**Code:**

#include <iostream>

#include <string>

#include <cmath>

using namespace std;

//character stack class

class *StackChar*

{

    char \*arr;

    int top;

    int maxSize;

public:

    //constructor

    //size of array will be 30 else what user gives

    StackChar(int *ms* = 30)

    {

        arr = **new** char[*ms*];

        maxSize = *ms*;

        top = -1;

    }

    //checking if the stack is empty or not

    bool isEmpty()

    {

        return top < 0;

    }

    //checking if the stack is full or not

    bool isFull()

    {

        return top == (maxSize - 1);

    }

    void Push(char *element*)

    {

        if (!isFull())

        {

            //pushing the character in the array

            arr[++top] = *element*;

        }

    }

    char Pop()

    {

        /\*list is not empty and we have to remove

            \* the element at the top and

            \* return the element

            \*/

        char element = ' ';

        if (!isEmpty())

        {

            element = arr[top];

            arr[top] = -1;

            top--;

        }

        return element;

    }

    void Clear()

    {

        //reinitilazing the stack from the start

        top = -1;

    }

    //returns the value of the peak element to n if stack is not empty

    void peak(char \**n*)

    {

        if (isEmpty())

        {

        }

        \**n* = arr[top];

    };

};

//integer stack class

class *StackInt*

{

    int \*arr;

    int top;

    int maxSize;

public:

    //constructor

    //size of array will be 30 else what user gives

    StackInt(int *ms* = 30)

    {

        arr = **new** int[*ms*];

        maxSize = *ms*;

        top = -1;

    }

    //checking if the stack is empty or not

    bool isEmpty()

    {

        return top < 0;

    }

    //checking if the stack is full or not

    bool isFull()

    {

        return top == (maxSize - 1);

    }

    void Push(int *element*)

    {

        if (!isFull())

        {

            //pushing the character in the array

            arr[++top] = *element*;

        }

        else

            cout << "Stack is already Full" << endl;

    }

    int Pop()

    {

        /\*list is not empty and we have to remove

        \* the element at the top and

        \* return the element

        \*/

        int element = NULL;

        if (!isEmpty())

        {

            element = arr[top];

            arr[top] = -1;

            top--;

        }

        return element;

    }

    void Clear()

    {

        //reinitilazing the stack from the start

        top = -1;

    }

    //returns the value of the peak element to n if stack is not empty

    void peak(int \**n*)

    {

        if (isEmpty())

        {

            return;

        }

        \**n* = arr[top];

    };

};

//checks the opening brackets

bool isOpeningBracket(char *c*)

{

    return (*c* == '(' || *c* == '{' || *c* == '[');

}

//checks closing brackets

bool isClosingBracket(char *c*)

{

    return (*c* == ')' || *c* == '}' || *c* == ']');

}

//checks if character is a operator or not

bool isOperator(char *c*)

{

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

}

//checking the precedence of the operators to see if we have to remove operators form stack or not

bool correctPrecedence(char *top*, char *o*)

{

    return (isOpeningBracket(*o*) || isOpeningBracket(*top*) || ((*o* == '/' || *o* == '\*') && (*top* == '+' || *top* == '-')) || (*o* == '^' && *top* != '^'));

}

//in development feature to check all kinds of brackets

void isSqClBracket(char *bracket*)

{

    if (*bracket* == ']')

    {

    }

}

void isFlClBracket(char *bracket*)

{

    if (*bracket* == '}')

    {

    }

}

void isCClBracket(char *bracket*)

{

    if (*bracket* == ')')

    {

    }

}

//Infix to Postfix Class

class *InfixToPostfix*

{

    //variables to use in Infix to postfix Evaluation

*StackInt* outputStack;

*string* number;

    double above, below, result;

    //variables to use in Infix to postfix conversation

*StackChar* operatorStack;

*string* postFixTerm = "", temp = "";

    char top\_char = ' ', temp\_char\_1 = ' ';

    //public functions

public:

*string* InfixToPostfixConversation(*string* *infixExpression*)

    {

        //loop to read all the characters

        for (int i = 0; i <= *infixExpression*.size(); i++)

        {

            //till where it is a digit it will be stored in a variable

            if (isdigit(*infixExpression*[i]))

            {

                //incrementing

                temp += *infixExpression*[i];

            }

            else

            {

                //appending the number to the postfixterm from the temo

                //and adding a whitespace in after it

                postFixTerm += temp;

                postFixTerm += " ";

                temp = "";

                //getting the top character of the stack and using it

                operatorStack.peak(&top\_char);

