**DSA LAB – 7**

**Name:** Etcherla Sai Manoj **Mis. No:** 112015044 **Branch:** CSE

**Question1:**

**Code:**

#include<iostream>

#include<string.h>

#include<math.h>

using namespace std;

template <class S>

class Stack

{

struct Node

{

S data;

Node \*next;

};

Node \*top;

public:

Stack();

void push(S);

S pop();

S ele\_top();

int is\_empty();

void eval\_reverse();

void show\_exp();

};

template <class S>

Stack<S>::Stack()

{

top=NULL;

}

template <class S>

void Stack<S>::push(S x)

{

Node \*new\_node;

new\_node=new Node;

new\_node->data=x;

new\_node->next=top;

top=new\_node;

}

template <class S>

S Stack<S>::pop()

{

S x;

Node \*temp;

temp=top;

x=temp->data;

top=top->next;

delete temp;

return x;

}

template <class S>

S Stack<S>::ele\_top()

{

return top->data;

}

template <class S>

int Stack<S>::is\_empty()

{

if(top==NULL)

return 1;

return 0;

}

template <class S>

void Stack<S>::eval\_reverse()

{

Node \*prev,\*current;

if(top!=NULL)

{

prev=top;

top=top->next;

current=top;

prev->next=NULL;

while(top!=NULL)

{

top=top->next;

current->next=prev;

prev=current;

current=top;

}

}

top=prev;

}

template <class S>

void Stack<S>::show\_exp()

{

Node \*temp;

temp=top;

while(temp!=NULL)

{

cout<<temp->data<<" ";

temp=temp->next;

}

cout<<"\n";

}

class expression

{

char infix[40],postfix[40],prefix[40];

public:

expression();

int sequence(char);

void prefixexp();

void postfixexp();

void prefixeval();

void postfixeval();

void strrev(char []);

};

expression::expression()

{

infix[0]='\0';

prefix[0]='\0';

postfix[0]='\0';

}

void expression::prefixexp()

{

char ch;

Stack<char> s;

int i, j = 0;

cout<<"Enter the infix expression : ";

cin.ignore();

cin.getline(infix,20);

for (i=strlen(infix)-1;i>=0;i--)

{

switch (infix[i])

{

case ')':

s.push(')');

break;

case '+':

case '-':

case '/':

case '\*':

case '%':

case '^':

while (!s.is\_empty() && sequence(s.ele\_top()) >= sequence(infix[i]))

{

prefix[j] = s.pop();

j++;

}

s.push(infix[i]);

break;

case '(':

ch = s.pop();

while (ch != ')')

{

prefix[j] = ch;

j++;

ch = s.pop();

}

break;

default:

prefix[j] = infix[i];

j++;

}

}

while (!s.is\_empty())

{

prefix[j] = s.pop();

j++;

}

prefix[j] = '\0';

strrev(prefix);

cout<<"Prefix expression is : "<<prefix<<endl;

}

void expression::postfixexp()

{

char ch;

Stack<char> s;

int i, j = 0;

cin.ignore();

cout<<"Enter the infix expression : ";

cin.getline(infix,20);

for (i=0; infix[i] != '\0'; i++)

{

switch (infix[i])

{

case '(':

s.push('(');

break;

case '+':

case '-':

case '/':

case '\*':

case '%':

case '^':

while (!s.is\_empty() && sequence(s.ele\_top()) > sequence(infix[i]))

{

postfix[j] = s.pop();

j++;

}

s.push(infix[i]);

break;

case ')':

ch = s.pop();

while (ch != '(')

{

postfix[j] = ch;

j++;

ch = s.pop();

}

break;

default:

postfix[j] = infix[i];

j++;

}

}

while (!s.is\_empty())

{

postfix[j] = s.pop();

j++;

}

postfix[j] = '\0';

cout<<"Postfix expression is : "<<postfix<<endl;

}

void expression:: prefixeval()

{

Stack<char> s;

int i,j=0,op1,op2,vals[20];

cin.ignore();

cout<<"Enter the prefix expression : ";

cin.getline(prefix,40);

for(i=0;prefix[i]!='\0';i++)

