Tutorial 5

Name: Shubham Santosh Dorake
PRN: B24CE1042
DIV: CE1-B

Title : Parenthesis Checker Problem

Statement : Write a program using a stack for push, pop, peek, and isEmpty operations. Write isBalanced() Function that iterates through the input expression, pushes opening brackets onto the stack. For closing brackets, it checks the top of the stack for a matching opening bracket. Ensures that all opening brackets are matched by the end of the traversal. Main Function: Accepts a string expression from the user. Uses isBalanced() to determine if the parentheses in the expression are balanced.

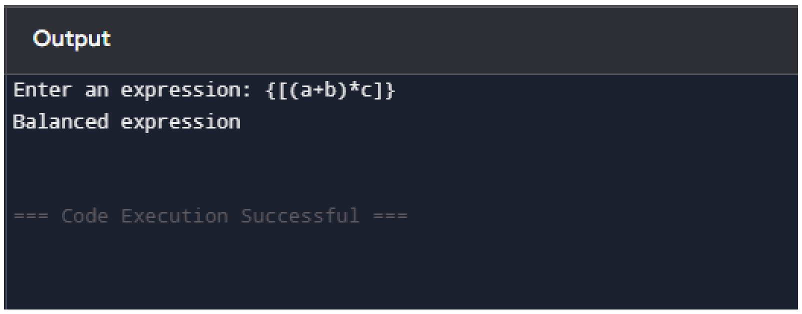
Code :

```
#include <iostream>
#include <string>
using namespace std;
#define MAX 100   // Maximum size of stack
class Stack {
char arr[MAX];
int top;
public:
Stack() {
top = -1;
}
void push(char c) {
if (top == MAX - 1) {
cout << "Stack overflow!" << endl;
```

```
} else {
arr[++top] = c;
}
}
void pop() {
if (top == -1) {
cout << "Stack underflow!" << endl;
} else {
top--;
}
}
char peek() {
if (top == -1)
return '\0';
else
return arr[top];
}
bool isEmpty() {
return (top == -1);
}
};
bool isMatchingPair(char open, char close) {
return (open == '(' && close == ')') ||
(open == '{' && close == '}') ||
(open == '[' && close == ']');
}
bool isBalanced(string expr) {
Stack s;
for (char ch : expr) {
if (ch == '(' || ch == '{' || ch == '[') {
s.push(ch);
}
else if (ch == ')' || ch == '}' || ch == ']') {
if (s.isEmpty() || !isMatchingPair(s.peek(), ch))
return false;
```

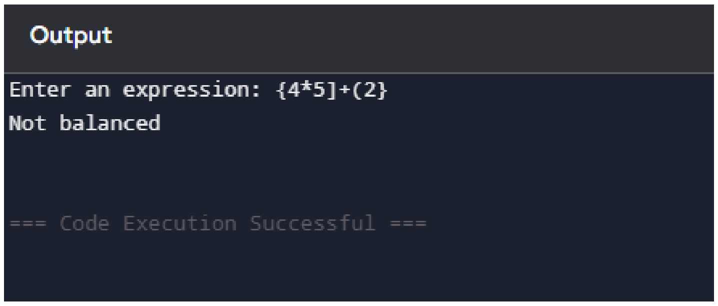
```
s.pop();
}
}
return s.isEmpty();
}
int main() {
string expression;
cout << "Enter an expression: ";
cin >> expression;
if (isBalanced(expression))
cout << "Balanced expression" << endl;
else
cout << "Not balanced" << endl;
return 0;
}
```

Output :



Output

```
Enter an expression: {[(a+b)*c]}
Balanced expression


=== Code Execution Successful ===
```



Output

```
Enter an expression: {4*5]+(2}
Not balanced


=== Code Execution Successful ===
```

Title : Syntax Parsing in Programming Languages Problem
Statement : Parsing expressions is a key step in many compilers and language processors. When a language's syntax requires parsing mathematical or logical expressions, converting between infix and postfix notation ensures that expressions are evaluated correctly. Accept an infix expression and show the expression in postfix form.

