```
//*************************************     PROGRAM IDENTIFICATION   ***********************************
//*                                                                                                    *
//*     PROGRAM FILE NAME:   Program4.cpp           ASSIGNMENT  #:  4           Grade: _______       *
//*                                                                                                    *
//*     PROGRAM AUTHOR:       ____________________________________                                     *
//*                                    Adrian Beloqui                                                  *
//*                                                                                                    *
//*     COURSE #:   CSC  36000  11                                  DUE DATE:    Mar 10, 2017          *
//*                                                                                                    *
//******************************************************************************************************

//*************************************    PROGRAM DESCRIPTION   ***************************************
//*                                                                                                    *
//*      PROCESS: This program is designed to read a file with an unknown number of mathematical       *
//*               expressions in INFIX notation. Then it is to convert the expression to a POSTFIX     *
//*               notation showing each step of the process. Then it is to calculate the expression to *
//*               obtain the final result of the mathematical expression while showing each step of the*
//*               calculation. Then it is to print the initial expression with its final result.       *
//*                                                                                                    *
//*      USER DEFINED                                                                                  *
//*      MODULES      : main - Controlls the flow of the entire program, calling functions is the      *
//*                            right sequence and printing the labels into the output file.            *
//*                     Header - Prints a header in the output file.                                   *
//*                     Footer - Prints a footer in the output file.                                   *
//*                     PageBreak - Adds end lines to the output file.                                 *
//*                     StackCLASS::StackCLASS - Initializes the private members of the class.         *
//*                     StackCLASS::Push - Inserts a node into the stack.                              *
//*                     StackCLASS::Pop- Removes a nodes from the stack.                               *
//*                     StackCLASS::IsEmpty - Checks if the stack is empty.                            *
//*                     StackCLASS::PrintRightToLeft - Prints a stack with a right alignment           *
//*                     StackCLASS::PrintLeftToRight - Prints the stack reversed with a left alignment *
//*                     ReadNode - Reads a node and puts the values into its members                   *
//*                     ReadInputLine - Reads a line from the input file.                              *
//*                     ConvertExpToStack - Converts an expression to a stack of nodes                 *
//*                     ConvertInfixToPostfix - Converts an INFIX expression to a POSTFIX expression   *
//*                     getPrecedence - Calculates the predecedence of some operators.                 *
//*                     reverseStack - Reverses the nodes of a stack.                                  *
//*                     CalculatePostfixExp - Calculates a POSTFIX expression.                         *
//*                     ApplyOperation - Processes a mathematical operation accordingly to its parameters *
//*                     PrintResult - Prints the result of the calculation of a mathematical expression.  *
//*                                                                                                    *
//******************************************************************************************************

    //Imports
#include <string>
#include <fstream>
#include <iomanip>

    //Definition of constants
#define NOT !
#define LinesPerPage 67

    //Definition of namespace
using namespace std;

//Definition of a node structure
struct NodeTYPE{
    char element;
    int number;

    NodeTYPE *nextPtr;
};

    //Definition of classes
class StackCLASS{
public:
```

```
        // Constructor
    StackCLASS(){ TopPtr = NULL; }

        // Functions
    void Push(NodeTYPE);
    void Pop(ofstream &, NodeTYPE &);
    bool IsEmpty()  { return (TopPtr == NULL); }
    void PrintRightToLeft(ofstream &);
    void PrintLeftToRight(ofstream &, bool);
private:
    NodeTYPE *TopPtr;
};

    //Function prototypes definitions
void Header(ofstream &);
void Footer(ofstream &);
void PageBreak(ofstream &, int &);
void ReadNode(NodeTYPE *, char);
void ReadInputLine(ifstream &, char [], bool &);
void ConvertExpToStack(char [], StackCLASS &);
void ConvertInfixToPostfix(ofstream &, StackCLASS &, StackCLASS &, StackCLASS &);
int getPrecedence(char);
StackCLASS reverseStack(ofstream &, StackCLASS);
void CalculatePostfixExp(ofstream &, StackCLASS &, StackCLASS &);
int ApplyOperation(char, int, int);
void PrintResult(ofstream &, char [], NodeTYPE);

//************************************* FUNCTION MAIN  ***********************************************
int main()
{
    ifstream InFile;
    ofstream OutFile;

        //Set initial variables
    int linesWritten = 0;
    bool endOfFile = false;
    char expression[31];
    NodeTYPE newNode;
    StackCLASS tempStack;
    StackCLASS *infixStack = new StackCLASS();
    StackCLASS *postfixStack = new StackCLASS();
    StackCLASS *contentStack = new StackCLASS();

        // Open the input file
    InFile.open("stack_in.txt", ios::in);

        // Create the output file
    OutFile.open("output4.txt", ios::out);

        // Print the header in the output file.
