6/10/25

WAP to convert a given valid parenthesized infix arithmetic expression to postfix expression. The expression consists of single character operands and the binary operator + (plus), - (minus), * (multiply) and / (divide).

Functions required
push( )
pop( )
peek( )
precedance( )
associativity( )
infix to postfix (infix[ ], postfix[ ])

1. Input an expression from the user.
2. Check whether it is a valid expression or not using balanced parenthesis.
3. Declare two arrays to store the entered expression and the postfix expression.
4. Run a loop to check the elements of entered ~~expression~~.
5. If operands arrive, just print (store it in postfix array).
   - If the stack is empty or contains open (left) parenthesis on top, push the incoming operator onto the stack.
   - If incoming symbol is '(', push it onto the stack.
   - If incoming symbol is ')', pop the stack and store the operators into postfix until left parenthesis is found.
   - If incoming symbol has high precedance than top of the stack, push it on to the stack
   - If incoming symbol has lower precedence than top of the stack, pop and print (store) the top. Then test the incoming operator against the new top of the stack.
   - If incoming symbol has equal precedence with the top of the stack, use associativity rule.
   - At the end of the expression, pop and print (store) all the operators of stack
6. Associativity rule:

   Left to right: pop and store the top of the stack and then push the incoming operator.

   Right to left: push the incoming operator.
7. for priority refer */ (Left-Right), +- (Left to Right).

8. At the end of expression, print the elements of postfix array.

program

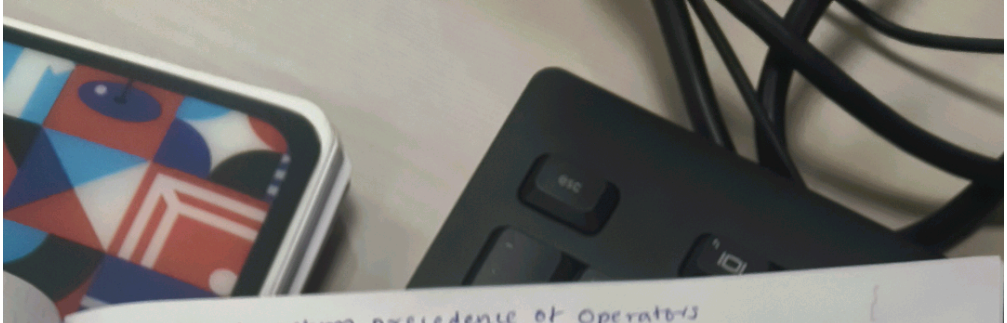
```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#define MAX 100

char stack[MAX];
int top=-1;

// Push function
void push(char c) {
    if (top == MAX-1)
    {
        printf("Stack overflow\n");
        return;
    }
    stack[++top]=c;
}

// Pop function
char pop() {
    if (top==-1)
    {
        printf("Stack underflow\n");
        return -1;
    }
    return stack[top--];
}

// peek function
char peek() {
    if (top==-1) return -1;
    return stack[top];
}
```



```
// function to return precedence of operators
int precedence (char op) {
    switch (op) {
        case '+':
        case '-':
            return 1;
        case '*':
        case '/':
            return 2;
        case '^':
            return 3;  // highest precedence
        case '(':
            return 0;
    }
    return -1;
}

// Function to return associativity
// 0 = left to right
   1 = Right to left
int associativity (char op) {
    if (op=='^')
        return 1; // right to left
    return 0;  // +, -, *, / → left to right.
}

// function to convert infix to postfix
void infixtopostfix (char infix[], char postfix[]) {
    int i, k=0;
    char c;
    for (i=0; infix[i]!='\0'; i++) {
        c= infix[i];
        if (isalnum(c)) {
            // operand → directly to postfix
            postfix[k++]=c;
        }
        else if (c=='(')
            push(c);
```

```
        }
        else if (c == ')') {
            while (peek() != '(') {
                postfix[k++] = pop();
            }
            pop(); // discard '('
        }
        else {
            // operator
            while (top != -1 &&
                   ((precedence[peek()] > precedence(c)) ||
                    (precedence(peek()) == precedence(c) &&
                     associativity(c) == 0))) { // L to R
                postfix[k++] = pop();
            }
            push(c);
        }
    }
    // Pop remaining operators
    while (top != -1) {
        postfix[k++] = pop();
    }
    postfix[k] = '\0';
}

int main() {
    char infix[MAX], postfix[MAX];
    printf("Enter a valid parenthesized infix expression: ");
    scanf("%s", infix);
    infixtopostfix(infix, postfix);
    printf("Postfix expression : %s\n", postfix);
    return 0;
}
```

output:

Enter a valid parenthesized infix expression: A * B + C * D - E

Postfix Expression : AB*CD*+E-

Enter a valid parenthesized infix expression : (A + (B * C - (D/E^F) * G) * H)

Postfix Expression : ABC* DEF ^/G* -H * + *

6/10