Government College of Engineering, Karad Programming for Problem Solving Lab

Nanekar Saurabh Rajesh

20141212

I1

**Experiment No. 3**

**Title**: Implement stack as an ADT to perform expression conversion and evaluation for infix to

Postfix.

**Outcome:** Students can perform expression conversion and evaluation for infix to postfix and its related applications.

**Theory:**

One of the applications of Stack is in the conversion of arithmetic expressions in high-level programming languages into machine readable form. As our computer system can only understand and work on a binary language, it assumes that an arithmetic operation can take place in two operands only e.g., **A+B, C\*D,D/A** etc. But in our usual form an arithmetic expression may consist of more than one operator and two operands e.g. **(A+B)\*C(D/(J+D))**.

These complex arithmetic operations can be converted into polish notation using stacks which then can be executed in two operands and an operator form.

## Infix Expression

It follows the scheme of **<operand><operator><operand>** i.e. an <operator> is preceded and succeeded by an <operand>. Such an expression is termed infix expression. E.g., **A+B**

## Postfix Expression

It follows the scheme of **<operand><operand><operator>** i.e. an <operator> is succeeded by both the <operand>. E.g., **AB+**

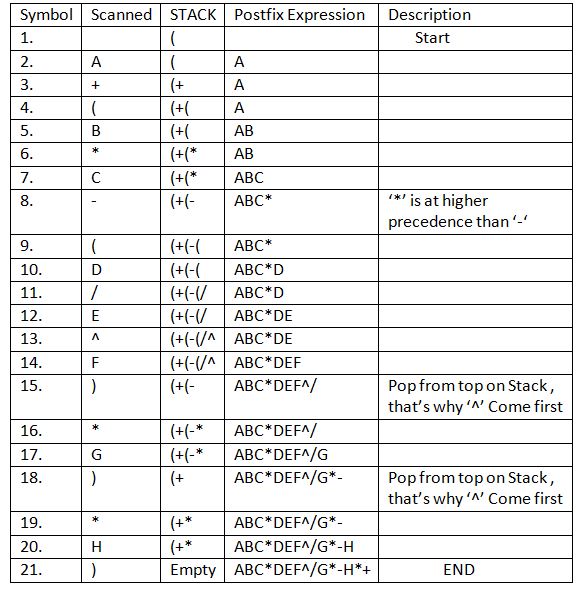
### Algorithm to convert Infix to Postfix

Let, X is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression Y.

1. Push “(“onto Stack, and add “)” to the end of X.
2. Scan X from left to right and repeat Step 3 to 6 for each element of X until the Stack is empty.
3. If an operand is encountered, add it to Y.
4. If a left parenthesis is encountered, push it onto Stack.
5. If an operator is encountered ,then:
   1. Repeatedly pop from Stack and add to Y each operator (on the top of Stack) which has the same precedence as or higher precedence than operator.
   2. Add operator to Stack.  
      [End of If]
6. If a right parenthesis is encountered ,then:
   1. Repeatedly pop from Stack and add to Y each operator (on the top of Stack) until a left parenthesis is encountered.
   2. Remove the left Parenthesis.  
      [End of If]  
      [End of If]

**Let’s take an example to better understand the algorithm**

Infix Expression: **A+ (B\*C-(D/E^F)\*G)\*H**, where **^** is an exponential operator.



**Resultant Postfix Expression: ABC\*DEF^/G\*-H\*+**

The Postfix notation is used to represent algebraic expressions. The expressions written in postfix form are evaluated faster compared to infix notation as parenthesis are not required in postfix.

Following is an algorithm for evaluation postfix expressions.   
1) Create a stack to store operands (or values).   
2) Scan the given expression and do the following for every scanned element.   
 a) If the element is a number, push it into the stack   
 b) If the element is an operator, pop operands for the operator from the stack. Evaluate the operator and push the result back to the stack   
3) when the expression is ended, the number in the stack is the final answer