    Header(OutFile);
        // Add amount of lines written into the output file
    linesWritten += 3;
        // Print separator line
    OutFile << "================================================="
        << "=========================================================" << endl << endl;
        // Add amount of lines written into the output file
    linesWritten += 2;

        // Read a line of input
    ReadInputLine(InFile, expression, endOfFile);

    do {
         // Convert the line read into an infix expression
        ConvertExpToStack(expression, *infixStack);
```

```
    // Print labels
OutFile << setw(55) << right << "CONVERSION DISPLAY" << endl;
OutFile << setw(25) << right << "Infix Expression" << setw(31) <<
    right << "Postfix Expression" << setw(25) << right << "Stack Contents" << endl;
OutFile << setw(81) << right << "(Top to Bottom)" << endl;

    // Print stacks before conversion from infix to postfix
infixStack->PrintRightToLeft(OutFile);
OutFile << setw(17) << left << ' ';
postfixStack->PrintLeftToRight(OutFile, false);
OutFile << setw(3) << left << ' ';
contentStack->PrintLeftToRight(OutFile, false);
OutFile << endl;
linesWritten += 4;

    // Check if the infix and content stacks are not empty
while ((NOT infixStack->IsEmpty()) || (NOT contentStack->IsEmpty())){
        // Process a step of the conversion from infix to postfix
    ConvertInfixToPostfix(OutFile, *infixStack, *postfixStack, *contentStack);
        // Print stacks
    infixStack->PrintRightToLeft(OutFile);
    OutFile << setw(17) << left << ' ';
    postfixStack->PrintLeftToRight(OutFile, false);
    OutFile << setw(3) << left << ' ';
    contentStack->PrintLeftToRight(OutFile, false);
    OutFile << endl;
    linesWritten += 1;
}

OutFile << endl;
linesWritten += 1;

    // Reverse the stack to process it from left to right
postfixStack = &(reverseStack(OutFile, *postfixStack));

    // Print labels
OutFile << setw(55) << right << "EVALUATION DISPLAY" << endl;
OutFile << setw(25) << right << "POSTFIX Expression" << setw(56)
    << right << "Stack Contents" << endl;
OutFile << setw(81) << right << "(Top to Bottom)" << endl;

    // Print the stacks before start the calculation of the postfix expression
postfixStack->PrintRightToLeft(OutFile);
OutFile << setw(45) << left << ' ';
contentStack->PrintLeftToRight(OutFile, true);
OutFile << endl;
linesWritten += 4;

    // Check if the postfix stack is empty
while (NOT postfixStack->IsEmpty()){
        // Process one step for the calculation of the expresion
    CalculatePostfixExp(OutFile, *postfixStack, *contentStack);
        // Print stacks
    postfixStack->PrintRightToLeft(OutFile);
    OutFile << setw(45) << left << ' ';
    contentStack->PrintLeftToRight(OutFile, true);
    OutFile << endl;
    linesWritten += 1;
}
    // Get last node of the content stack
contentStack->Pop(OutFile, newNode);

    // Print the stacks
postfixStack->PrintRightToLeft(OutFile);
OutFile << setw(45) << left << ' ';
contentStack->PrintLeftToRight(OutFile, true);
```

```
        OutFile << endl << endl;
        linesWritten += 2;

            // Print labels
        OutFile << left << "ORIGINAL EXPRESSION AND THE ANSWER:" << endl;
            // Print the result lable
        PrintResult(OutFile, expression, newNode);
        linesWritten += 2;

            // Print page break
        PageBreak(OutFile, linesWritten);

            // Clear the stacks creating new ones
        infixStack = new StackCLASS();
        postfixStack = new StackCLASS();
        contentStack = new StackCLASS();

            // Read a new line of input
        ReadInputLine(InFile, expression, endOfFile);

    } while (NOT endOfFile);

        // Print the footer into the output file.