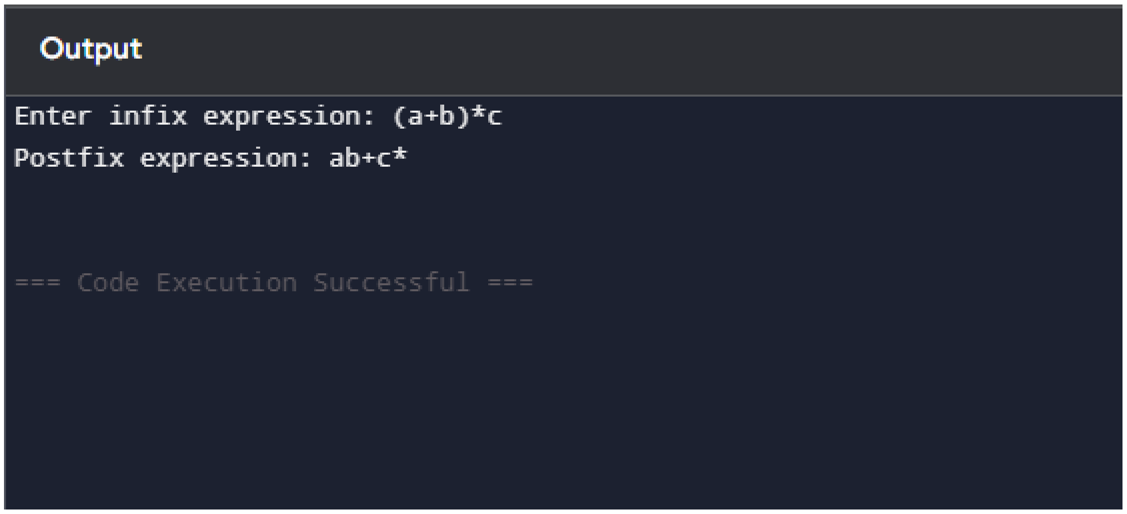
Code :

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

#define MAX 100

class Stack {
   char arr[MAX];
   int top;

public:
   Stack() { top = -1; }

   void push(char c) {
      if (top == MAX - 1) {
         cout << "Stack overflow!" << endl;
      } else {
         arr[++top] = c;
      }
   }

   void pop() {
      if (top == -1) {
         cout << "Stack underflow!" << endl;
      } else {
         top--;
      }
   }
```

```
    char peek() {
        if (top == -1)
            return '\0';
        return arr[top];
    }

    bool isEmpty() {
        return (top == -1);
    }
};

// Function to define operator precedence
int precedence(char c) {
    if (c == '^')
        return 3;
    else if (c == '*' || c == '/')
        return 2;
    else if (c == '+' || c == '-')
        return 1;
    else
        return -1;
}

// Function to convert infix to postfix
string infixToPostfix(string s) {
    Stack st;
    string result = "";

    for (int i = 0; i < s.length(); i++) {
        char c = s[i];

        // Operand
        if ((c >= 'a' && c <= 'z') ||
            (c >= 'A' && c <= 'Z') ||
            (c >= '0' && c <= '9')) {
            result += c;
        }

        // Opening bracket
        else if (c == '(') {
            st.push(c);
        }

        // Closing bracket
        else if (c == ')') {
            while (!st.isEmpty() && st.peek() != '(') {
                result += st.peek();
                st.pop();
            }
            if (!st.isEmpty()) st.pop(); // Remove '('
        }

        // Operator
        else {
            // Handle precedence and right-associativity of '^'
            while (!st.isEmpty() &&
                  precedence(st.peek()) >= precedence(c)) {
                if (c == '^' && precedence(st.peek()) == precedence(c))
                    break; // Right-associative operator
                result += st.peek();
                st.pop();
            }
            st.push(c);
        }
    }

    // Pop remaining operators
    while (!st.isEmpty()) {
        result += st.peek();
        st.pop();
    }

    return result;
```

```
}

int main() {
    string exp;
    cout << "Enter infix expression: ";
    cin >> exp;

    cout << "Postfix expression: " << infixToPostfix(exp) << endl;
    return 0;
}
```

Output :



Output

```
Enter infix expression: (a+b)*c
Postfix expression: ab+c*


=== Code Execution Successful ===
```