{

if(isalpha(prefix[i]))

{

cout<<"Enter value for operand "<<prefix[i]<<": ";

cin>>vals[j];

j++;

}

if(isdigit(prefix[i]))

{

vals[j]=((int)prefix[i]-48);

j++;

}

}

j--;

for(i=strlen(prefix)-1;i>=0;i--)

{

if(isalpha(prefix[i]))

{

s.push(vals[j]);

j--;

}

else if(isdigit(prefix[i]))

{

s.push(vals[j]);

j--;

}

else

{

op1 = s.pop();

op2 = s.pop();

if (prefix[i] == '+')

s.push(op1+op2);

else if (prefix[i] == '-')

s.push(op1-op2);

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

s.push(op1\*op2);

else if (prefix[i] == '/')

s.push(op1/op2);

else if (prefix[i] == '%')

s.push(op1%op2);

else

s.push(pow(op1,op2));

}

}

cout<<"Result of evaluating expression is "<<(int)s.pop()<<endl;

}

void expression::postfixeval()

{

Stack<char> s;

int i,op1,op2,val;

cin.ignore();

cout<<"Enter the postfix expression : ";

cin.getline(postfix,40);

for(i=0;postfix[i]!='\0';i++)

{

if(isalpha(postfix[i]))

{

cout<<"Enter value for operand "<<postfix[i]<<": ";

cin>>val;

s.push(val);

}

else if(isdigit(postfix[i]))

{

val=(int(postfix[i])-48);

s.push(val);

}

else

{

op2 = s.pop();

op1 = s.pop();

if (postfix[i] == '+')

s.push(op1+op2);

else if (postfix[i] == '-')

s.push(op1-op2);

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

s.push(op1\*op2);

else if (postfix[i] == '/')

s.push(op1/op2);

else if (postfix[i] == '%')

s.push(op1%op2);

else

s.push(pow(op1,op2));

}

}

cout << "Result of evaluating expression is " << (int)s.pop() <<endl;

}

int expression::sequence(char ch)

{

if (ch == '^' || ch == '$')

return 6;

if (ch == '/' || ch == '\*' || ch == '%')

return 5;

if (ch == '+' || ch == '-')

return 4;

return 0;

}

void expression::strrev(char prefix[])

{

Stack<char> s;

int i;

for(i=0;i<strlen(prefix);i++)

s.push(prefix[i]);

for(i=0;i<strlen(prefix);i++)

prefix[i]=s.pop();

prefix[i]='\0';

}

int main()

{

expression e;

Stack<char> s;

int choice,result;

while(1)

{

cout << "\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*MENU\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"<< endl;

cout << "1. Conversion of infix expression to prefix expression"<<endl;

cout << "2. Conversion of infix expression to postfix expression"<<endl;

cout << "3. Evaluation of prefix expression"<<endl;

cout << "4. Evaluation of postfix expression"<<endl;

cout << "5. Exit program"<<endl;

cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"<< endl;

cout<<"\nEnter your choice : ";

cin>>choice;

switch(choice)

{

case 1:

e.prefixexp();

break;

case 2:

e.postfixexp();

break;

case 3:

e.prefixeval();

break;

case 4:

e.postfixeval();

break;

case 5:

return 0;

default:

cout<<"\nError in choice, try again"<<endl;