    Footer(OutFile);

    return 0;
}
//***********************************  END OF FUNCTION MAIN  ****************************************

//************************************** FUNTION PAGEBREAK  ****************************************
void PageBreak(ofstream &Outfile, int &limit)
{
        // Receives - The output file and the amount of lines written in the current page.
        // Task - Add end lines to the output file.
        // Returns -  The output file and the amount of lines written in the current page.

        // Calculate amount of blank lines needed for new page
    limit = LinesPerPage - limit;

        // Print blank lines
    for (int i = 0; i < limit; i++){
        Outfile << endl;
    }
        // Reset amount of lines writen in one page
    limit = 0;
}
//***********************************  END OF FUNCTION PAGEBREAK  ***************************************

//************************************** FUNCTION PRINTLEFTTORIGHT ***********************************
void StackCLASS::PrintLeftToRight(ofstream &Outfile, bool isCalculation)
{
        // Receives - The output file, and a boolean
        // Task - Print the stack from top to bottom with a left alignment
        // Returns - The output file

    NodeTYPE *current;
    current = TopPtr;
    string expression;

        // Check if the stack is empty
    if (current == NULL) {
        Outfile << setw(25) << left << "EMPTY" << setfill(' ');
        return;
    }
        // Traverse the stack
    while (current != NULL)
```

```
    {
            // Check if the node contains an operator
        if (current->number == 0){
                // Convert operator to string and add it to the expression
            string str(1, current->element);
            expression.append(str);
        }
        else {
                // Check if the stack printed contains calculations
            if (isCalculation){
                    // Convert number to string
                string temp = to_string(current->number);
                    // Add the number in reverse order to the expression
                expression.append(string(temp.rbegin(), temp.rend()));
            }
            else {
                    // Convert the operand to a string and add it to the expression
                expression.append(to_string(current->number));
            }
        }

        current = current->nextPtr;
    }

        // Print the expression reversed
    Outfile << setw(25) << left << string(expression.rbegin(), expression.rend()) << setfill(' ');


    return;
}
//***********************************  END OF FUNCTION PRINTLEFTTORIGHT  ********************************

//*************************************** FUNCTION PRINTRIGHTTOLEFT ***********************************
void StackCLASS::PrintRightToLeft(ofstream &Outfile)
{
        // Receives - The output file
        // Task - Print a stack with a right alignment
        // Returns - The output file

    NodeTYPE *current;
    current = TopPtr;
    string expression;

        // Check if the stack is empty
    if (current == NULL) {
        expression.append("EMPTY");
    }
        // Traverse the stack
    while (current != NULL)
    {
            // Check if the node contains an operator
        if (current->number == 0){
                // Convert operator to string and add it to the expression
            string str(1, current->element);
            expression.append(str);
        }
        else {
                // Convert the operand to a string and add it to the expression
            expression.append(to_string(current->number));
        }

        current = current->nextPtr;
    }

        // Print expression with a right alignment
    Outfile << setw(25) << right << expression;
```

```
    return;
}
//***********************************  END OF FUNCTION PRINTRIGHTTOLEFT  ********************************

//************************************** FUNCTION POP ***************************************************
void StackCLASS::Pop(ofstream &outFile, NodeTYPE &node)
