­­

|  |
| --- |
| laptop on office table |
|  |

Expression evaluator

By: Zahra Esmaeili

This document covers the code and algorithm used to evaluate an expression entered as a string of characters.

The code considers operator precedence and also assumes that the maximum number of entered characters is 50. The code was compiled and debugged using Microsoft Visual Studio 2019. Appendix A shows the results from test cases in the exercise and Appendix B includes the code.

Contents

[1-Code Architecture 3](#_Toc32665666)

[2-Parser 3](#_Toc32665667)

[2-1 Balanced parenthesis check 3](#_Toc32665668)

[2-2 Operand recognition 3](#_Toc32665669)

[2-3 “(“ recognition 4](#_Toc32665670)

[2-4 “)“ recognition 4](#_Toc32665671)

[2-5 Operations recognition 4](#_Toc32665672)

[2-6 Operation priorities 4](#_Toc32665673)

[2-7 Final calculations 4](#_Toc32665674)

[3-Calculator 5](#_Toc32665675)

[4-Errors 5](#_Toc32665676)

[Appendix A 6](#_Toc32665677)

[Appendix B 7](#_Toc32665678)

1-Code Architecture

I have used two stacks as the memory in this code, operator stack and operand stack. The code consists of two different parts:

* Parser: which is the major part gets the user string, distinguishes operands from operators, checks the syntax for unallowed characters and unbalanced parenthesis and creates integers from user strings and finally pushes and pops the appropriate data to the stack based on some specific roles.
* Calculator: which calculates the result of executing operations on the operand and puts the result in the corresponding integer variable.

2-Parser

The parser roles are as follows:

# 2-1 Balanced parenthesis check

First, the parser checks for the balance of parenthesis by a simple counter. This counter increments when parsing a “(“ and decrements when a “)” is parsed. If the final value of this counter is not 0 then an unbalanced nested expression has been occurred.

while (expression[i] != '\0')

{

if (expression[i] == '(')

nested\_exp++;

else if (expression[i] == ')')

nested\_exp--;

i++;

}

Snippet 2-1-1

Now we have the length of the string and all the following processes are done from the end of the string towards the start.

# 2-2 Operand recognition

As the input characters are ASCII and based on the problem’s assumption the numbers which are allowed to be entered by the user are from 0 to 9, their corresponding ASCII code falls between 48 to 57. So, to extract the original number from the ASCII code we should subtract 48 from provided ASCII code.

Next, an integer number should be formed from the string of numbers, i.e. “12345” should be converted to 12345. I wrote a simple piece of code for this purpose:

operand0 = pow(10, k) \* (expression[j] - 48) + operand0;

Snippet 2-2-1

This line basically subtracts 48 from the ASCII code and multiplies it by the appropriate power of 10 and finally adds the result to the previous result. This operation continues until reaching a character whose ASCII code doesn’t belong to this range, i.e. 48 to 57.

# 2-3 “(“ recognition

As we are moving back into the string, this character means that a nested expression is completed and should be evaluated. If the user enters a valid string which is the case because the code finds the unbalanced nested expressions and warns the user, a number is always available before this character. When this character is observed, the result of operand recognition, i.e. operand0, will be pushed to the operand stack and the value of operand0 and k in the snippet 2-2-1 will be changed to 0. Then, the innermost nested expression should be calculated. This is done by double popping the numbers in the operand stack and performing the top operation of the operation stack on them. The result will be pushed to the operand stack. This will be done until reaching a “)“ at the top of the operator stack.

# 2-4 “)“ recognition

When this character is observed, it will simply be pushed into the operation stack.

# 2-5 Operations recognition

We are moving backward in the string. So, when an operation is observed there could be two different situations:

1. A “(“ is located before the operation. In this case, operation will be pushed into the operator stack.
2. A number is located before the operation. The number will be pushed into operand stack while the operation is pushed into operator stack in this case.

# 2-6 Operation priorities

We know that the parenthesis has the highest priority in the operators followed by multiply/division and addition /subtraction respectively. The parenthesis priority was handled before. So, I added a flag to show if the previous operation is multiply/division or not. Moving back in the string, when encountering an operation, the program checks the priority flag of the previous operation. If the flag is true, the calculation for the previous operation will be performed first by popping the corresponding operands and operation from their stack and then the result will be pushed to the operand stack. Then, the new operation will be pushed into the operator stack.

