```
/**
 * Program that converts Infix expression to Postfix and performs evaluation
 * after taking input for individual variables.
 *
 * By: Santa Basnet
 * Data: 2021-11-23
 * Everest Engineering College.
*/

#include <cstdlib>
#include <cstring>
#include <cstdio>
#include <valarray>

/**
 * Boolean literals.
 */
#define TRUE 1
#define FALSE 0
#define TERMINATION '\0'

/**
 * Define operators.
 */
const char PLUS = '+';
const char MINUS = '-';
const char MULTIPLY = '*';
const char DIVIDE = '/';
const char EXPONENT = '^';
const char LEFT_PARENTHESIS = '(';
const char RIGHT_PARENTHESIS = ')';

/**
 * Definition of operator precedence.
 */
const int DEFAULT_PRECEDENCE = 0;
const int PLUS_PRECEDENCE = 1;
const int MINUS_PRECEDENCE = 1;
const int MULTIPLY_PRECEDENCE = 2;
const int DIVIDE_PRECEDENCE = 2;
const int EXPONENT_PRECEDENCE = 3;

/**
 * Returns the precedence value of the operator.
 */
int precedenceOf(char inputChar) {
    switch (inputChar) {
        case PLUS:
            return PLUS_PRECEDENCE;
        case MINUS:
            return MINUS_PRECEDENCE;
        case MULTIPLY:
            return MULTIPLY_PRECEDENCE;
        case DIVIDE:
            return DIVIDE_PRECEDENCE;
        case EXPONENT:
            return EXPONENT_PRECEDENCE;
        default :
            return DEFAULT_PRECEDENCE;
    }
}

/**
 * Expression representation.
 */
const int MAX_SIZE = 128;
struct ArithmeticExpression {
    int topOfStack;
    char operatorStack[MAX_SIZE];
    char postfixExpression[MAX_SIZE];
} expression;

/**
 * Intermediate Postfix expression.
 */
struct IntermediateExpression {
    int topOfStack;
    double stack[MAX_SIZE];
    char inputVariable[MAX_SIZE];
    double values[MAX_SIZE];
} evaluation;

/**
 * Checks if the input character is of operator type or not.
 */
int isOperator(char inputChar) {
    switch (inputChar) {
        case PLUS:
        case MINUS:
        case MULTIPLY:
        case DIVIDE:
        case EXPONENT:
            return TRUE;
        default :
            return FALSE;
    }
}

/**
 * Error: Invalid expression and exit from the program with error code 0.
 */
void invalidExpression() {
    printf("\nThe input expression is not valid.");
    exit(0);
}
```

```c
/**
 * Stack Full Error.
 */
void stackFullError() {
    printf("\nStack full, unable to compute further.");
    exit(0);
}

/**
 * Stack Empty Error.
 */
void stackEmptyError() {
    printf("\nStack empty, unable to compute further.");
    exit(0);
}

/**
 * Divide by zero Error.
 */
 void divideByZeroError(){
    printf("\nDivide by zero.");
    exit(0);
 }

/**
 * Checks if the input character is of operand or not.
 * We consider all the single lower case English alphabets are the operands.
 */
int isOperand(char inputChar) {
    if (inputChar >= 'a' && inputChar <= 'z') return TRUE;
    else return FALSE;
}

/**
 * Declare Input Expression.
 */
char *inputExpression;

/**
 * Read input expression from console.
 */
void readInputExpression() {
    size_t buffer = 128;
    printf("\nInput Expression: \n");
    getline(&inputExpression, &buffer, stdin);
}

/**
 * Scan index for input character array.
 */
int scannedIndex = 0;

/**
```

```c
 * Returns an input character.
 */
char scanInput() {
    if (scannedIndex < strlen(inputExpression)) return inputExpression[scannedIndex++];
    else return TERMINATION;
}

/**
 * Checks if the stack is full.
 */
int isStackFull() {
    return expression.topOfStack >= MAX_SIZE ? TRUE : FALSE;
}

/**
 * Checks if the stack is empty or not.
 */
int isStackEmpty() {
    return expression.topOfStack == 0 ? TRUE : FALSE;
}

/**
 * Perform push operation.
 */
void push(char inputChar) {
    if (isStackFull()) stackFullError();
    expression.operatorStack[expression.topOfStack++] = inputChar;
    expression.operatorStack[expression.topOfStack] = TERMINATION;
}

/**
 * Perform the pop operation.