                //if string character is a operator and its precedence is correct

                //push it into the stack

                if (correctPrecedence(top\_char, *infixExpression*[i]))

                {

                    operatorStack.Push(*infixExpression*[i]);

                }

                //if there is a closing bracket in the string expression

                //pop all the operators one by one till closing bracket comes

                else if (isClosingBracket(*infixExpression*[i]))

                {

                    //if the character pooped is opening bracket and there was no operator in between them

                    //then loop does not run

                    //else loop pops operators one by one and add them in stack

                    temp\_char\_1 = operatorStack.Pop();

                    while (!isOpeningBracket(temp\_char\_1))

                    {

                        //incrementing the operator in the output string from stack

                        postFixTerm += temp\_char\_1;

                        postFixTerm += ' ';

                        temp\_char\_1 = operatorStack.Pop();

                    }

                }

                else

                {

                    //when the operator precdence is such that operator of

                    //equal precedence is in input expression and stack top

                    if (!correctPrecedence(top\_char, *infixExpression*[i]) || *infixExpression*[i] == top\_char)

                    {

                        postFixTerm += operatorStack.Pop();

                        postFixTerm += ' ';

                    }

                    //when the operator precdence is such that operator of

                    //greater precedence is in input expression and stack top

                    if (!correctPrecedence(top\_char, *infixExpression*[i]))

                    {

                        //while the correct precedence doesnot come it pops the elements form the stack

                        // and pushes them in the string output

                        while (!operatorStack.isEmpty() && !correctPrecedence(top\_char, *infixExpression*[i]))

                        {

                            postFixTerm += operatorStack.Pop();

                            postFixTerm += ' ';

                            operatorStack.peak(&top\_char);

                        }

                        operatorStack.Push(*infixExpression*[i]);

                    }

                }

            }

        }

        //returning the postfix term

        return postFixTerm;

    }

    //to calculate the expression

    double PostfixEvaluation(*string* *postfixExpression*)

    {

        //loop till it reads all the characters

        for (int i = 0; i < *postfixExpression*.size(); i++)

        {

            //if string character is a digit

            if (isdigit(*postfixExpression*[i]))

            {

                number += *postfixExpression*[i];

            }

            //if string character is a space

            else if (*postfixExpression*[i] == ' ' && number != "")

            {

                //stod is used to convert the string into a double

                //then pushing the converted number into the output stack

                outputStack.Push(stod(number));

                number = "";

            }

            //if string character is an operator

            else if (isOperator(*postfixExpression*[i]))

            {

                //poping the most two above elemets from the stack

                above = outputStack.Pop();

                below = outputStack.Pop();

                /\*

                evaluating different conditions of the operators and evaluating

                the answers and pushing it again in the stack to continue the operations

                \*/

                //for sum

                if (*postfixExpression*[i] == '+')

                {

                    result = above + below;

                }

                // for subtraction

                else if (*postfixExpression*[i] == '-')

                {

                    result = below - above;

                }

                //for multiplication

                else if (*postfixExpression*[i] == '\*')

                {

                    result = above \* below;

                }

                //for divison

                else if (*postfixExpression*[i] == '/')

                {

                    result = below / above;

                }

                //for power

                else if (*postfixExpression*[i] == '^')

                {

                    result = pow(below, above);

                }

                //pushing the result in the output stack

                outputStack.Push(result);

            }

        }

        return result;

    }

};

//Function to remove the all the white spaces form the expression

template <typename *T*, typename *P*>

*T* remove\_if(*T* *beg*, *T* *end*, *P* *pred*)

{

*T* dest = *beg*;

    for (*T* itr = *beg*; itr != *end*; ++itr)

        if (!*pred*(\*itr))

            \*(dest++) = \*itr;

    return dest;

}

int main()

{

*InfixToPostfix* infixToPostfix;

*string* infix , postfix;

    double result;

    //getting the infix expression from the user

    cout << "Enter the infix expression:  ";

    getline(cin, infix);

    //removing all the whitespaces form the infix input

    infix.erase(remove\_if(infix.begin(), infix.end(), ::isspace), infix.end());

    //printing the infix expression

    cout << "The infix without spaces is :  " << infix << endl;

    //printing the postfix expression

    postfix = infixToPostfix.InfixToPostfixConversation(infix);

    cout << "The PostFix Expression is :  " << postfix << endl;

    //printing the answer

    result = infixToPostfix.PostfixEvaluation(postfix);

    cout << "The evaluated answer is :" << result << endl;

    return 0;

}

**Text

Description automatically generated**