**Example:**   
Let the given expression be “2 3 1 \* + 9 -“. We scan all elements one by one.   
1) Scan ‘2’, it’s a number, so push it to stack. Stack contains ‘2’   
2) Scan ‘3’, again a number, push it to stack, stack now contains ‘2 3’ (from bottom to top)   
3) Scan ‘1’, again a number, push it to stack, stack now contains ‘2 3 1’   
4) Scan ‘\*’, it’s an operator, pop two operands from stack, apply the \* operator on operands, we get 3\*1 which results in 3. We push the result ‘3’ to stack. The stack now becomes ‘2 3’.   
5) Scan ‘+’, it’s an operator, pop two operands from stack, apply the + operator on operands, we get 3 + 2 which results in 5. We push the result ‘5’ to stack. The stack now becomes ‘5’.   
6) Scan ‘9’, it’s a number, we push it to the stack. The stack now becomes ‘5 9’.   
7) Scan ‘-‘, it’s an operator, pop two operands from stack, apply the – operator on operands, we get 5 – 9 which results in -4. We push the result ‘-4’ to the stack. The stack now becomes ‘-4’.   
8) There are no more elements to scan, we return the top element from the stack (which is the only element left in a stack).

## Advantage of Postfix Expression over Infix Expression

An infix expression is difficult for the machine to know and keep track of precedence of operators. On the other hand, a postfix expression itself determines the precedence of operators (as the placement of operators in a postfix expression depends upon its precedence).Therefore, for the machine it is easier to carry out a postfix expression than an infix expression.

**Analysis:**



**List of similar programs: Solve any one.**

1. Write a program for implementation of Infix to Postfix using different Precedence Values for In-Stack and Out-Stack.
2. Write a program to convert Infix notation to Expression Tree.
3. Write a program [to check for Balanced Brackets in an expression (well-formedness) using Stack](https://www.geeksforgeeks.org/check-for-balanced-parentheses-in-an-expression/?ref=rp" \o "Permalink to Check for Balanced Brackets in an expression (well-formedness) using Stack)

**Title Program:**  Implement stack as an ADT to perform expression conversion and evaluation for infix to Postfix.

**Source code of Implemented Programs:**

//Nanekar Saurabh Rajesh\_20141212\_I1

#include<stdio.h>

char stack[15];

int top=-1;

void push(char x)

{

    stack[++top]=x;;

}

char pop()

{

    if(top==-1)

    return -1;

    else

    return stack[top--];

}

int priority(char x)

{

    if(x=='(')

    return 0;

    if(x=='+'||x=='-')

    return 1;

    if(x=='\*'||x=='/')

    return 2;

    if (x=='^')

    return 3;

}

int main()

{

    char exp[20];

    char \*e,x;

    printf("Enter the expression::");

    scanf("%s",exp);

    e=exp;

    while(\*e!='\0')

    {

        if(isalnum(\*e))

        printf("%c",\*e);

        else if(\*e=='(')

        push(\*e);

        else if(\*e==')')

        {

            while((x=pop())!='(')

            printf("%c",x);

        }

        else

        {

            while(priority(stack[top])>=priority(\*e))

            printf("%c",pop());

            push(\*e);

        }

        e++;

    }

    while(top!=-1)