# 2-7 Final calculations

When reaching the start index of the string (because we are moving backward from the last index to the first index), there would be some operations and operands remaining in their stacks. To calculate the final result, two operands are popped from the operand stack and the operation popped from the operator stack will be performed on them each time until the empty flag of the operator stack is set. There will be one operand still remaining in the operand stack which is the final result.

3-Calculator

The calculator is a function named as “calculate\_expression”. This function gets two operands and one operator and returns the result. This is basically done using a switch case on the operator. We know that the result of / in C is always an integer value. So, I didn’t consider the type of the result as double.

4-Errors

The evaluator returns the following errors:

1. Division by zero
2. Missing bracket
3. Unexpected character

Appendix A

The code was written and compiled in Microsoft Visual Studio 2019 and test cases in the exercise were used to validate the results with following answers.

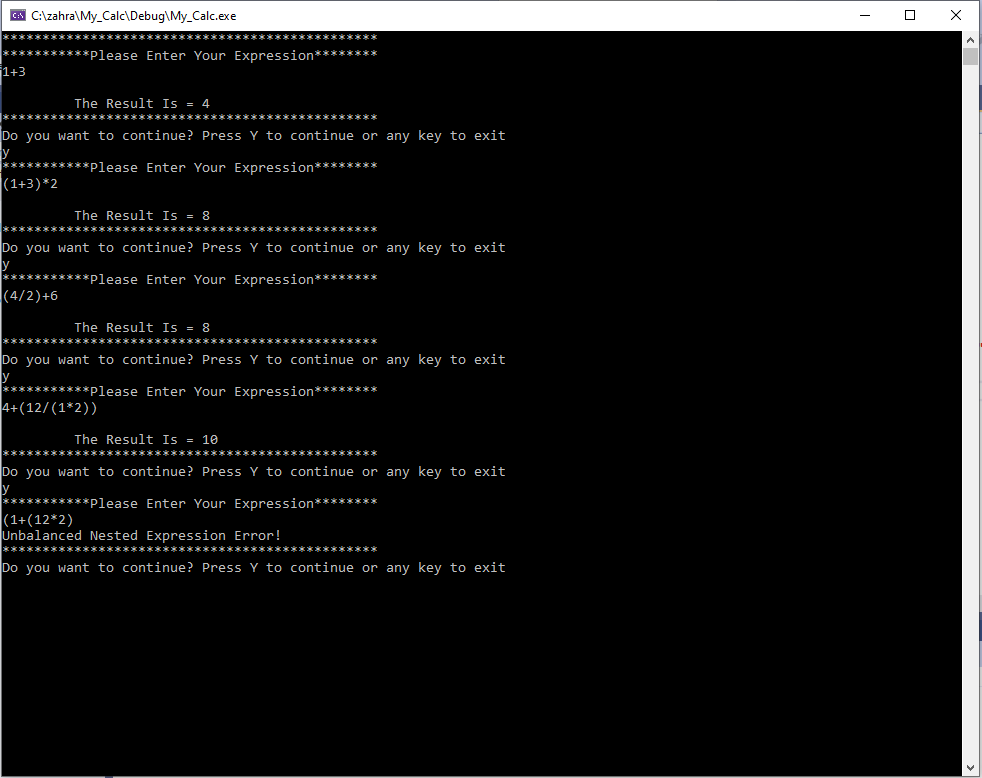


Figure A-1

Appendix B

#include <iostream>

#include <stack>

using namespace std;

bool evalute(const char\* expression, int& result);

bool calculate\_expression(int operand1, int operand2, char operator1, int& result);

int main()

{

char\* exp = new char[51];

char ch;

int result1;

bool repeat\_flag = true;

std::cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n";

while (repeat\_flag)

{

std::cout << "\*\*\*\*\*\*\*\*\*\*\*Please Enter Your Expression\*\*\*\*\*\*\*\*\n";

std::cin.get(exp, 50);

if (!evalute(exp, result1))

std::cout << "\n The Result Is = " << result1 << "\n";

std::cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n";

std::cout << "Do you want to continue? Press Y to continue or any key to exit\n";

std::cin >> ch;

if (ch != 'Y' && ch !='y')

repeat\_flag = false;

cin.clear();

std::cin.ignore(50, '\n');

}

delete[] exp;

return 0;

}

/ Parsing is the major part of the evalute function that distinguishes operands from operators

bool evalute(const char\* expression, int& result)

