**Assignment 3**

**Task:**

Create a class for propositional symbols. For example: If you have a Symbol class for creating any propositional symbol, it should be written as: p = Symbol('P') Define unary and binary logic operations/logical connectives for handling simple propositional logic. (Define operations for not, or, and, conditional, and double conditional) The inputs to your operations are all the possible values (boolean) of the constituent symbols and the outputs are also boolean values. Include another operation for printing a truth table for an operation that should mimic a truth table in the propositional logic .Using the defined symbols and operations, find the truth values and print the truth tables of the following propositions: (write a function to evaluate complex propositions)

1. ~P->Q

2. ~P ∧ ~Q

3. ~P ∨ ~Q

4. ~P->~Q

5. ~P <-> ~Q

6. (P ∨ Q)∧(~P->Q)

7. ((P ∨ Q)->~R)

8. (((P ∨ Q)->~R) <-> ((~P∧~Q)->~R))

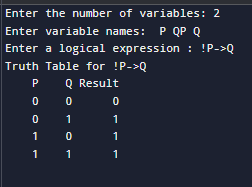
9. (((P->Q)∧(Q->R))->(Q->R))

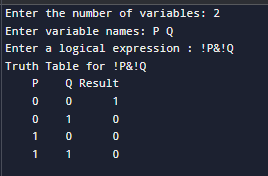
10. (((P->(Q∨R)) -> (~P∧~Q∧~R)))

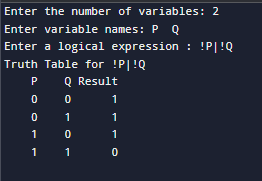
**Code:**

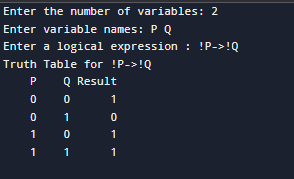
|  |
| --- |
| #include <bits/stdc++.h>  using namespace std;  class LogicalExpression {  public:      string expression;      vector<char> variables;      map<char, bool> values;      LogicalExpression() {          extractVariables();      }      void extractVariables() {          cout << "Enter the number of variables: ";          int num;          cin >> num;          cout << "Enter variable names: ";          for (int i = 0; i < num; i++) {              char var;              cin >> var;              variables.push\_back(var);          }      }      void setExpression(const string &expr) {          this->expression = expr;      }      int getPrecedence(const string &op) {          if (op == "!") return 4;          if (op == "&") return 3;          if (op == "|") return 2;          if (op == "->") return 1;          if (op == "<->" || op == "=") return 0;          return -1;      }      string convertToPostfix(const string &infix) {          stack<string> operators;          stringstream output;          string token;          for (size\_t i = 0; i < infix.size(); i++) {              if (infix[i] == ' ') continue;              if (infix[i] == '<' && i + 2 < infix.size() && infix[i + 1] == '-' && infix[i + 2] == '>') {                  token = "<->";                  i += 2;              } else if (infix[i] == '-' && i + 1 < infix.size() && infix[i + 1] == '>') {                  token = "->";                  i++;              } else if (isalnum(infix[i])) {                  token = string(1, infix[i]);              } else {                  token = string(1, infix[i]);              }              if (isalnum(token[0])) {                  output << token << " ";              } else if (token == "(") {                  operators.push(token);              } else if (token == ")") {                  while (!operators.empty() && operators.top() != "(") {                      output << operators.top() << " ";                      operators.pop();                  }                  if (!operators.empty()) operators.pop();              } else {                  while (!operators.empty() && getPrecedence(operators.top()) >= getPrecedence(token)) {                      output << operators.top() << " ";                      operators.pop();                  }                  operators.push(token);              }          }          while (!operators.empty()) {              output << operators.top() << " ";              operators.pop();          }          return output.str();      }      bool evaluateExpression(const string &postfix, const map<char, bool> &valMap) {          stack<bool> operands;          stringstream ss(postfix);          string token;          while (ss >> token) {              if (isalnum(token[0])) {                  operands.push(valMap.at(token[0]));              } else if (token == "!") {                  if (operands.empty()) {                      cerr << "Error: Missing operand for NOT operation!\n";                      return false;                  }                  bool val = operands.top();                  operands.pop();                  operands.push(!val);              } else {                  if (operands.size() < 2) {                      cerr << "Error: Insufficient operands for binary operation: " << token << "\n";                      return false;                  }                  bool val2 = operands.top(); operands.pop();                  bool val1 = operands.top(); operands.pop();                  if (token == "&") operands.push(val1 && val2);                  else if (token == "|") operands.push(val1 || val2);                  else if (token == "->") operands.push(!val1 || val2);                  else if (token == "<->" || token == "=") operands.push(val1 == val2);                  else {                      cerr << "Error: Unknown operator encountered: " << token << "\n";                      return false;                  }              }          }          if (operands.size() != 1) {              cerr << "Error: Malformed expression, operands left in stack!\n";              return false;          }          return operands.top();      }      void generateTruthTable() {          string postfix = convertToPostfix(expression);          int rows = pow(2, variables.size());          cout << "Truth Table for " << expression << "\n";          for (char var : variables) {              cout << setw(5) << var;          }          cout << " Result\n";            for (int i = 0; i < rows; i++) {              for (size\_t j = 0; j < variables.size(); j++) {                  values[variables[j]] = (i & (1 << (variables.size() - j - 1))) != 0;                  cout << setw(5) << (values[variables[j]] );              }              cout << setw(7) << (evaluateExpression(postfix, values) ) << "\n";          }      }  };  int main() {      LogicalExpression logicExp;      cout << "Enter a logical expression: ";      string inputExpr;      cin.ignore();      getline(cin, inputExpr);      logicExp.setExpression(inputExpr);      logicExp.generateTruthTable();      return 0;  } |

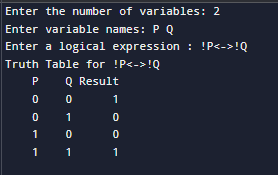
**Output:**

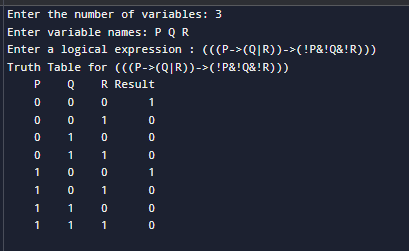
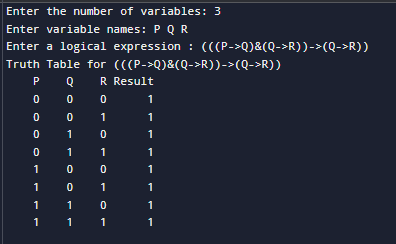
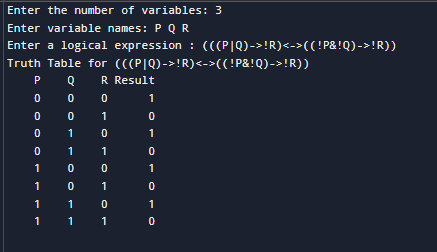
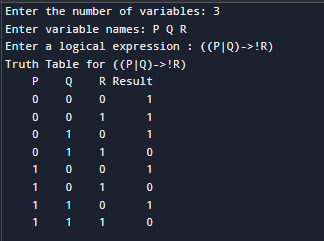
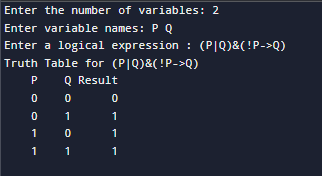
****

****

****

****

****

****