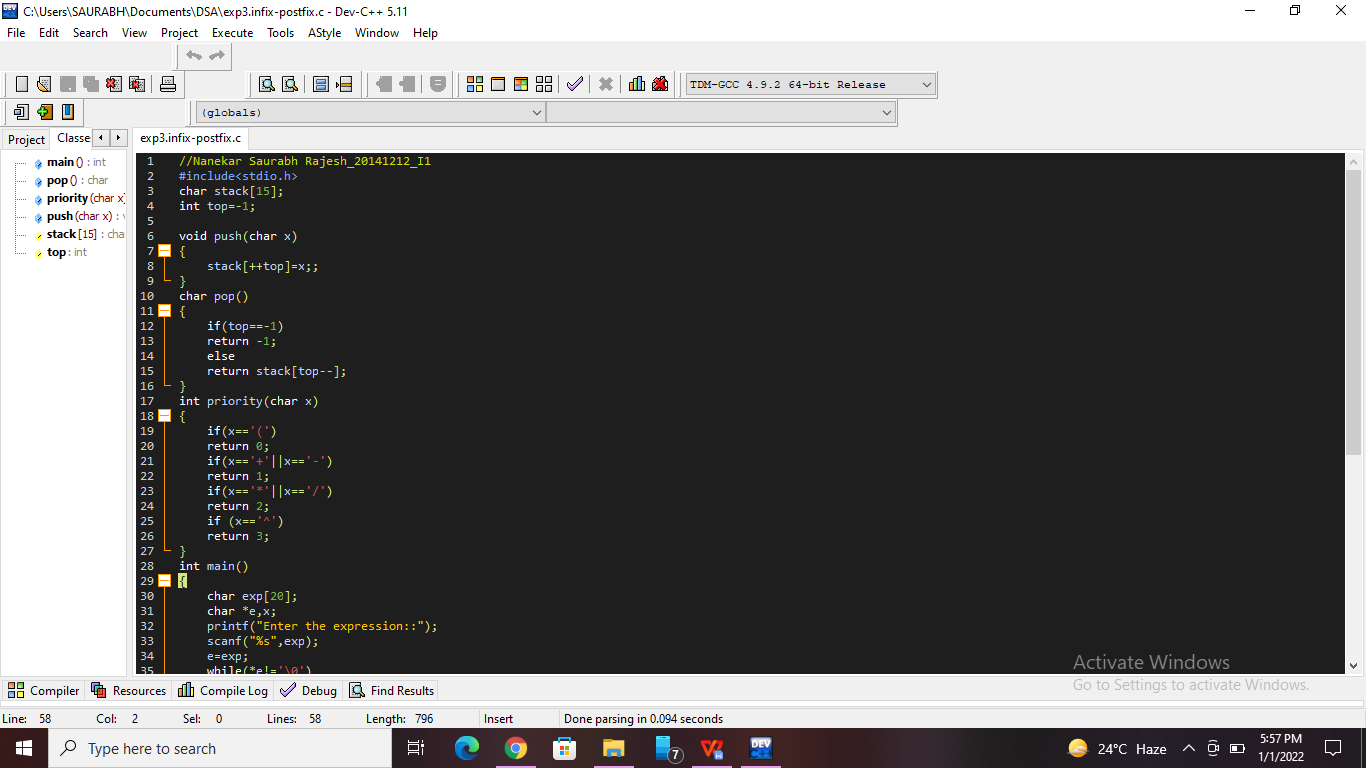
    {