{

int i = 0;

int j = 0;

int k = 0;

int nested\_exp = 0;

int operand0 = 0;

int operand1 = 0;

int operand2 = 0;

char operator1;

bool priority\_flag = false;

int calc\_result = 0;

stack <int> stack\_operand;

stack <char> stack\_operator;

while (expression[i] != '\0')//This counter increments when parsing a “(“ and decrements when a “)” is parsed

{

if (expression[i] == '(')

nested\_exp++;

else if (expression[i] == ')')

nested\_exp--;

if(nested\_exp < 0) // Unusual usage of bracket or unbalanced number of bracket

{

std::cout << "Incorrect Usage of Bracket or Unbalanced Number of Bracket Error!\n";

return 1;

}

// checking the first character

if (expression[0] == '+' || expression[0] == '-' || expression[0] == '\*' || expression[0] == '/')

{

std::cout << "First Character Should Be A Digit Or '(' Error!\n";

return 1;

}

i++;

}

if (nested\_exp != 0)

{

std::cout << "Unbalanced Nested Expression Error!\n";

return 1;

}

j = i-1; // size of expression

while (j >= 0 )

{

if (expression[j] == ')')

{

stack\_operator.push(expression[j]);

priority\_flag = false;

j--;

}

else if ( expression[j] == 32) //Space detection

j--;

else if (expression[j] > 47 && expression[j] < 58) //Operand recognition

{

// the corresponding ASCII code falls between 48 to 57

operand0 = pow(10, k) \* (expression[j] - 48) + operand0;

k++; // k represents the number of digits in integer number

if (!j)

{

stack\_operand.push(operand0);

operand0 = 0;

k = 0;

}

j--;

}

//Operations recognition

else if (expression[j] == '+' || expression[j] == '-' || expression[j] == '\*' || expression[j] == '/')

{// check priority

if ( priority\_flag && (expression[j] == '+' || expression[j] == '-') )

{

if (k > 0) // means that we have a operand that has been not pushed yet

{

operand1 = operand0;

operand0 = 0;

k = 0;

}

//pop two operands and one operatore from stacks

operand2 = stack\_operand.top();

stack\_operand.pop();

operator1 = stack\_operator.top();

stack\_operator.pop();

if (calculate\_expression(operand1, operand2, operator1, calc\_result))

return 1;

stack\_operand.push(calc\_result);

priority\_flag = false;// priority operation has been done

}

else if (k > 0) // means that we have a operand that has been not pushed yet

{

stack\_operand.push(operand0);

operand0 = 0;

k = 0;

}

stack\_operator.push(expression[j]);

if (expression[j] == '\*' || expression[j] == '/')

priority\_flag = true;

j--;

}

else if (expression[j] == '(' )

{

if(!stack\_operator.empty())

{

if (k > 0)

{

stack\_operand.push(operand0);// means that we have a operand that has been not pushed yet

operand0 = 0;

k = 0;

}

operator1 = stack\_operator.top();

while (operator1 != ')')

{

operand1 = stack\_operand.top();

stack\_operand.pop();

operand2 = stack\_operand.top();

stack\_operand.pop();

operator1 = stack\_operator.top();

stack\_operator.pop();

if (calculate\_expression(operand1, operand2, operator1, calc\_result))

return 1;

stack\_operand.push(calc\_result);

operator1 = stack\_operator.top();

}

stack\_operator.pop();

priority\_flag = false;

}

j--;

}

else

{

std::cout << "UnExpected Character Error!\n";

return 1;

}

}// parssing is finished

while (!stack\_operator.empty()) //check final condition to reach the result

//just execute operation on remaing datas in two stack

{

//pop two operands and one operatore from stacks each time

operand1 = stack\_operand.top();

stack\_operand.pop();

operand2 = stack\_operand.top();

stack\_operand.pop();

operator1 = stack\_operator.top();

stack\_operator.pop();

if (calculate\_expression(operand1, operand2, operator1, calc\_result))

return 1;

stack\_operand.push(calc\_result);

}

result = stack\_operand.top();

stack\_operand.pop();

return 0;

}

/\*

This function gets two operands and one operator and returns the result

\*/

bool calculate\_expression(int operand1, int operand2, char operator1, int& result)

{

switch (operator1)

{

case '+':

result = operand1 + operand2;

break;

case '-':

result = operand1 - operand2;

break;

case '\*':

result = operand1 \* operand2;

break;

case '/':

if(operand2 != 0)

result = operand1 / operand2;

else

{

std::cout << "Division By Zero!\n";

return 1;

}

break;

default:

break;

}

return 0;

}