**Practical No. 7**

**Aim:** Write a program to evaluate a postfix expression using stacks.

**Theory**

A **postfix expression** (Reverse Polish Notation) writes each operator after its operands (e.g., 23+ for 2 + 3). Postfix expressions are convenient for computers because they remove the need for parentheses and precedence rules during evaluation. A stack is the ideal data structure for evaluation:

* Scan the expression left to right.
* Push operands onto the stack.
* When an operator is encountered, pop the top two operands, apply the operator (operand1 operator operand2), and push the result back.
* After the entire expression is processed, the remaining value on the stack is the result.

This practical uses a **simple** version that accepts single-digit operands (0–9) and no spaces, supporting + - \* / (integer division).

**Algorithm (step-by-step)**

1. Read the postfix expression as a string (no spaces; single-digit operands).
2. Initialize an empty integer stack.
3. For each character ch in the expression:
   * If ch is a digit: convert to integer and push on stack.
   * Else if ch is an operator (+ - \* /):
     + Pop operand2 from stack (top).
     + Pop operand1 from stack (next top).
     + Compute operand1 op operand2.
     + Push the result back onto stack.
   * Else: report invalid character and stop.
4. After processing all characters, if the stack has exactly one value, that is the result; otherwise report an invalid expression error.
5. Print the result and exit.

**Program**

#include <stdio.h> // Provides printf and scanf

#include <string.h> // Provides strlen for string length

#include <ctype.h> // Provides isdigit to check digits

#define MAX 100 // Maximum length of input expression and stack capacity

int main() { // Program execution starts here

char expr[MAX]; // Buffer to store postfix expression (single-digit operands)

int stack[MAX]; // Integer stack to store operands and intermediate results

int top = -1; // Stack top initialized to -1 means stack is empty

int i; // Loop index

// Prompt user for input expression

printf("Enter postfix expression (single-digit operands, e.g. 23+4\*): ");

scanf("%s", expr); // Read string input (stops at whitespace)

// Process each character of the expression

for (i = 0; i < (int)strlen(expr); i++) { // Loop through all characters

char ch = expr[i]; // Current character under examination

if (isdigit(ch)) { // If character is a digit (operand)

int val = ch - '0'; // Convert char digit to integer (ASCII trick)

stack[++top] = val; // Push the integer onto stack (increment top then assign)

}

else { // Otherwise expect an operator

if (top < 1) { // Need two operands for a binary operator

printf("Error: Insufficient operands for operator '%c'\n", ch); // Show error

return 1; // Exit due to malformed expression

}

int operand2 = stack[top--]; // Pop operand2 (right operand)

int operand1 = stack[top--]; // Pop operand1 (left operand)

int result = 0; // Variable to hold operation result

if (ch == '+') // If operator is '+'

result = operand1 + operand2; // Perform addition

else if (ch == '-') // If operator is '-'

result = operand1 - operand2; // Perform subtraction

else if (ch == '\*') // If operator is '\*'

result = operand1 \* operand2; // Perform multiplication

else if (ch == '/') { // If operator is '/'

if (operand2 == 0) { // Check for division by zero

printf("Error: Division by zero\n"); // Report error

return 1; // Exit program

}

result = operand1 / operand2; // Perform integer division

}

else { // If character is not a supported operator

printf("Error: Unsupported operator '%c'\n", ch); // Report invalid operator

return 1; // Exit

}

stack[++top] = result; // Push the computed result back onto the stack

}

}

// After processing all characters, there must be exactly one value on the stack

if (top != 0) { // If stack doesn't contain exactly one element

printf("Error: Invalid postfix expression. Stack has %d elements\n", top + 1); // Show error

return 1; // Exit with error

}

// The top of stack is the final result

printf("Result = %d\n", stack[top]); // Print the final evaluation result

return 0; // End program successfully

}

**Output**

**Example 1 - multiplication and addition**

Enter postfix expression (single-digit operands, e.g. 23+4\*): 234\*+

Result = 14

Explanation: 3 \* 4 = 12, then 2 + 12 = 14.

**Example 2 - subtraction and multiplication**

Enter postfix expression (single-digit operands, e.g. 23+4\*): 52-3\*

Result = 9

Explanation: 5 - 2 = 3, then 3 \* 3 = 9.

**Example 3 - division**

Enter postfix expression (single-digit operands, e.g. 23+4\*): 82/

Result = 4

Explanation: 8 / 2 = 4 (integer division).

**Example 4 - malformed expression (too few operands)**

Enter postfix expression (single-digit operands, e.g. 23+4\*): 5+

Error: Insufficient operands for operator '+'

Note: Your instruction mentioned "Output section must show the sequence of disk movements" - that applies to Tower of Hanoi, not postfix evaluation. For postfix, the correct output is the numeric result; above are sample runs showing results and error messages.

**Conclusion**

* A simple stack-based postfix evaluator was implemented that handles single-digit operands and operators + - \* /.
* Students learned how stacks are used to evaluate expressions: push operands, pop for operators, compute and push results.
* Basic error handling (insufficient operands, division by zero, unsupported operators) was included to make the program robust for common mistakes.
* To extend this lab: support multi-digit numbers (space-separated tokens) or add exponentiation and unary operators.