{
        // Receives - The output file, and a node
        // Task - Delete a node from a stack, and return it
        // Returns - The output file, and a node

    NodeTYPE *p;
        // Check if the stack is empty
    if (IsEmpty())
    {
        outFile << " Stack is empty. " << endl;
        outFile << " Delete Operation Failed. " << endl;
        return;
    }
        // Save element in the first node
    node.element = TopPtr->element;
    node.number = TopPtr->number;
        // Adjust Stack Top
    p = TopPtr;
    TopPtr = TopPtr->nextPtr;

    delete p;
    return;
}
//***********************************  END OF FUNCTION POP  *****************************************

//************************************** FUNCTION PUSH ***************************************************
void StackCLASS::Push(NodeTYPE node)
{
        // Receives - A node
        // Task - Insert a node into the stack
        // Returns - nothing

        // Create a new node
    NodeTYPE *p;
    p = new NodeTYPE;

        // Copy the data of the node to the new node
    p->element = node.element;
    p->number = node.number;
    p->nextPtr = TopPtr;
    TopPtr = p;

    return;
}
//***********************************  END OF FUNCTION PUSH  *****************************************

//************************************** FUNCTION PRINTRESULT ***************************************
void PrintResult(ofstream &Outfile, char expression[], NodeTYPE resultNode)
{
        // Receives - The output file, a char array and a node
        // Task - Print the expression and its result.
        // Returns - The output file

    string finalResult;

        // Traverse the character array
    for (int i = 0; i < strlen(expression); i++){
            // Convert character to string
        string str(1, expression[i]);
```

```
            // Add string to the final string
        finalResult.append(str);
        finalResult.append(" ");
    }

    finalResult.append(" =  ");
        // Add the result to the final string
    finalResult.append(to_string(resultNode.number));

        // Print the final string containing the expression and its result
    Outfile << setw(60) << right << finalResult;

    return;
}
//***********************************  END OF FUNCTION PRINTRESULT  ************************************

//*************************************** FUNTION CALCULATEPOSTFIXEXP  ********************************
void CalculatePostfixExp(ofstream &outFile, StackCLASS &postfixStack, StackCLASS &contentsStack)
{
        // Receives - The output file, the postfix stack and the content stack
        // Task - Perfrom one step of the calculation of a postfix expression
        // Returns - The output file, the postfix stack and the content stack

    NodeTYPE postfixNode, firstContentNode, secondContentNode, resultNode;

        // Get node from the postfix stack
    postfixStack.Pop(outFile, postfixNode);
        // Step 2
        // Check if the node is an operand
    if (postfixNode.number != 0){
            // Push operand to the content stack
        contentsStack.Push(postfixNode);
    }
    else {
            // Step 3
            // Get two operands from the content stack
        contentsStack.Pop(outFile, firstContentNode);
        contentsStack.Pop(outFile, secondContentNode);
            // Apply operation accordingly to the operand to the to operands
        resultNode.number = ApplyOperation(postfixNode.element, firstContentNode.number,
            secondContentNode.number);
            // Insert the result to the content stack
        contentsStack.Push(resultNode);
    }
    return;
}
//***********************************  END OF FUNCTION CALCULATEPOSTFIXEXP  ****************************

//************************************** FUNTION APPLYOPERATION  ***************************************
int ApplyOperation(char op, int firstOperand, int secondOperand)
{
        // Receives - A character, and two operands
        // Task - Apply an operation indicated by an operand to the two operands.
