Evaluation of arithmetic expressions

An expression consists of two components namely operands and operators.

Let’s convert a little complex expression with parentheses. Below is the given infix expression,

**( ( A + B ) — C \* ( D / E ) ) + F**

The given expression has parentheses to denote the precedence. So let’s start with the conversion with two empty elements respectively,

1. An empty expression string
2. An empty operator stack

The first token to encounter is an open parenthesis, add it to the operator stack.

1. Expression string:
2. Operator Stack: **(**
3. Remaining expression: **( A + B ) - C \* ( D / E ) ) + F**

The second token to encounter is again an open parenthesis, add it to the stack.

1. Expression string:
2. Operator Stack: **( (**
3. Remaining expression: **A + B ) - C \* ( D / E ) ) + F**

Next token un the expression is an operand “A”, so add it to the expression string.

1. Expression string: **A**
2. Operator Stack: **( (**
3. Remaining expression: **+ B ) - C \* ( D / E ) ) + F**

Afterward, we have an operator “+”, so add it to the stack.

1. Expression string: **A**
2. Operator Stack: **( ( +**
3. Remaining expression: **B ) - C \* ( D / E ) ) + F**

Then we have an operand, so add it to the expression string.

1. Expression string: **A B**
2. Operator Stack: **( ( +**
3. Remaining expression: **) - C \* ( D / E ) ) + F**

Next token in the given infix expression is a close parenthesis, as we encountered a close parenthesis we should pop the expressions from the stack and add it to the expression string until an open parenthesis popped from the stack.

1. Expression string: **A B +**
2. Operator Stack: **(**
3. Remaining expression: **- C \* ( D / E ) ) + F**

Notice here we didn’t push the close parenthesis to the stack, instead, we pooped out the operator “+” and added it to the expression string and pooped out one open parenthesis from the stack as well.

Next, we are encountering with an operator “-”, so push it to the stack.

1. Expression string: **A B +**
2. Operator Stack: **( -**
3. Remaining expression: **C \* ( D / E ) ) + F**

Next is an operand “C”, so add it to the expression string,

1. Expression string: **A B + C**
2. Operator Stack: **( -**
3. Remaining expression: **\* ( D / E ) ) + F**

Next is an operator “\*”, so push it to the stack.

1. Expression string: **A B + C**
2. Operator Stack: **( - \***
3. Remaining expression: **( D / E ) ) + F**

Next is an open parenthesis, so add it to the stack.

1. Expression string: **A B + C**
2. Operator Stack: **( - \* (**
3. Remaining expression: **D / E ) ) + F**

Next is an operand “D”, so add it to the expression string.

1. Expression string: **A B + C D**
2. Operator Stack: **( - \* (**
3. Remaining expression: **/ E ) ) + F**

Next we encounter an operator “/”, so push it to the stack.

1. Expression string: **A B + C D**
2. Operator Stack: **( - \* ( /**
3. Remaining expression: **E ) ) + F**

Then an oprand “E”, add it to the expression string.

1. Expression string: **A B + C D E**
2. Operator Stack: **( - \* ( /**
3. Remaining expression: **) ) + F**

Then a close parenthesis, as we saw earlier, we should not push it to the stack instead we should pop all the operators from the stack and add it to the expression string until we encounter an open parenthesis. Then pop the open parenthesis from the stack but don’t add it to the expression string.

1. Expression string: **A B + C D E /**
2. Operator Stack: **( - \***
3. Remaining expression: **) + F**

Next token is again a close paranthesis, so we will pop all the operators and add them to the expression string until we reach the open parenthesis and we will pop the open parenthesis as well from the operator stack.

1. Expression string: **A B + C D E / \* -**
2. Operator Stack:
3. Remaining expression: **+ F**

Next token is an operator “+”, so push it to the stack.

1. Expression string: **A B + C D E / \* -**
2. Operator Stack: **+**
3. Remaining expression: **F**

Next token is an operand, “F”. Add it to the expression string.

1. Expression string: **A B + C D E / \* - F**
2. Operator Stack: **+**
3. Remaining expression:

As we processed the whole infix expression, now the operator stack has to be cleared by popping out each remaining operator and adding them to the expression string.