}

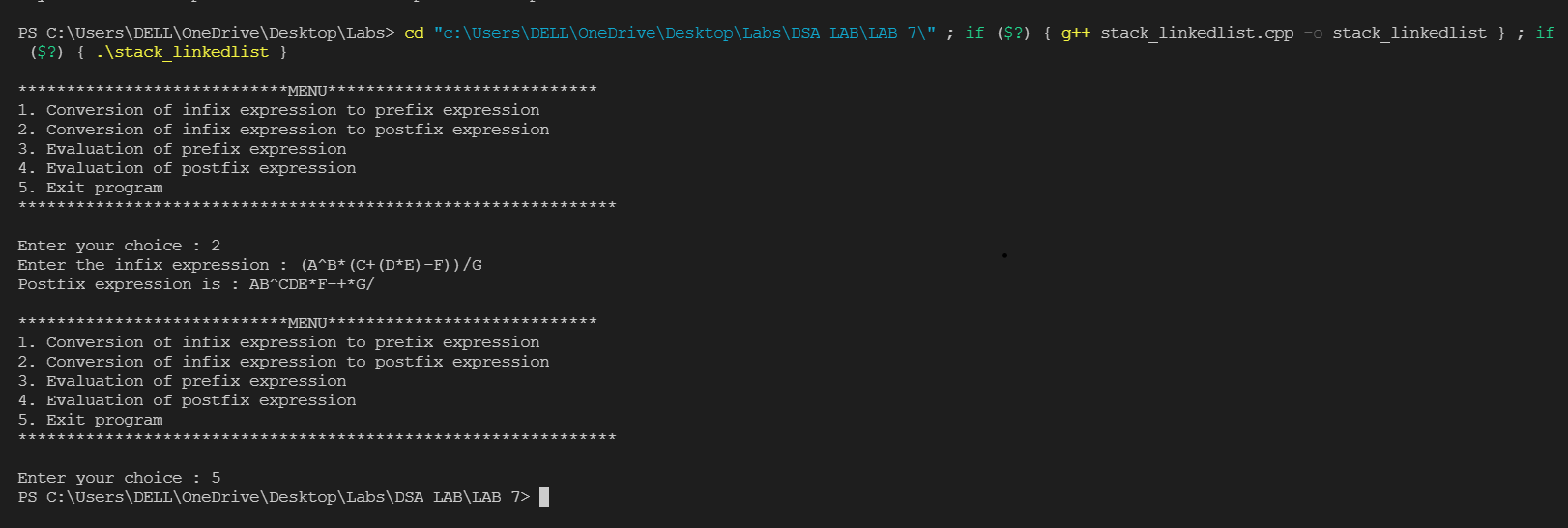
}

return 0;