 * @return character popped from the stack.
 */
char pop() {
    if (isStackEmpty()) stackEmptyError();
    char charOutput = expression.operatorStack[--expression.topOfStack];
    expression.operatorStack[expression.topOfStack] = TERMINATION;
    return charOutput;
}

/**
 * Output to postfix expression.
 */
void outputToExpression(char outputChar) {
    size_t index = strlen(expression.postfixExpression);
    expression.postfixExpression[index] = outputChar;
    expression.postfixExpression[index + 1] = TERMINATION;
}

/**
 * Perform the pop operation and update the output expression.
 */
```

```
void popAndUpdateOutput() {
    char poppedChar = pop();
    while (poppedChar != LEFT_PARENTHESIS) {
        outputToExpression(poppedChar);
        poppedChar = pop();
    }
}

/**
 * Perform pop and push operation using operator precedence.
 */
void popAndPushOutput(char inputOperator) {
    char topOperator = pop();
    while (precedenceOf(topOperator) >= precedenceOf(inputOperator)) {
        outputToExpression(topOperator);
        topOperator = pop();
    }
    push(topOperator); // One more operator has been taken out.
    push(inputOperator);
}

/**
 * Display the input and output states.
 */
void displayOutputState() {
    printf("\n%16s | %16s | %16d", expression.operatorStack, expression.postfixExpression,
expression.topOfStack);
}

/**
 * Display the title for the output.
 */
void displayTitle() {
    printf("\n%16s | %16s | %16s", "Operator Stack", "Postfix Expression", "Top of Stack");
    printf("\n----------------------------------------------------------------------");
}

/**
 * Display final output.
 */
void displayPostfixOutput() {
    printf("\n----------------------------------------------------------------------");
    printf("\nFinal Postfix Expression: %s", expression.postfixExpression);
    printf("\n----------------------------------------------------------------------\n");
}

/**
 * Convert Infix expression to Postfix expression.
 */
void convertToPostfix() {
    char inputChar = scanInput();
    while (inputChar != TERMINATION) {
        if (inputChar == LEFT_PARENTHESIS) push(inputChar);
```

```c
        else if (isOperand(inputChar)) outputToExpression(inputChar);
        else if (isOperator(inputChar)) popAndPushOutput(inputChar);
        else if (inputChar == RIGHT_PARENTHESIS) popAndUpdateOutput();
        else invalidExpression();
        displayOutputState();
        inputChar = scanInput();
    }
}

/**
 * Initialize the expression representation. The input expression always have
 * new line character at the end of it. (This is from the use of getline function.)
 * So, it needs to be handled in initialize function.
 */
void initializeExpressionStack() {
    expression.topOfStack = 0;
    push(LEFT_PARENTHESIS);
    expression.postfixExpression[0] = TERMINATION;
    inputExpression[strlen(inputExpression) - 1] = RIGHT_PARENTHESIS;
    inputExpression[strlen(inputExpression)] = TERMINATION;
}

/**
 * Operation for infix to postfix conversion.
 */
void operationOfInfixToPostfix() {
    readInputExpression();
    initializeExpressionStack();
    displayTitle();
    convertToPostfix();
    displayPostfixOutput();
}

/**
 * Initialize evaluation stack.
 */
void initializeEvaluationStack() {
    evaluation.topOfStack = -1;
}

/**
 * Check if the operand is already added or not.
 */
int isAlreadyAdded(int index, char inputVariable) {
    int found = FALSE;
    for (int current = 0; current <= index; current++) {
        if (evaluation.inputVariable[current] == inputVariable) {
            found = TRUE;
            break;
        }
    }
    return found;
}
```

```c
/**
 * Returns the value input for a variable.
 */
double inputValueOf(char operand) {
    int index = -1;
    for (int current = 0; current < strlen(evaluation.inputVariable); current++) {
        if (evaluation.inputVariable[current] == operand) {
            index = current;
            break;
        }
    }
    return evaluation.values[index];
}

/**
 * Read input values for all the operands.