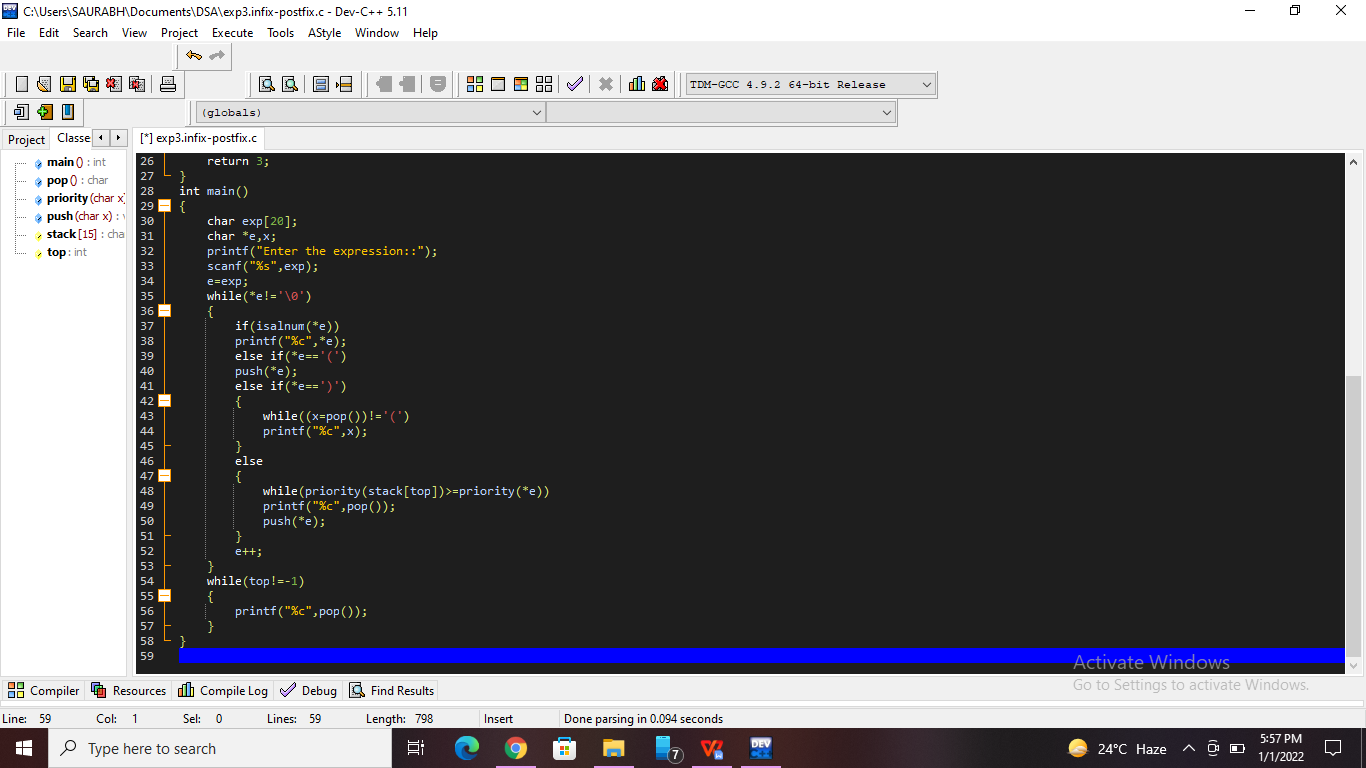
        printf("%c",pop());

