```
/*WAP to convert an infix expression into a prefix expression*/
#include<iostream>
#include<stack>
#include<locale> //for function isalnum()
#include<algorithm>
using namespace std;
int preced(char ch)
{
   if (ch == '+' || ch == '-')
   {
      return 1;    //Precedence of + or - is 1
   }
   else if (ch == '*' || ch == '/')
   {
      return 2;    //Precedence of * or / is 2
   }
   else if (ch == '$')
   {
      return 3;    //Precedence of $ is 3
   }
   else
   {
      return 0;
   }
}
string inToPost(string infix)
{
   stack<char> stk;
   stk.push('#');    //add some extra character to avoid underflow
   string postfix = "";   //initially the postfix string is empty
   string::iterator it;
   for (it = infix.begin(); it != infix.end(); it++)
   {
      if (isalnum(char(*it)))
         postfix += *it;    //add to postfix when character is letter or number
      else if (*it == '(')
         stk.push('(');
```

```
      else if (*it == '$')
         stk.push('$');
      else if (*it == ')')
      {
         while (stk.top() != '#' && stk.top() != '(')
         {
            postfix += stk.top();    //store and pop until ( has found
            stk.pop();
         }
         stk.pop();    //remove the '(' from stack
      }
      else
      {
         if (preced(*it) > preced(stk.top()))
            stk.push(*it);    //push if precedence is high
         else
         {
            while (stk.top() != '#' && preced(*it) <= preced(stk.top()))
            {
               postfix += stk.top();    //store and pop until higher precedence is
found
               stk.pop();
            }
            stk.push(*it);
         }
      }
   }
   while (stk.top() != '#')
   {
      postfix += stk.top();    //store and pop until stack is not empty
      stk.pop();
   }
   return postfix;
}
string inToPre(string infix)
{
   string prefix;
```

```
   reverse(infix.begin(), infix.end());    //reverse the infix expression
   string::iterator it;
   for (it = infix.begin(); it != infix.end(); it++)     //reverse the parenthesis after
reverse
   {
      if (*it == '(')
         *it = ')';
      else if (*it == ')')
         *it = '(';
   }
   prefix = inToPost(infix);              //convert new reversed infix to postfix form.
   reverse(prefix.begin(), prefix.end());    //again reverse the result to get final
prefix form
   return prefix;
}
int main()
{
   string infix;
   cout << "Use '+'  ,  '-'  , '*', '/' and '$' (for exponentiation)." << endl;
   cout << "Enter Infix Expression." << endl;
   cin >> infix;
   cout << "Infix expression is: " << endl << infix << endl;
   cout << "Prefix expression is: " << endl << inToPre(infix) << endl;
   return 0;
}

/*WAP to convert an infix expression into a prefix expression*/
#include<iostream>
#include<string>
#define max 15
using namespace std;
template<class T>
class Stack
{
   T data[max];
   int top;
public:
```

```
Stack():top(-1) {}
void push(T value)
{
   if(top==max-1)
   {
      cout<<"overflow"<<endl;
   }
   else
      data[++top]=value;
}
T pop()
{
   if(top==-1)
   {
      cout<<"underflow"<<endl;
   }
   else
   {
      return data[top--];
   }
}
T peek()
{
   if(top==-1)
   {
      cout<<"underflow"<<endl;
   }
   else
   {
      return data[top];
   }
}
void display()
{
   cout<<"------------------XX--------------"<<endl;
   for(int i=top; i>-1; i--)
   {
```

```
        cout<<data[i]<<endl;
      }
      cout<<"------------------XX--------------"<<endl;
   }

};
//precision check
int precision_check(char x)
{
   if(x=='$')
   {
      return 3;
   }
   else if(x=='*' || x=='/')
   {
      return 2;
   }
   else if(x=='+' || x=='-')
   {
      return 1;
   }
   else
   {
      return NULL;
   }
}
//infix expression to postfix expression
string infix_to_Allupostfix(string expression)
{
   Stack<char>converter;
   string postfix;
   char y;
   converter.push('(');
   for(auto x:expression)
   {
      if(x =='(')
      {
```

```
            converter.push(x);
        } // if left bracket is encountered
        else if(x == ')')
        {
            while(converter.peek() != '(')
            {
                y=converter.pop();

                postfix+=y;
            }
            converter.pop();
        }
        else if(x =='*' || x =='+' || x =='-' || x =='$' || x=='/')  //if operator is
encounter
        {
            if(converter.peek() =='(' )
            {
                converter.push(x);
            } // if left bracket is at top
            else if(precision_check(x)>=precision_check(converter.peek()))
            {
                converter.push(x);
            } // if operator is at top
            else
            {
                y=converter.pop();
                postfix+=y;
                converter.push(x);
            }
        }
        else  //if operand or character is encountered
        {
            postfix+=x;
        }
    }
    return postfix;
}
```

```
//driver function
int main()
{
   string expression;
   string rev_expression;
   string prefixexp;
   Stack<char>inverse;              //for invering sting
   cout<<"Enter your expression "<<endl;
   getline(cin,expression);
   for(auto x:expression)
   {
      inverse.push(x);
   }
   inverse.display();
   for(int i=0; i<expression.length(); i++)     //for inversing the given expression
   {
      if(inverse.peek()==')')
      {
         inverse.pop();
         rev_expression+='(';
      }
      else if(inverse.peek()=='(')
      {
         inverse.pop();
         rev_expression+=')';
      }
      else
      {
         rev_expression+=inverse.pop();
      }
   }
   //inverse.display();
   rev_expression+=')';
   //cout<<rev_expression<<endl;
   string x=infix_to_Allupostfix(rev_expression);
   cout<<"Before inverse: "<<x<<endl;
   for(auto i:x)            //for inversing the postfix to prefix
```

```
    {
        inverse.push(i);
    }
    for(int i=0; i<x.length(); i++)
    {
        prefixexp+=inverse.pop();
    }
    cout<<"prefix expression"<<prefixexp<<endl;
    return 0;
}
```