

} else if (infix[i] == ')') {

while (top > -1 && stack[top] != '(')

postfix[j++] = stack[top--];

top--;

} else if (isOperator(infix[i])) {

while (top > -1 && precedence(stack[top]) >= precedence(infix[i]))

postfix[j++] = stack[top--];

stack[++top] = infix[i];

}

}

while (top > -1) {

if (stack[top] == '(') {

free(postfix);

return "Invalid Expression";

}

postfix[j++] = stack[top--];

}

postfix[j] = '\0';

return postfix;

}

// Driver code

int main() {

char infix[MAX\_EXPR\_SIZE];

printf("Enter an infix expression: ");

scanf("%s", infix);

char\* postfix = infixToPostfix(infix);

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

int result = evaluatePostfix(postfix);

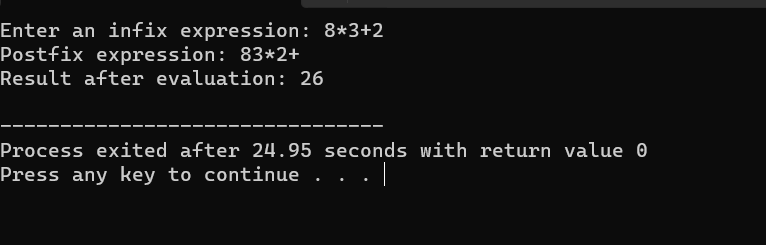
printf("Result after evaluation: %d\n", result);

free(postfix);

return 0;

}

**Output**

****