C++ Programming Work

Parse Arithmetic Expressions in the Standard BODMAS Format Program Code Description

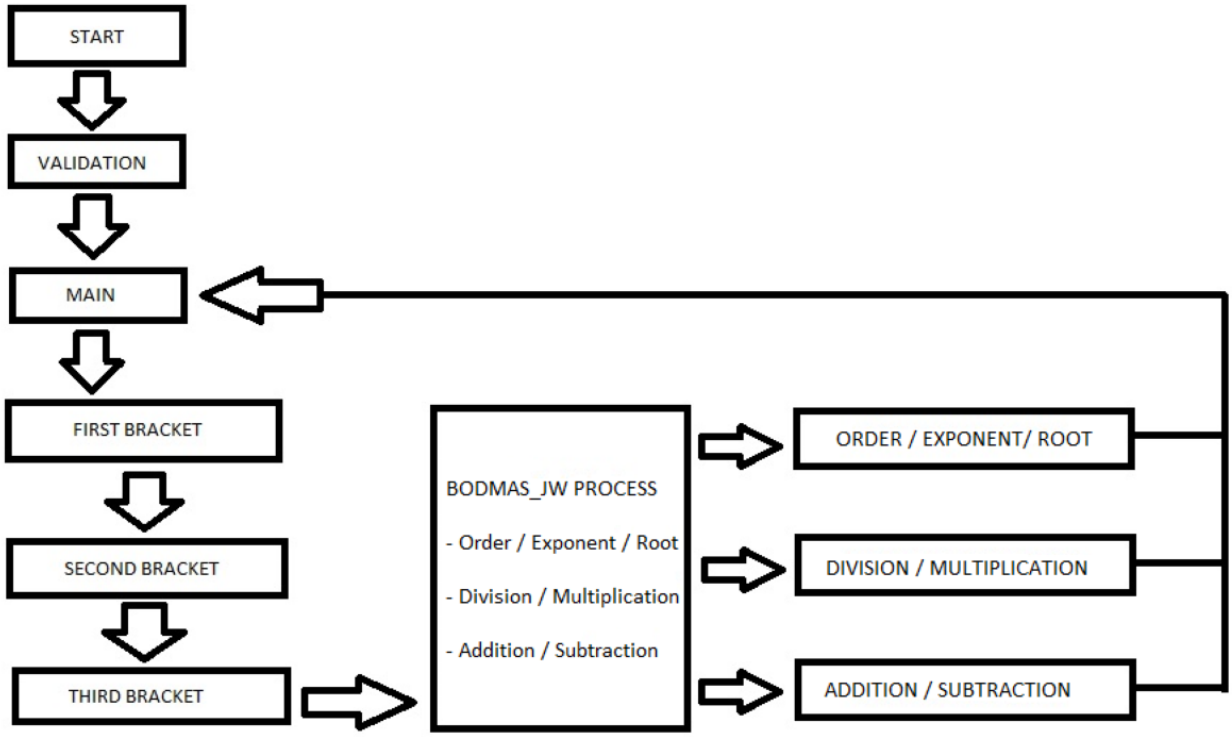
April 20, 2021

Jeremy J. Williams

1. Pseudocode

- Begin Program
    - To Start: Enter Your Expression
        - Use Numbers and/or + - * / ^ r . ( ) [ ] { }
    - To End: Enter 'Exit' To End Program
- Create and Open Final Results File
- Detect the invalid characters in the expression.
- Place multiplication (*) near the brackets left and right.
- Send the expression to detect the brackets and this time detect first brackets “()”
- If found, then pass the retrieve expression from first bracket to BODMAS_JW process.
    - If not found, then it sends to detect second bracket “{}”.
- If found, then pass the retrieve expression from first bracket to BODMAS_JW process.
    - If not found, then it sends to detect second bracket “[]”.
- If found, then pass the retrieve expression from first bracket to BODMAS_JW process.
    - If not found, then it passes to the BODMAS_JW process.
- BODMAS_JW (order/exponent/division/multiplication/addition/subtraction) sends the expression to the order/exponent/root process and if detects then calculates and return back expression to the BODMAS_JW.
    - BODMAS_JW sends it to the START position.
- BODMAS_JW sends it to the division/multiplication detection and if detects then calculates and return expression to the BODMAS_JW.
    - BODMAS_JW sends it to the START position.
- BODMAS_JW sends it to the addition/multiplication detection and if detects then calculates and return back expression to the BODMAS_JW.
    - BODMAS_JW sends it to the START position.
- Finish the expression calculation.
- Update, Saved and Closed Final Results File
- End Program

2. Workflow Block Diagram



3. Source Code

3.1 Global Declarations

The following code are global declarations of headers, variables, and functions marker.

```
#include <iostream>
#include <string>
#include <cmath>
#include <stdio.h>
#include <fstream>

bool bCancelFlag = false;

std::string validate_expression(std::string sEXP);
std::string insert_multiplication_near_bracket(std::string sEXP);
std::string calculate_operators(std::string sEXP);
std::string first_bracket(std::string sEXP);
std::string second_bracket(std::string sEXP);
std::string third_bracket(std::string sEXP);
std::string BODMAS_JW(std::string sEXP);
std::string order_exponent_root(std::string sEXP);
std::string division_multiplication(std::string sEXP);
std::string addition_subtraction(std::string sEXP);
std::string remove_right_zeros(std::string sNUMBER);
using namespace std;
```

3.2 Main

The following code is the main function for the entry point of the program.

