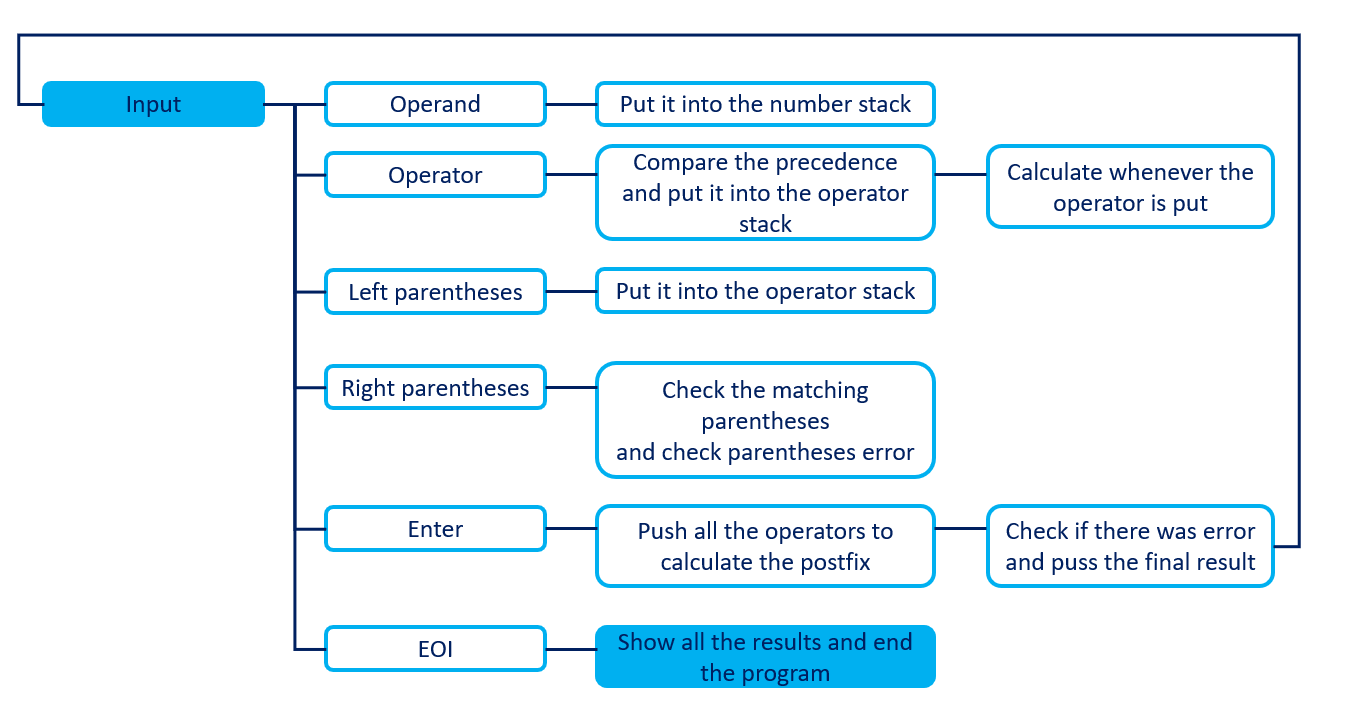
# HW2 Caculator

2015707056

Objective: Write a C++ program that inputs general real-numerical expressions line by line and then outputs their results. “EOI” means the end of input.

Compiler: Visual studio 2019

Algorithm schematic:



General idea of the program:

As you can see above algorithm schematic, the function will be differently executed by the input. Depends on input variable I categorized the input into 6 categories which is operand, operator, left parentheses, right parentheses, enter and lastly EOI. Depending on the types of input program will execute the different function. For instance, operand it will be stored in number stack. The operator will be stored in operator stack after comparing existing operator in the operator stack. If the new operator has higher or same precedence the operator will be put in postfix operator (queue) and call the calculating postfix function. Left parentheses goes to same stack as operator goes and when right parentheses are put program tries to search matching parentheses. The enter put and final end to the figures that has typed and push the final results to final\_result queue.

Own opinion:

At first, I tried to made a program which just calculate the figures as they put. But it has a huge hole while making the program. Accordingly, I made a infix to postfix program to calculate easier. Then I faced another problem. There is no way to store postfix expressions. The postfix expression has both double and char variables. When I try to pot char variables from double stack or queue the results automatically convert the char variable to double value. Therefore, I call the evaluate postfix function whenever the postfix expression has new operator. Then in the end, when user typed enter the top of the number stack becomes the final results as long as they don’t have any error of parentheses or divided by 0.