Here we have the operator “+” on the stack, so we will pop out the operator “+” from the stack and will add it to the expression string. So the resultant Postfix expression would look like below,

Final Postfix expression: **A B + C D E / \* - F +**

**Algorithm**  
**1.** Scan the infix expression from left to right.  
**2.** If the scanned character is an operand, output it.  
**3.** Else,  
…..**3.1** If the precedence of the scanned operator is greater than the precedence of the operator in the stack(or the stack is empty or the stack contains a ‘(‘ ), push it.  
…..**3.2** Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator. After doing that Push the scanned operator to the stack. (If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)  
**4.** If the scanned character is an ‘(‘, push it to the stack.  
**5.** If the scanned character is an ‘)’, pop the stack and and output it until a ‘(‘ is encountered, and discard both the parenthesis.  
**6.** Repeat steps 2-6 until infix expression is scanned.  
**7.** Print the output  
**8.** Pop and output from the stack until it is not empty.

|  |
| --- |
| We use the following straight-forward algorithm to convert infix expression to a postfix expression :- 1) Scan the given expression from left to right. 2) First operator seen is simply pushed onto stack. 3) If we see an operand, append it to the **postfix** expression. 4) If we see an operator **(x)**, pop off all the operators which are of lower precedence than **‘x’** and append them to the **postfix** expression. Then, push the operator **‘x’** onto stack. 5) If we see an opening parenthesis, simply push it onto stack. 6) If we see a closing parenthesis, pop off all elements from stack till opening parenthesis and append them to **postfix** expression except the opening & closing parenthesis. 7) Finally, pop off all the elements (operators) from stack till it’s empty and append them to **postfix** expression. |

/\* C++ implementation to convert infix expression to postfix\*/

#include <iostream>

#include <stack>

using namespace std;

//function to return precedence of operators

int precdence(char c)

{

int weight;

if(c=='^')

weight=3;

else if(c=='\*' || c=='/')

weight=2;

else if(c=='+' || c=='-')

weight=1;

else

weight=-1;

return weight;

}

//function to convert infix to postfix

void infixTopostfix(string s)

{

stack <char> st;

st.push('N');

int stlen=s.length();

string ps="";

for(int i=0;i<stlen;i++)

{

///if the scanned character is an operand, add it to output string

if((s[i]>='a' && s[i]<='z')||(s[i]>='A' && s[i]<='Z'))

{

ps=ps+s[i];

}

//if the scanned character is an '(', push it to the stack

else if(s[i]=='(')

st.push('(');

// if the scanned character is an ')', pop and add to ps string

// until an '(' bracket is encountered.

else if(s[i]==')')

{

while(st.top()!='N' && st.top()!='(')

{

char c=st.top();

ps=ps+c;

st.pop();

}

// if it is an '(' simply pop it

if(st.top()=='(')

{

st.pop();

}

}

//if an operator is scanned

else

{

while(st.top()!='N' && precdence(s[i])<=precdence(st.top()))

{

char c=st.top();

st.pop();

ps=ps+c;

}

st.push(s[i]);

} //else

} //for

//pop all the remaining elements from the stack

while(st.top()!='N')

{

char c=st.top();

st.pop();

ps=ps+c;

}

cout<<ps<<endl;

}

int main()

{

string exp="(m-n)/p\*q+r^s\*t";

infixTopostfix(exp);

return 0;

}

**Postfix expression Evaluation**

**Algorithm:**

* Scan input expression from left to right
  + If scanned input is an operand, push it into the stack
  + If scanned input is an operator, pop out two values from stack. Then, perform operation between popped values and then push back the result into the stack.
* Repeat above two steps till all the characters are scanned.
* After all characters are scanned, there will be only one element in the stack, and this is the result of given expression.

//postfix evaluation

#include <iostream>

#include <stack>

#include <string.h>

#include <ctype.h>

#include <math.h>

using namespace std;

bool isOperator(char ch)

{

if(ch=='+' || ch=='-' || ch=='\*'|| ch=='/'||ch=='^')

return true;

else

return false;

}

int postfixcalc(int op1,int op2, char op)

{

int ans;

switch(op)

{

case '+':

ans=op2+op1;

break;

case '-':

ans=op2-op1;

break;

case '\*':

ans=op2\*op1;

break;

case '/':

ans=op2/op1;

break;

case '^':

ans=pow(op2,op1);

break;

}

return ans;

}

int main()

{

stack<int> st;

int op1,op2;

// int x,x1;

int result,len;

int buffer[40];

char exp[40];

cout<<"\nenter the postfix expression"<<endl;

gets(exp);

len=strlen(exp);

cout<<"the length of the string is : "<<len<<endl;

for(int i=0;i<len;i++)

{

if(isdigit(exp[i]))

{

buffer[i]=exp[i]-'0'; //to convert char to integer

st.push(buffer[i]);

}

else if(isOperator(exp[i]))

{

op1=st.top();

st.pop();

op2=st.top();

st.pop();

result=postfixcalc(op1,op2,exp[i]);

st.push(result);

}

}

cout<<"the answer is : "<<st.top();

return 0;

}