*//Q1 code to print 2 to the power n combinations*

*//for given n value*

#include <iostream>

#include <cmath>

using namespace std;

int main()

{

    int n;

    cout<<"Please give a value to print 2^n propositional variables:";

    cin>>n;

*// number of combinations for given n value*

    int NumOfCom;

    NumOfCom = pow(2, n);

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

    {

        cout<<i<<"\t";

        for (int j = n - 1; j >= 0; j--)

        {

            cout << ((i >> j) & 1)<<" ";

        }

        cout << endl;

    }

}

*/\**

*Q2 implement the evaluation of the given formula*

*using the values of the propositional variables*

*\*/*

#include <iostream>

#include <cmath>

#include <string>

using namespace std;

*// function to evaluate the given well-formed formula*

bool eva\_formula(string formula, int n, int values[])

{

}

int main()

{

    string formula;

    cout<<"Give your WFF :";*//WFF=well formed formula*

    cin >> formula;

    int n;

    cout<<"Please give a number :";*//for number of propositional variables*

    cin >> n;

    int Number\_Of\_Combinations = pow(2, n);

    for (int i = 0; i < Number\_Of\_Combinations; i++)

    {

        int values[n]; *// values of the propositional variables*

        for (int j = 0; j < n; j++)

        {

*// get the j-th bit of i*

            values[j] = (i >> j) & 1;

        }

*//We have to "Evaluate the given formula using the values of the propositional variables"*

        if (eva\_formula(formula, n, values))

        {

*// print the values of the propositional variables that make the formula true*

            for (int j = 0; j < n; j++)

                cout << values[j] << " ";

            cout << endl;

        }

    }

    return 0;

}

*//Q5 AND operator*

#include <iostream>

#include <string>

using namespace std;

#define AND &&

#define OR ||

#define NOT !

bool P = true;

bool Q = true;

int main() {

  cout << "P\tQ\tPandQ" << endl;

  cout << P << "\t" << Q << "\t" << (P AND Q) << endl;

  P = true; Q = false;

  cout << P << "\t" << Q << "\t" << (P AND Q) << endl;

  P = false; Q = true;

  cout << P << "\t" << Q << "\t" << (P AND Q) << endl;

  P = false; Q = false;

  cout << P << "\t" << Q << "\t" << (P AND Q) << endl;

  return 0;

}

//Q3 POSTFIX and PREFIX

#include <iostream>

#include <stack>

#include <string>

using namespace std;

string postfixToPrefix(string exp)

{

stack<string> stack;

for (int i = 0; i < exp.length(); i++)

{

// If the scanned character is an operand,

// push it to the stack.

if (isdigit(exp[i]))

stack.push(string(1, exp[i]));

// If the scanned character is an operator,

// pop two elements from the stack, perform

// the operation and push the result back to

// the stack.

else

{

string val1 = stack.top();

stack.pop();

string val2 = stack.top();

stack.pop();

// The expression is in postfix notation,

// so the operator appears after the

// operands. We need to reverse the order

// to obtain the prefix notation.

string temp = val1 + val2 + exp[i];

// Push the result back to the stack.

stack.push(temp);

}

}

// The final value in the stack is the result

// of the expression in prefix notation.

return stack.top();

}

string prefixToPostfix(string exp)

{

stack<string> stack;

for (int i = exp.length() - 1; i >= 0; i--)

{

// If the scanned character is an operand,

// push it to the stack.

if (isdigit(exp[i]))

stack.push(string(1, exp[i]));

// If the scanned character is an operator,

// pop two elements from the stack, perform

// the operation and push the result back to

// the stack.

else

{

string val1 = stack.top();

stack.pop();

string val2 = stack.top();

stack.pop();

// The expression is in prefix notation,

// so the operator appears before the

// operands. We need to reverse the order

// to obtain the postfix notation.

string temp = val1 + val2 + exp[i];

// Push the result back to the stack.

stack.push(temp);

}

}

return stack.top();

}

int main()

{

string exp1,exp2 ;

cout<<"enter postfix and prefix expressions repectively:\n";

cin>>exp1>>exp2;

cout << postfixToPrefix(exp1) << endl;

cout << prefixToPostfix(exp2) << endl;

return 0;

}

//Q4 Evaluate postfix expression and return the result

#include <iostream>

#include <stack>

#include <string>

using namespace std;

// Function to evaluate postfix expression and return the result

double evaluatePostfix(string exp)

{

// Create a stack

stack<double> stack;

// Scan all characters one by one

for (int i = 0; i < exp.length(); i++)

{

// If the scanned character is an operand (number here),

// push it to the stack

if (isdigit(exp[i]))

stack.push(exp[i] - '0');

// If the scanned character is an operator, pop two

// elements from stack and apply the operator

else

{

double val1 = stack.top();

stack.pop();

double val2 = stack.top();

stack.pop();

switch (exp[i])

{

case '+':

stack.push(val2 + val1);

break;

case '-':

stack.push(val2 - val1);

break;

case '/':

stack.push(val2 / val1);

break;

case '\*':

stack.push(val2 \* val1);

break;

}

}

}

// Return the result from the stack

return stack.top();

}

int main()

{

string exp ;

cout<<"enter a postfix expression:";

cin>>exp;

cout << "Result: " << evaluatePostfix(exp) << endl;

return 0;

}

//Q6 CNF formula to DNF

#include <iostream>

#include <string>

#include <vector>

#include <algorithm>

using namespace std;

// function to convert CNF formula to DNF

string convertToDNF(string cnf)

{

// split the string into clauses separated by '&'

vector<string> clauses;

string clause = "";

for (char ch: cnf)

{

if (ch == '&')

{

clauses.push\_back(clause);

clause = "";

}

else

clause += ch;

}

clauses.push\_back(clause);

// split each clause into terms separated by '|'

vector<vector<string>> terms;

for (string clause: clauses)

{

vector<string> termList;

string term = "";

for (char ch: clause)

{

if (ch == '|')

{

termList.push\_back(term);

term = "";

}

else

term += ch;

}

termList.push\_back(term);

terms.push\_back(termList);

}

// convert each clause to DNF

vector<vector<string>> dnfTerms;

for (auto termList: terms)

{

vector<string> dnfTermList;

for (string term: termList)

{

string dnfTerm = "";

for (char ch: term)

{

// negate negated terms and remove negation from others

if (ch == '!')

dnfTerm += (dnfTerm.length() == 0) ? "" : "|";

else

dnfTerm += ch;

}

dnfTermList.push\_back(dnfTerm);

}

dnfTerms.push\_back(dnfTermList);

}

// combine all terms into a single DNF formula

string dnf = "";

for (auto termList: dnfTerms)

{

for (string term: termList)

dnf += term + "&";

dnf.pop\_back(); // remove last '&'

dnf += "|";

}

dnf.pop\_back(); // remove last '|'

return dnf;

}

int main()

{

string cnf = "!x1|!x2&!x3|x4&x5";

string dnf = convertToDNF(cnf);

cout << "CNF formula: " << cnf << endl;

cout << "DNF formula: " << dnf << endl;

return 0;

}