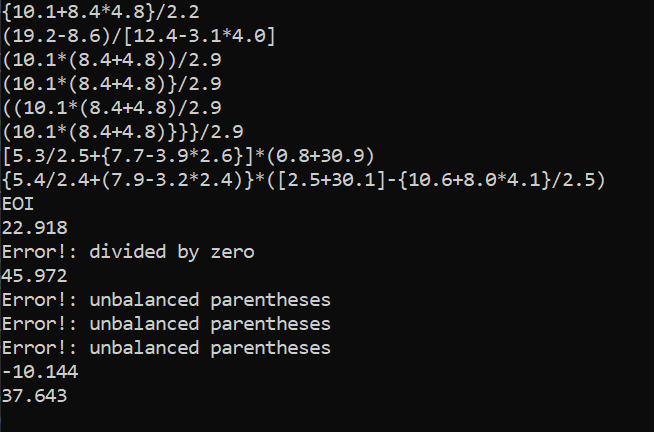
For error handling, I use error handling variable which is errorp and errorz. The last letter means the reason of the error. When the error handling variable is not 0. I put the order of the line into error index.

When the EOI is typed, the all the result should be shown. It should show the first answer first so I used queue to display all the result. As might be expected, when error index match with current final result index they will show the reason of the error.

While making this calculation program I learned how to deal with errors. There were so many abort errors. My first code usually tried to access even though the stack or queue is empty. Consequently my program terminated itself many times and have to discover which line makes the aborting error. Then I started check if the stack, queue is empty or not before accessing. This helps me much to correct memory error.

Example:

|  |  |
| --- | --- |
| Figure | Answer |
| {10.1+8.4\*4.8}/2.2 | 22.918 |
| (19.2-8.6)/[12.4-3.1\*4.0] | div by 0 |
| (10.1\*(8.4+4.8))/2.9 | 45.972 |
| (10.1\*(8.4+4.8)}/2.9 | mismatched parenthese |
| ((10.1\*(8.4+4.8)/2.9 | left unbalanced parenthese |
| (10.1\*(8.4+4.8)}}}/2.9 | right unbalanced parenthese |
| [5.3/2.5+{7.7-3.9\*2.6}]\*(0.8+30.9) | -10.144 |
| {5.4/2.4+(7.9-3.2\*2.4)}\*([2.5+30.1]-{10.6+8.0\*4.1}/2.5) | 37.643 |
| EOI |  |



Overall code:

//2015707056 조아라 ARA JO

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <cctype> // Provides isdigit

#include <cstdlib> // Provides EXIT\_SUCCESS

#include <cstring> // Provides strchr

#include <iostream> // Provides cout, cin, peek, ignore

#include <stack> // Provides the stack template class

#include <queue> // Provides the queue

#include <iomanip> // Provides the setprecision

using namespace std;

void read\_and\_to\_postfix(istream& ins);

void evaluate\_postfix(stack<double>& results, queue<char>& postfix\_operators, int& errorz);

bool HasHigherPrecedence(char op1, char op2);

bool IsLeftParentheses(char C);

bool IsRightParentheses(char C);

bool IsOperator(char C);

int GetOperatorWeight(char op);

void oppositeparentheses(char& c);

int main(void) {

read\_and\_to\_postfix(cin);

return EXIT\_SUCCESS;

}

void read\_and\_to\_postfix(istream& ins)

{

const char DECIMAL = '.';

stack<double> numbers;

stack<char> operators;

stack<double> results;

queue<char> postfix\_operators;

queue<double> final\_results;

queue<int> error\_index;

double number;

char symbol;

char s[100];

char init[4];

strcpy(init, "EOI");

int order\_of\_line = 1;

int num\_of\_parentheses = 0;

int errorz = 0;

int errorp = 0;

while (ins)

{

if (isdigit(ins.peek()) || (ins.peek() == DECIMAL)) {

//If the input stream is number or start with the DECIMAL read the input and put it in the stack named result

ins >> number;

results.push(number);

}

else if (IsOperator(ins.peek())) {

//If the input stream value is operator go through while loop

//until the operator stack is empty and the current operator has lower precedence

//If the operator has higher or same precedence put the operator to the stack named posfix\_operator and do the postfix calculation

ins >> symbol;

while (!operators.empty() && !IsLeftParentheses(operators.top()) && HasHigherPrecedence(operators.top(), symbol)) {

postfix\_operators.push(operators.top());

evaluate\_postfix(results, postfix\_operators, errorz);

operators.pop();

}operators.push(symbol);

}

else if (IsLeftParentheses(ins.peek())) {

//If the input stream is one of left parentheses push it in the operators

//and count the number of parentheses to check the total input parentheses is balanced

ins >> symbol;

operators.push(symbol);

num\_of\_parentheses++;

}

else if (IsRightParentheses(ins.peek())) {

//If the input stream is one of right parentheses first search for matching left parentheses.

//If there is matching left parentheses, pop it and break the loop.

//If there is no matching but still left parentheses it is parentheses error.

//Increase the errorp which counts the error of parenthese.