 */
void readInputValues() {
    int index = -1;
    for (int current = 0; current < strlen(expression.postfixExpression); current++) {
        char inputChar = expression.postfixExpression[current];
        if (isOperand(inputChar) && !isAlreadyAdded(index,
expression.postfixExpression[current])) {
            evaluation.inputVariable[++index] = inputChar;
            evaluation.inputVariable[index + 1] = TERMINATION;
            printf("Input '%c': ", inputChar);
            scanf("%lf", &evaluation.values[index]);
        }
    }
}

/**
 * Postfix index to scan from left to right.
 */
int postfixIndex = 0;

/**
 * Scan from postfix expression.
 */
char scanPostfix() {
    if (postfixIndex >= strlen(expression.postfixExpression)) return TERMINATION;
    else return expression.postfixExpression[postfixIndex++];
}

/**
 * Push operand to evaluation stack.
 */
void pushEvaluation(double operandValue) {
    evaluation.stack[++evaluation.topOfStack] = operandValue;
}

/**
```

```
 * Pop operand for evaluation.
 */
double popEvaluation() {
    return evaluation.stack[evaluation.topOfStack--];
}

/**
 * Pop two operands and perform evaluation here.
 */
void popEvaluateAndPush(char operatorChar) {
    double operandTwo = popEvaluation();
    double operandOne = popEvaluation();
    if (operandOne == TERMINATION || operandTwo == TERMINATION) invalidExpression();
    else {
        switch (operatorChar) {
            case PLUS:
                pushEvaluation(operandOne + operandTwo);
                break;
            case MINUS:
                pushEvaluation(operandOne - operandTwo);
                break;
            case MULTIPLY:
                pushEvaluation(operandOne * operandTwo);
                break;
            case DIVIDE:
                if (operandTwo == 0.0) divideByZeroError();
                pushEvaluation(operandOne / operandTwo);
                break;
            case EXPONENT:
                pushEvaluation(pow(operandOne, operandTwo));
                break;
        }
    }
}

/**
 * Evaluate Postfix Expression.
 */
void evaluateExpression() {
    char inputChar = scanPostfix();
    while (inputChar != TERMINATION) {
        if (isOperand(inputChar)) pushEvaluation(inputValueOf(inputChar));
        else if (isOperator(inputChar)) popEvaluateAndPush(inputChar);
        else invalidExpression();
        inputChar = scanPostfix();
    }
}

/**
 * Returns the final output from top of the evaluation stack.
 * @return topOfStackEvaluation
 */
double topOfEvaluationStack() {
    return evaluation.stack[evaluation.topOfStack];
```

```c
}


/**
 * Display Final Result.
 */
void displayEvaluationResult() {
    printf("\n----------------------------------------------------------------------");
    printf("\nFinal Evaluation Output: %.2lf", topOfEvaluationStack());
    printf("\n----------------------------------------------------------------------\n");
}

/**
 * Evaluation of expression.
 */
void evaluatePostfixExpression() {
    initializeEvaluationStack();
    readInputValues();
    evaluateExpression();
    displayEvaluationResult();
}

/**
 * The main program starts here.
 * @return 1 for successful exit.
 */
int main() {
    operationOfInfixToPostfix();
    evaluatePostfixExpression();
    return 1;
}
```

Output:

```
Input Expression:
((a-(b+c))*d)^(e+f)

  Operator Stack | Postfix Expression |     Top of Stack
------------------------------------------------------------------------
              (( |                    |               2
             ((( |                    |               3
             ((( |                  a |               3
            (((- |                  a |               4
           (((-( |                  a |               5
           (((-( |                 ab |               5
          (((-(+ |                 ab |               6
          (((-(+ |                abc |               6
            (((- |               abc+ |               4
              (( |              abc+- |               2
             ((* |              abc+- |               3
             ((* |             abc+-d |               3
               ( |            abc+-d* |               1
              (^ |          abc+-d* |                2
             (^( |          abc+-d* |                3
             (^( |         abc+-d*e |                3
            (^(+ |         abc+-d*e |                4
            (^(+ |        abc+-d*ef |                4
              (^ |       abc+-d*ef+ |                2
                 |      abc+-d*ef+^ |                0
----------------------------------------------------------------------
Final Postfix Expression: abc+-d*ef+^
----------------------------------------------------------------------
Input 'a': 8
Input 'b': 3
Input 'c': 1
Input 'd': 8
Input 'e': 1
Input 'f': 1

----------------------------------------------------------------------
Final Evaluation Output: 1024.000000
----------------------------------------------------------------------
```