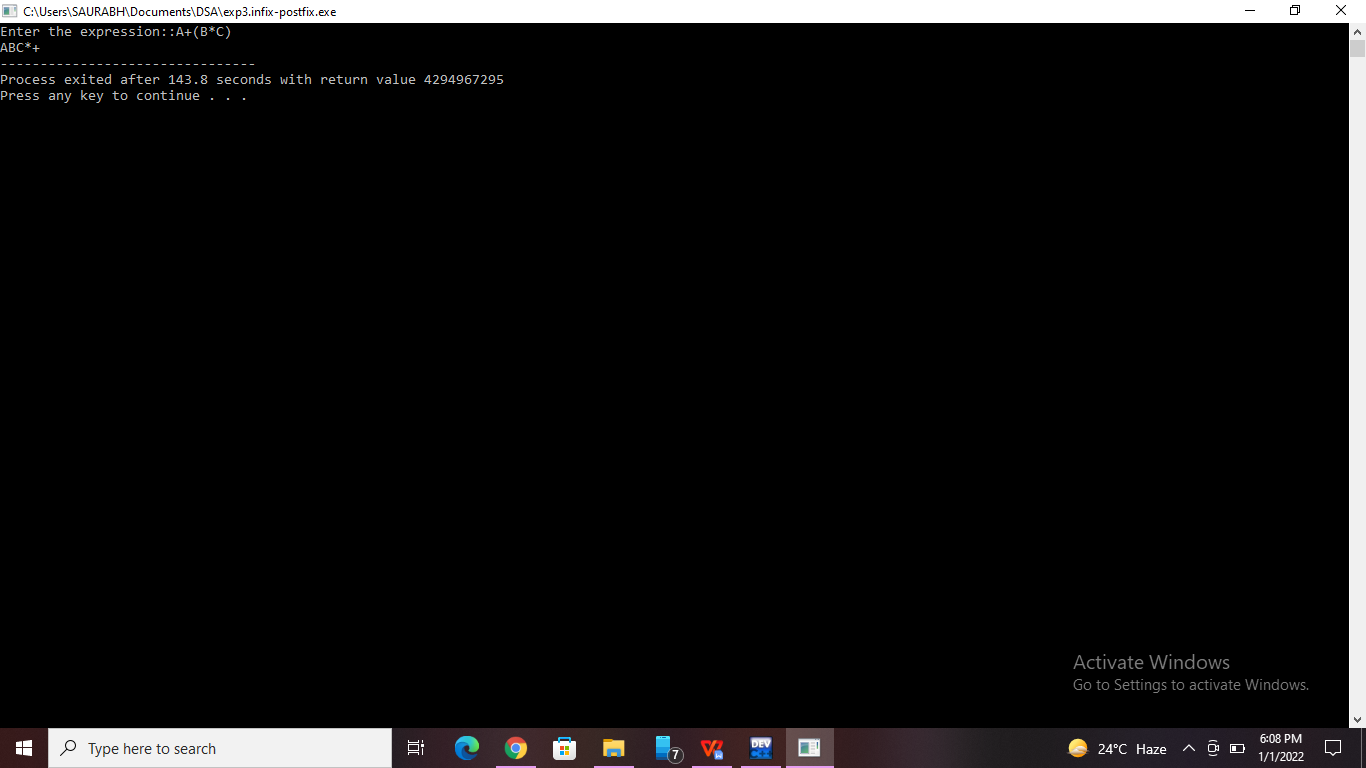
    }

}

**Screenshots of Output:**

****

****

****

**Practice Program:**

Write a program [to check for Balanced Brackets in an expression (well-formedness) using Stack](https://www.geeksforgeeks.org/check-for-balanced-parentheses-in-an-expression/?ref=rp" \o "Permalink to Check for Balanced Brackets in an expression (well-formedness) using Stack)

//Nanekar Saurabh Rajesh

#include<stdio.h>

#include<string.h>

#include<stdlib.h>

#define MAX 30

int top=-1;

int stack[MAX];

void push(char);

char pop();

int match(char a,char b);

int check(char []);

int main()

{

        char exp[MAX];

        int valid;

        printf("Enter an algebraic expression : ");

        gets(exp);

        valid=check(exp);

        if(valid==1)

                printf("Valid expression\n");

        else

                printf("Invalid expression\n");

                return 0;

}

int check(char exp[] )

{

        int i;

        char temp;

        for(i=0;i<strlen(exp);i++)

        {

                if(exp[i]=='(' || exp[i]=='{' || exp[i]=='[')

                        push(exp[i]);

                if(exp[i]==')' || exp[i]=='}' || exp[i]==']')

                        if(top==-1)    /\*stack empty\*/

                        {

                                printf("Right parentheses are more than left parentheses\n");

                                return 0;

                        }

                        else

                        {

                                temp=pop();

                                if(!match(temp, exp[i]))

                                {

                                        printf("Mismatched parentheses are : ");

                                        printf("%c and %c\n",temp,exp[i]);

                                        return 0;

                                }

                        }

        }

        if(top==-1)

        {

                printf("Balanced Parentheses\n");

                return 1;

        }

        else

        {

                printf("Left parentheses more than right parentheses\n");

                return 0;

        }

}

int match(char a,char b)

{

        if(a=='[' && b==']')

                return 1;

        if(a=='{' && b=='}')

                return 1;

        if(a=='(' && b==')')

                return 1;

        return 0;

}

void push(char item)

{

        if(top==(MAX-1))

        {

                printf("Stack Overflow\n");

                return;

        }

        top=top+1;

        stack[top]=item;

}

char pop()

{

        if(top==-1)

        {

                printf("Stack Underflow\n");