//While poping the operators untill find the lest parentheses, and the stack operator gets empty it is also parentheses error.

ins >> symbol;

num\_of\_parentheses--;

oppositeparentheses(symbol);

while (!operators.empty()) {

if (operators.top() == symbol) {

operators.pop();

break;

}

if (IsOperator(operators.top())) {

postfix\_operators.push(operators.top());

operators.pop();

evaluate\_postfix(results, postfix\_operators, errorz);

}

else if (IsLeftParentheses(operators.top())) {

operators.pop();

errorp++;

}

if (operators.empty()) {

errorp++;

break;

}

}

}

else if (ins.peek() == '\n') {

//If the enter is typed, ignore the current input and push all the oprators into postfix oprator stack.

//Then calculate the postfix expression

//The final result will be on the top of the stack result

//Check first if there were parentheses error or divided by 0 error by looking at errorp,errorz index.

//If there were errors puth the 0 or 1 to final results and push the error line into queue error index

ins.ignore();

while (!operators.empty() && IsOperator(operators.top())) {

postfix\_operators.push(operators.top());

operators.pop();

}

evaluate\_postfix(results, postfix\_operators, errorz);

if (num\_of\_parentheses != 0)

errorp++;

//This will discern if there is error or not.

if (errorz == 0 && errorp == 0) {

final\_results.push(results.top());

final\_results.front();

}

else if (errorz != 0) {

final\_results.push(0);

error\_index.push(order\_of\_line);

errorz = 0;

}

else if (errorp != 0) {

final\_results.push(1);

error\_index.push(order\_of\_line);

errorp = 0;

num\_of\_parentheses = 0;

}

//This clears the stack and queue that has used.

while (!results.empty()) {

results.pop();

}

while (!operators.empty()) {

operators.pop();

}

while (!postfix\_operators.empty()) {

postfix\_operators.pop();

}

order\_of\_line++;

}

else if (ins.peek() == 'E' || ins.peek() == 'O' || ins.peek() == 'I') {

//If "EOI" typed straight then show all the result in the queue final\_results

if (ins.peek() == 'E') //If 'E' has typed, clear the array to compare with new inputs

for (int i = 0; i < sizeof(s); i++)

s[i] = 0;

ins >> s;

if (strcmp(s, init) == 0) {

int i = 1;

while (!final\_results.empty()) {

if (!error\_index.empty()) {

if (i == error\_index.front() && final\_results.front() == 0) {

cout << "Error!: divided by zero" << endl;

error\_index.pop();

}

else if (i == error\_index.front() && final\_results.front() == 1) {

cout << "Error!: unbalanced parentheses" << endl;

error\_index.pop();

}

else

cout << fixed << setprecision(3) << final\_results.front() << endl;

final\_results.pop(); i++;

continue;

}

cout << fixed << setprecision(3) << final\_results.front() << endl;

final\_results.pop(); i++;

}

return;

}

}

else

ins.ignore();

}

}

//This function evaluates the postfix expression whenever the oprator is pushed

void evaluate\_postfix(stack<double>& results, queue<char>& postfix\_operators, int& errorz)

{

double operand1, operand2;

char symbol;

while (!postfix\_operators.empty() && !results.empty()) {

operand2 = results.top(); results.pop();

operand1 = results.top(); results.pop();

symbol = postfix\_operators.front(); postfix\_operators.pop();

if (symbol == '+')

results.push(operand1 + operand2);

else if (symbol == '-')

results.push(operand1 - operand2);

else if (symbol == '\*')

results.push(operand1 \* operand2);

else if (symbol == '/') {

if (operand2 == 0) {

errorz++;

results.push(0);

}

else

results.push(operand1 / operand2);

}

}

}

//The HasHigherPrecedence function return the oprator has higher precedence or not by comparing the weight of the two input

bool HasHigherPrecedence(char op1, char op2)

{

int op1Weight = GetOperatorWeight(op1);

int op2Weight = GetOperatorWeight(op2);

return (op1Weight >= op2Weight);

}

//The GetOperatorWeight function return the weight of the current operator.

int GetOperatorWeight(char op)

{

int weight = 0;

switch (op)

{

case '+':

case '-':

weight = 1;

break;

case '\*':

case '/':

weight = 2;

break;

}

return weight;

}

//The IsLeftParentheses function checks if the input is one of the left parentheses.

bool IsLeftParentheses(char C) {

return (C == '(' || C == '{' || C == '[');

}

//The IsRightParentheses function checks if the input is one of the right parentheses.

bool IsRightParentheses(char C) {

return (C == ')' || C == '}' || C == ']');

}

//The IsOperator function checks if the input is one of the operators(+-\*/)

bool IsOperator(char C) {

return (C == '+' || C == '-' || C == '\*' || C == '/');

}

//The oppositeparentheses function returns the opposite parentheses.

void oppositeparentheses(char& c) {

if (c == ')')

c = '(';

else if (c == ']')

c = '[';

else if (c == '}')

c = '{';

}