        // Returns - The result of the operation

    int result = 0;
        // Switch the operand and apply the corresponding operation to the operands.
    switch (op){
    case '+':
        result = secondOperand + firstOperand;
        break;
    case '-':
        result = secondOperand - firstOperand;
        break;
    case '*':
        result = secondOperand * firstOperand;
```

```
        break;
    case '/':
        result = secondOperand / firstOperand;
        break;
    }
    return result;
}
//***********************************  END OF FUNCTION APPLYOPERATION  ***********************************

//************************************** FUNTION CONVERTINFIXTOPOSTFIX  ******************************
void ConvertInfixToPostfix(ofstream &outFile, StackCLASS &infixStack, StackCLASS &postfixStack,
    StackCLASS &contentsStack)
{
        // Receives - The output file, and a stack for each expression (infix and postfix) and the
        //            contents stack
        // Task - Process one step of the conversion of a infix expression to a postfix expression
        // Returns - The output file, and a stack for each expression (infix and postfix) and the
        //           contents stack

    NodeTYPE infixNode, postfixNode, contentNode;

        // Get node from the infix expression
    infixStack.Pop(outFile, infixNode);

        // Step 2
        // If operand encountered, move it to the postfix expression
    if (infixNode.number != 0){
        postfixStack.Push(infixNode);
    }
    else {
            // Step 3
            // If a '(' is encountered, push it to the content stack
        if (infixNode.element == '('){
            contentsStack.Push(infixNode);
        }
        else {
                // Step 4
                // If a ')' is encountered, pop the contents stack
            if (infixNode.element == ')'){
                    // Pop first element
                contentsStack.Pop(outFile, contentNode);
                    // Check if the element is a matching '('
                while (contentNode.element != '('){
                        // Push the operand to the postfix expression
                    postfixStack.Push(contentNode);
                        // read next node
                    contentsStack.Pop(outFile, contentNode);
                }

            }
            else {
                    // Step 5.1
                    // Check if the content stack is empty
                if (contentsStack.IsEmpty()){
                        // Push node to the content stack
                    contentsStack.Push(infixNode);
                }
                else{
                        // Step 5.2
                        // Check if the content stack is empty
                    while (NOT contentsStack.IsEmpty()){
                            // Get node from the content stack
                        contentsStack.Pop(outFile, contentNode);
                            // Step 5.2.1
                            // Check if '(' is encountered or the the priority of the operator of the
                            // content stack is less than the priority of the main operator
```

```
                        if (contentNode.element == '(' ||
                            (getPrecedence(contentNode.element) < getPrecedence(infixNode.element))){
                                // Push the popped node to the content stack
                            contentsStack.Push(contentNode);
                                // Push the original node to the content stack
                            contentsStack.Push(infixNode);
                            return;
                        }
                            // Step 5.2.2
                            // Check if the content stack is empty
                        if (contentsStack.IsEmpty()){
                                // Push the popped node to the postfix stack
                            postfixStack.Push(contentNode);
                                // Push the original node to the content stack
                            contentsStack.Push(infixNode);
                            return;
                        }
                            // Step 5.2.3
                            // Check if the priority of the operator of the content stack is
                            // greater or equal to the main operator
                        if (getPrecedence(contentNode.element) >= getPrecedence(infixNode.element)){
                                // Push the popped node to the postfix stack
                            postfixStack.Push(contentNode);
                        }
                    }
                }
            }
        }
        // Step 6
        // Check if the infix stack is empty
    if (infixStack.IsEmpty()){
            // Check if the content stack is empty
        while (NOT contentsStack.IsEmpty()){
                // Pop a node from the content stack
            contentsStack.Pop(outFile, contentNode);
                // Push the node to the postfix stack
            postfixStack.Push(contentNode);
        }
    }
    return;
}
