program to evaluate infix expression using stacks in c

#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

#include <string.h>

#define MAX\_SIZE 100

// Stack data structure

struct Stack {

int top;

int items[MAX\_SIZE];

};

// Push an item to the stack

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

if (stack->top == MAX\_SIZE - 1) {

printf("Stack Overflow\n");

}

else {

stack->top++;

stack->items[stack->top] = item;

}

}

// Pop an item from the stack

int pop(struct Stack\* stack) {

if (stack->top == -1) {

printf("Stack Underflow\n");

return -1;

}

else {

int item = stack->items[stack->top];

stack->top--;

return item;

}

}

// Check if a character is an operator

int isOperator(char c) {

if (c == '+' || c == '-' || c == '\*' || c == '/') {

return 1;

}

else {

return 0;

}

}

// Get the precedence of an operator

int getPrecedence(char op) {

if (op == '\*' || op == '/') {

return 2;

}

else if (op == '+' || op == '-') {

return 1;

}

else {

return 0;

}

}

// Evaluate infix expression

int evaluateInfix(char\* infix) {

int i, len, operand1, operand2, result;

char token;

struct Stack operatorStack;

struct Stack operandStack;

// Initialize stacks

operatorStack.top = -1;

operandStack.top = -1;

// Get length of infix expression

len = strlen(infix);

// Traverse infix expression

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

token = infix[i];

// If current token is an operand

if (isdigit(token)) {

// Convert char to int and push onto operand stack

push(&operandStack, token - '0');

}

// If current token is an operator

else if (isOperator(token)) {

// While the top of the operator stack has higher or equal precedence,

// pop the top operator, pop the top two operands, apply the operator,

// and push the result onto the operand stack.

while (operatorStack.top != -1 && getPrecedence(operatorStack.items[operatorStack.top]) >= getPrecedence(token)) {

operand2 = pop(&operandStack);

operand1 = pop(&operandStack);

switch (pop(&operatorStack)) {

case '+':

result = operand1 + operand2;

break;

case '-':

result = operand1 - operand2;

break;

case '\*':

result = operand1 \* operand2;

break;

case '/':

result = operand1 / operand2;

break;

}

push(&operandStack, result);

}

// Push the current operator onto the operator stack

push(&operatorStack, token);

}

// If current token is a left parenthesis

else if (token == '(') {

push(&operatorStack, token);

}

// If current token is a right parenthesis

else if (token == ')') {

// While the top of the operator stack is not a left parenthesis,

// pop the top operator, pop the top two operands, apply the operator,

// and push the result onto// the operand stack.

while (operatorStack.items[operatorStack.top] != '(') {

operand2 = pop(&operandStack);

operand1 = pop(&operandStack);

switch (pop(&operatorStack)) {

case '+':

result = operand1 + operand2;

break;

case '-':

result = operand1 - operand2;

break;

case '\*':

result = operand1 \* operand2;

break;

case '/':

result = operand1 / operand2;

break;

}

push(&operandStack, result);

}

// Pop the left parenthesis from the operator stack

pop(&operatorStack);

}

}

// Evaluate remaining operators and operands on the stacks

while (operatorStack.top != -1) {

operand2 = pop(&operandStack);

operand1 = pop(&operandStack);

switch (pop(&operatorStack)) {

case '+':

result = operand1 + operand2;

break;

case '-':

result = operand1 - operand2;

break;

case '\*':

result = operand1 \* operand2;

break;

case '/':

result = operand1 / operand2;

break;

}

push(&operandStack, result);

}

// Return the result

return pop(&operandStack);

}

int main() {

char infix[MAX\_SIZE];

printf("Enter infix expression: ");

fgets(infix, MAX\_SIZE, stdin);

printf("Result: %d\n", evaluateInfix(infix));

return 0;

}

