PIMPRI CHINCHWAD EDUCATION TRUST's.
PIMPRI CHINCHWAD COLLEGE OF ENGINEERING
(An Autonomous Institute)

Class : SY BTech **Acad. Yr. 2025-26** **Semester : I**

Name of the student: Hariom Shrikrishna Gundale **PRN : 124B1B036**

Department: Computer Engineering **Division : A**

Course Name : Data Structures Laboratory **Course Code:BCE23PC02**

Completion Date : 26/10/2025

Assignment No: 11

Problem Statement: Write a C++ program to perform expression conversion and evaluation using stack operations.

The program should include:

- Checking for balanced parentheses
- Infix to Postfix and Infix to Prefix conversion
- Postfix to Infix and Prefix to Infix conversion
- Postfix and Prefix expression evaluation using stacks

Source Code:

```
#include <iostream>
#include <string>
using namespace std;

#define MAX 100

// ---------------- Stack for char ----------------
class StackChar
{
   char arr[MAX];
   int top;

public:
```

```
    StackChar() { top = -1; }
    bool isEmpty() { return top == -1; }
    bool isFull() { return top == MAX - 1; }
    void push(char ch)
    {
        if (!isFull())
            arr[++top] = ch;
    }
    char pop()
    {
        if (!isEmpty())
            return arr[top--];
        return '\0';
    }
    char peek()
    {
        if (!isEmpty())
            return arr[top];
        return '\0';
    }
};

// ---------------- Stack for int ----------------
class StackInt
{
    int arr[MAX];
    int top;

public:
    StackInt() { top = -1; }
    bool isEmpty() { return top == -1; }
    void push(int val) { arr[++top] = val; }
    int pop() { return arr[top--]; }
    int peek() { return arr[top]; }
};

// ---------------- Parenthesis Check ----------------
bool isMatchingPair(char a, char b)
```

```
{
    return (a == '(' && b == ')') || (a == '{' && b == '}') || (a == '[' && b == ']');
}
bool isBalanced(string expr)
{
    StackChar s;
    for (char ch : expr)
    {
        if (ch == '(' || ch == '{' || ch == '[')
            s.push(ch);
        else if (ch == ')' || ch == '}' || ch == ']')
        {
            if (s.isEmpty() || !isMatchingPair(s.pop(), ch))
                return false;
        }
    }
    return s.isEmpty();
}

// ---------------- Infix to Postfix ----------------
int precedence(char op)
{
    if (op == '+' || op == '-')
        return 1;
    if (op == '*' || op == '/')
        return 2;
    if (op == '^')
        return 3;
    return 0;
}
bool isOperator(char ch)
{
    return ch == '+' || ch == '-' || ch == '*' || ch == '/' || ch == '^';
}
string infixToPostfix(string infix)
{
    StackChar s;
    string postfix = "";
```

```
    for (char ch : infix)
    {
        if ((ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z') || (ch >= '0' && ch <= '9'))
            postfix += ch;
        else if (ch == '(')
            s.push(ch);
        else if (ch == ')')
        {
            while (!s.isEmpty() && s.peek() != '(')
                postfix += s.pop();
            s.pop();
        }
        else if (isOperator(ch))
        {
            while (!s.isEmpty() && precedence(ch) <= precedence(s.peek()))
                postfix += s.pop();
            s.push(ch);
        }
    }
    while (!s.isEmpty())
        postfix += s.pop();
    return postfix;
}

// ---------------- Infix to Prefix ----------------
string reverseString(string s)
{
    string r = "";
    for (int i = s.size() - 1; i >= 0; i--)
        r += s[i];
    return r;
}
string infixToPrefix(string infix)
{
    string rev = reverseString(infix);
    for (int i = 0; i < rev.size(); i++)
    {
        if (rev[i] == '(')
```

```
            rev[i] = ')';
        else if (rev[i] == ')')
            rev[i] = '(';
    }
    string post = infixToPostfix(rev);
    return reverseString(post);
}

// ---------------- Postfix to Infix ----------------