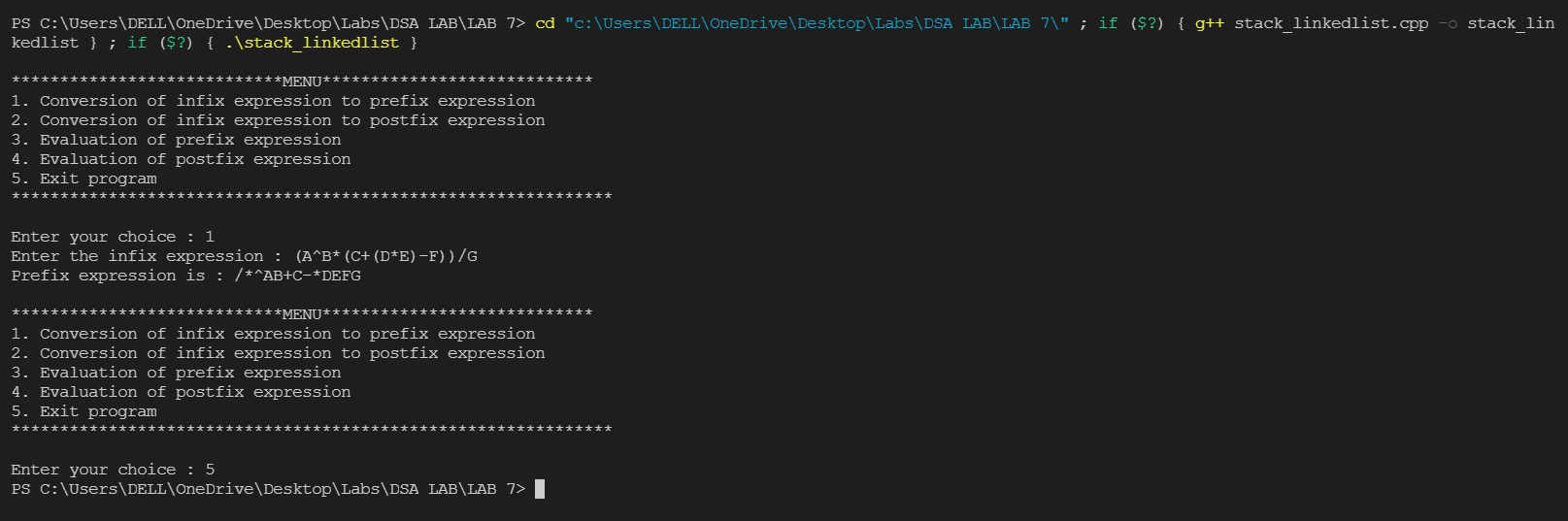
}

**Input & Output:**

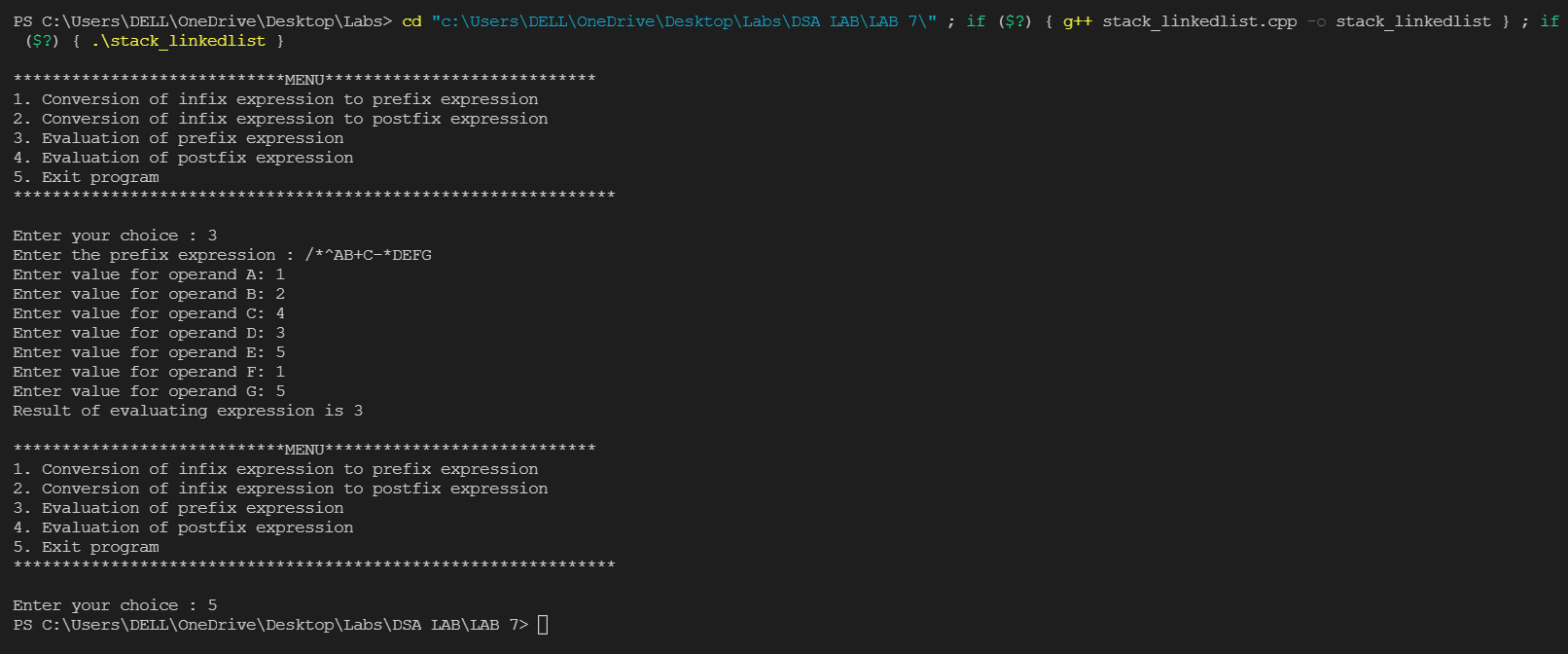
INFIX TO POSTFIX

****

INFIX TO PREFIX



EVALUATION OF PREFIX EXPRESSION



EVALUATION OF POSTFIX EXPRESSION