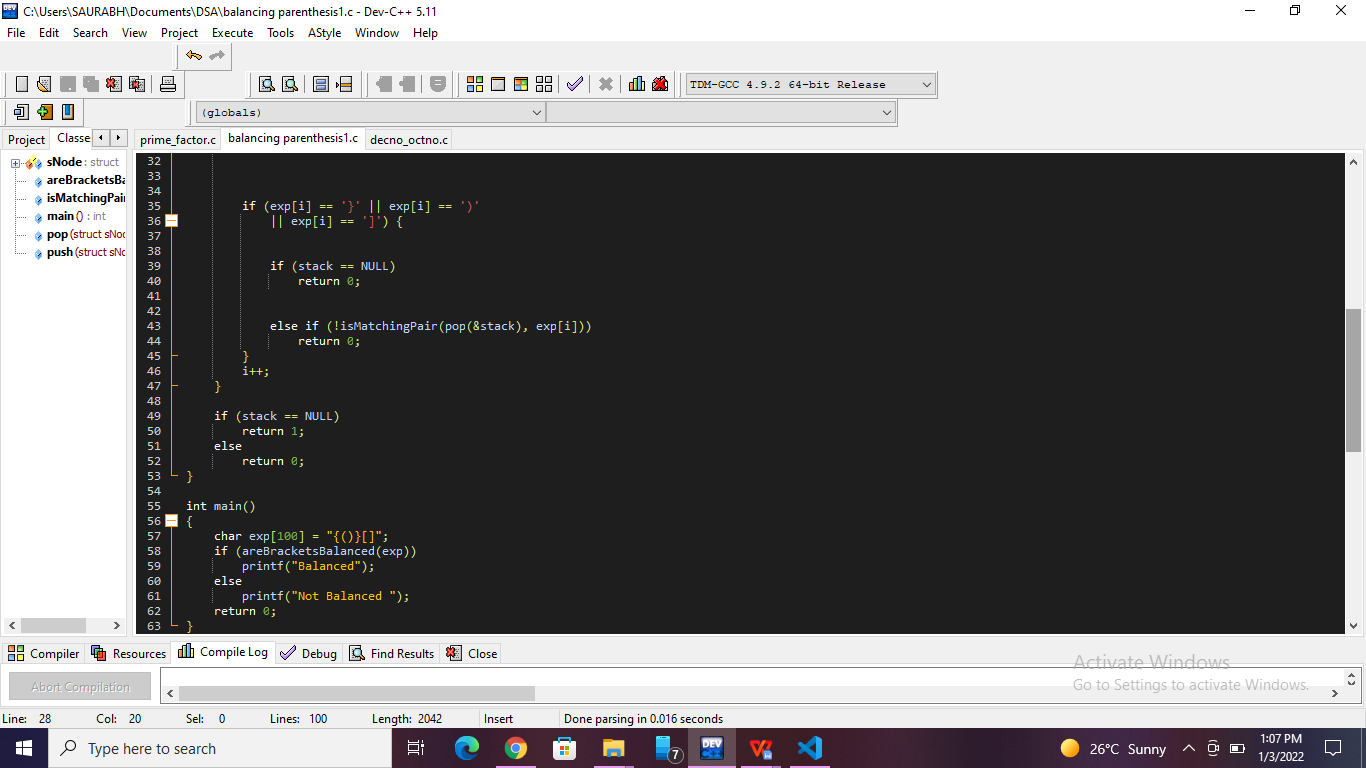
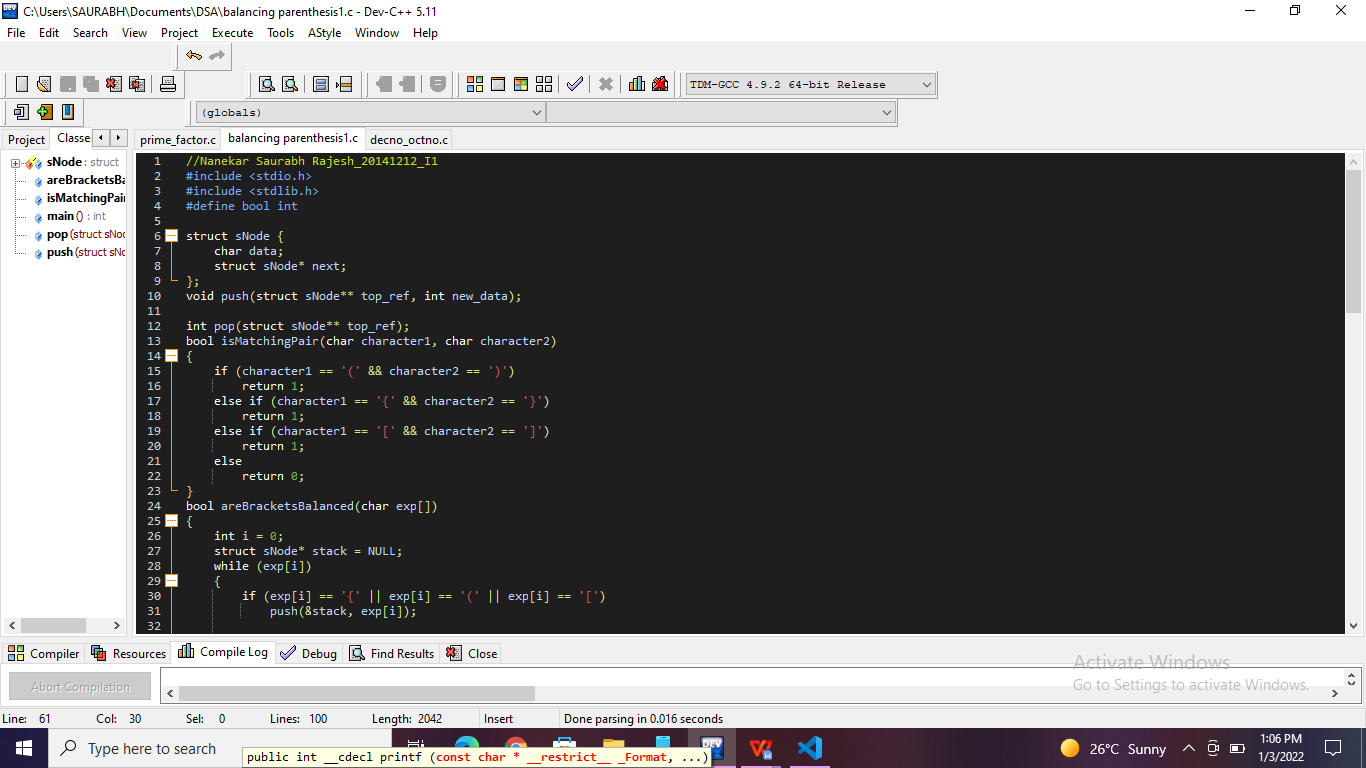
                exit(1);