string postfixToInfix(string postfix)
{
    string stack[MAX];
    int top = -1;
    for (char ch : postfix)
    {
        if ((ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z') || (ch >= '0' && ch <= '9'))
            stack[++top] = string(1, ch);
        else
        {
            string op2 = stack[top--];
            string op1 = stack[top--];
            stack[++top] = "(" + op1 + ch + op2 + ")";
        }
    }
    return stack[top];
}

// ---------------- Prefix to Infix ----------------
string prefixToInfix(string prefix)
{
    string stack[MAX];
    int top = -1;
    for (int i = prefix.size() - 1; i >= 0; i--)
    {
        char ch = prefix[i];
        if ((ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z') || (ch >= '0' && ch <= '9'))
            stack[++top] = string(1, ch);
        else
```

```
        {
            string op1 = stack[top--];
            string op2 = stack[top--];
            stack[++top] = "(" + op1 + ch + op2 + ")";
        }
    }
    return stack[top];
}

// ---------------- Postfix Evaluation ----------------
int postfixEval(string postfix)
{
    StackInt s;
    for (char ch : postfix)
    {
        if (ch >= '0' && ch <= '9')
            s.push(ch - '0');
        else
        {
            int b = s.pop();
            int a = s.pop();
            if (ch == '+')
                s.push(a + b);
            else if (ch == '-')
                s.push(a - b);
            else if (ch == '*')
                s.push(a * b);
            else if (ch == '/')
                s.push(a / b);
        }
    }
    return s.pop();
}

// ---------------- Prefix Evaluation ----------------
int prefixEval(string prefix)
{
    StackInt s;
```

```
    for (int i = prefix.size() - 1; i >= 0; i--)
    {
        char ch = prefix[i];
        if (ch >= '0' && ch <= '9')
            s.push(ch - '0');
        else
        {
            int a = s.pop();
            int b = s.pop();
            if (ch == '+')
                s.push(a + b);
            else if (ch == '-')
                s.push(a - b);
            else if (ch == '*')
                s.push(a * b);
            else if (ch == '/')
                s.push(a / b);
        }
    }
    return s.pop();
}

// ---------------- Main ----------------
int main()
{
    string expr;
    cout << "Enter expression: ";
    cin >> expr;

    cout << "\n--- Parenthesis Check ---\n";
    cout << (isBalanced(expr) ? "Balanced\n" : "Not Balanced\n");

    cout << "\n--- Infix to Postfix ---\n";
    string postfix = infixToPostfix(expr);
    cout << "Postfix: " << postfix << "\n";

    cout << "\n--- Infix to Prefix ---\n";
    string prefix = infixToPrefix(expr);
```

```
    cout << "Prefix: " << prefix << "\n";

    cout << "\n--- Postfix to Infix ---\n";
    cout << "Infix: " << postfixToInfix(postfix) << "\n";

    cout << "\n--- Prefix to Infix ---\n";
    cout << "Infix: " << prefixToInfix(prefix) << "\n";

    cout << "\n--- Postfix Evaluation ---\n";
    cout << "Result: " << postfixEval(postfix) << "\n";

    cout << "\n--- Prefix Evaluation ---\n";
    cout << "Result: " << prefixEval(prefix) << "\n";

    return 0;
}
```

Screen Shot of Output :

```
PS P:\DSA_Asssignment> g++ Assignment_11.cpp -o Assignment_11
PS P:\DSA_Asssignment> ./Assignment_11
Enter expression: ((5+2)*(8-3)/(4+1))

--- Parenthesis Check ---
Balanced

--- Infix to Postfix ---
Postfix: 52+83-*41+/

--- Infix to Prefix ---
Prefix: *+52/-83+41

--- Postfix to Infix ---
Infix: (((5+2)*(8-3))/(4+1))

--- Prefix to Infix ---
Infix: ((5+2)*((8-3)/(4+1)))

--- Postfix Evaluation ---
Result: 7

--- Prefix Evaluation ---
Result: 7
```

Conclusion:

Thus, we have successfully implemented the C++ Program to perform expression conversion and evaluation using stack operations.