//***********************************  END OF FUNCTION CONVERTINFIXTOPOSTFIX  ****************************

//************************************** FUNTION REVERSESTACK  ***************************************
StackCLASS reverseStack(ofstream &outFile, StackCLASS stack)
{
        // Receives - The output file, and a stack
        // Task - Reverse the nodes of a stack and put them into a new stack
        // Returns - A new stack with the nodes reversed

        // Create local variables
    StackCLASS tempStack;
    NodeTYPE tempNode;

        // Check if the original stack is empty
    while (NOT stack.IsEmpty()){
            // Get the node from the original stack
        stack.Pop(outFile, tempNode);
            // Instert the node into the reversed stack
        tempStack.Push(tempNode);
    }

    return tempStack;
}
//***********************************  END OF FUNCTION REVERSESTACK  ************************************
```

```
//************************************** FUNTION GETPRECEDENCE  **************************************
int getPrecedence(char op)
{
        // Receives - A character
        // Task - Return the priority number of an operator
        // Returns - The priority in the precedence of an operator

        // Check if it is division or multiplication
    if (op == '*' || op == '/'){
        return 1;
    }
    return 0;
}
//**********************************  END OF FUNCTION GETPRECEDENCE  **********************************

//************************************** FUNTION CONVERTEXPTOSTACK  **********************************
void ConvertExpToStack(char value[], StackCLASS &stack)
{
        // Receives - A character array and a stack
        // Task - Insert a new node for each character in the character array
        // Returns - The stack

    NodeTYPE *newNode;
        // Traverse the character array
    for (int i = strlen(value) - 1; i >= 0; i--){
            // Create a new node
        newNode = new NodeTYPE();
            // Insert the character to the correct member of the node
        ReadNode(newNode, value[i]);
            // Push node to the stack
        stack.Push(*newNode);
    }
    return;
}
//**********************************  END OF FUNCTION CONVERTEXPTOSTACK  ******************************

//************************************** FUNTION READINPUTLINE  **************************************
void ReadInputLine(ifstream &Infile, char value[], bool &endOfFile)
{
        // Receives - The input file, a character array, and a bool indicating the end of the file
        // Task - Read a line of input and check if it is the end of the file
        // Returns - The input file, a character array, and a bool indicating the end of the file

    Infile >> ws;
        // Read input line
    Infile.getline(value, 31);
        // Check if it is the end of the file
    if (strcmp(value, "X") == 0){
        endOfFile = true;
    }
    return;
}
//**********************************  END OF FUNCTION READINPUTLINE  **********************************

//************************************** FUNTION READNODE  *******************************************
void ReadNode(NodeTYPE *newNode, char value)
{
        // Receives - A node, and a character
        // Task - Insert the character into the correct member of the node
        // Returns - The node

        // Insert the character in the correct member of a node
    switch (value){
    case '(': case ')': case '+': case '-': case '*': case '/':
        newNode->element = value;
```

```
        break;
    default:
            // Convert character into integer and inser it to the node
        newNode->number = value - '0';
        break;
    }
    return;
}
//***********************************  END OF FUNCTION READNODE  ************************************

//*************************************** FUNCTION HEADER  ***************************************
void Header(ofstream &Outfile)
{
        // Receives - The output file
        // Task - Prints the output preamble
        // Returns - The output file

    Outfile << setw(30) << "Adrian Beloqui ";
    Outfile << setw(17) << "CSC 36000";
    Outfile << setw(15) << "Section 11" << endl;
    Outfile << setw(30) << "Spring 2017";
    Outfile << setw(20) << "Assignment #4" << endl;
    Outfile << setw(35) << "-------------------------------------------------";
    Outfile << setw(35) << "-------------------------------------------------------" << endl;
    return;
}
//***********************************  END OF FUNCTION HEADER  ************************************

//*************************************** FUNCTION FOOTER  ***************************************
void Footer(ofstream &Outfile)
{
        // Receives - The output file
        // Task - Prints the output salutation
        // Returns - The output file

    Outfile << endl;
    Outfile << setw(35) << "-------------------------------------------------" << endl;
    Outfile << setw(35) << "|               END OF PROGRAM OUTPUT          |" << endl;
    Outfile << setw(35) << "-------------------------------------------------" << endl;
    return;
}
//***********************************  END OF FUNCTION FOOTER  ************************************
```