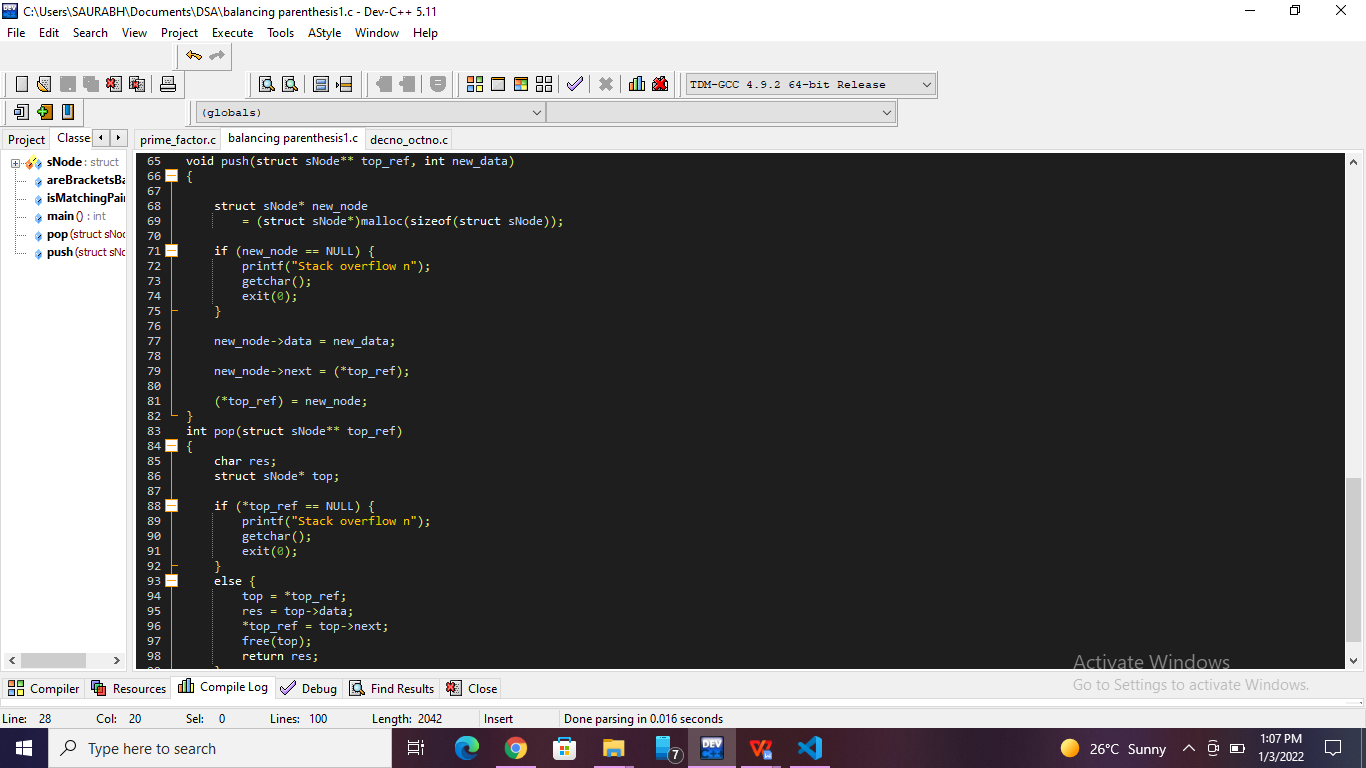
        }

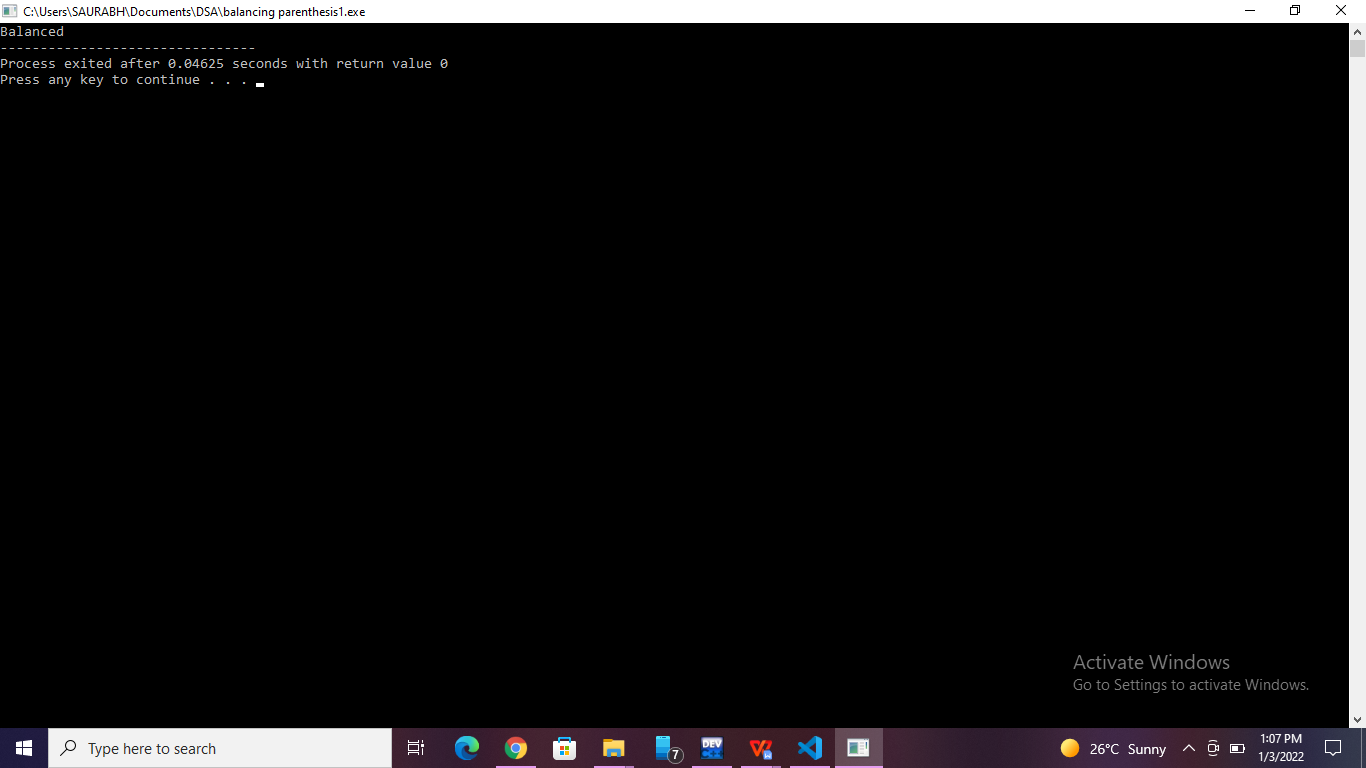
        return(stack[top--]);

}

**Practice Programme Screenshots:-**

****

****

****

**List of sample questions for oral examination:**

1. What are infix, prefix, and postfix notations?
2. What is the need for circular array to implement queue?
3. What is queue full condition if it is implemented with an array?
4. What are the applications of circular queue?
5. How a circular queue can be implemented using array?

**Conclusion:**

Infix expressions are readable and solvable by humans. We can easily distinguish the order of operators, and also can use the parenthesis to solve that part first during solving mathematical expressions. The computer cannot differentiate the operators and parenthesis easily, that's why postfix conversion is needed.