This function starts the program and takes the expression from the user and waits for the user to press the Enter Key to parse the calculator functions.

```
int main()
{

	std::string sEXP;
	std::string sRETURN;

	std::cout << "Standard BODMAS Calculator Program" << "\n";
	std::cout << "\n";
	std::cout << "Hello and Thank You For Your Time" << "\n";
	ofstream MyFileWrite ("MyFinalResults.txt");

AGAIN:
	std::cout << "To Begin: Enter Your Expression" << "\n";
	std::cout << "Use NUMBERs and/or + - * / ^ r . ( ) [ ] { }" << "\n";
	std::cout << "To End: Enter 'EXIT' To End Program" << "\n";
	std::cout << "Enter An Expression To Continue...\n"; // requesting to enter an expression
	std::cin >> sEXP; // reading expression
	if (sEXP == "EXIT") {
	    cout << "Final Results File Created/Updated Successfully!" << "\n";
	    cout << "Good Bye and Thank You For Your Time" << "\n";
	    cout << "Exiting Standard BODMAS Calculator Program..." << "\n";
	    return 0;

	}
	std::cout << "You have enter " << sEXP << "\n"; // expression confirmation
	system("PAUSE");
	sRETURN = validate_expression(sEXP);
	if (sEXP != sRETURN)
	{
		// invalid character has detected in the expression
		sEXP = sRETURN; // update new expression
		goto START; // going START to cancel the operation
	}
	sRETURN = insert_multiplication_near_bracket(sEXP);
	if (sEXP != sRETURN)
	{
		sEXP = sRETURN; // update new expression

	}

START:
	std::cout << "=" << sEXP << "\n"; // print expression

	if (bCancelFlag == true)
	goto CANCEL;

	sRETURN = calculate_operators(sEXP);
	if (sEXP != sRETURN)
	{
		sEXP = sRETURN; // update new expression
		goto START;
	}
	sRETURN = first_bracket(sEXP);
	if (sEXP != sRETURN)
	{
		sEXP = sRETURN; // update new expression
		goto START;
	}
	sRETURN = second_bracket(sEXP);
	if (sEXP != sRETURN)
	{
		sEXP = sRETURN; // update new expression
		goto START;
	}
	sRETURN = third_bracket(sEXP);
	if (sEXP != sRETURN)
	{
		sEXP = sRETURN; // update new expression
		goto START;
	}
	sRETURN = BODMAS_JW(sEXP);
	if (sEXP != sRETURN)
	{
```

```
                sEXP = sRETURN; // update new expression

                goto START;
        }

    MyFileWrite << sEXP;
    MyFileWrite.close ();
    MyFileWrite.open("MyFinalResults.txt", std::fstream::out | std::fstream::trunc);
    MyFileWrite << "Good News! Your Final Results is: " << sEXP;
    MyFileWrite.close ();
    cout << "Final Results Saved/Updated Successfully!" << "\n";


CANCEL:
        bCancelFlag = false;
        std::cout << "Enter 'EXIT' To End Program.\n";
        goto AGAIN;
        //system("PAUSE");
        return 0;

}
```

3.3 Validation

The following code is the validation function of the expression.

The function finds the invalid characters in the expression.

For example, not allowed characters: “a”, “b”, “c”, “&” etc.

And it allows "0123456789+-*/^r.()[]{}" only.

```
std::string validate_expression(std::string sEXP) {
        std::string sValidChars = "0123456789+-*/^r.()[]{}";
        std::string sMsg;
        std::size_t nFound = 0;

        nFound = sEXP.find_first_not_of(sValidChars);
        if (nFound != std::string::npos)
        {
                bCancelFlag = true;
                sMsg.append("Invalid character found in the expression '");
                std::string ch;
                ch = sEXP[nFound];
                sMsg.append(ch);
                sMsg.append("' .");
                return sMsg;

        }

        return sEXP;
}
```

3.4 Multiplication Sign

The following code uses a function to insert multiplication (*) sign between the number and bracket of the expression for an easy process in the program.

For example, if user enters expression “2+6{+5(3+9)+8}”, the function will return 2+6*{+5*(3+9)+8}.

```
std::string insert_multiplication_near_bracket(std::string sEXP){
        std::string sNUMBERs = "0123456789";
        std::string sFirstBrackets = "[{(";
        std::string sSecondBrackets = ")}]";
        std::string sFirstEXP;
        std::string sMiddleEXP;
        std::string sLastEXP;
```

```
	std::string sFinalEXP;
	std::size_t nStart = 0;
	std::size_t nEnd = 0;
	std::size_t nFound = 0;

	// inserting multiplication(*) between the number and first brackets [{(
	nFound = sEXP.find_first_of(sFirstBrackets);
	while (nFound != std::string::npos)
	{
		if (nFound != 0)
		{
			nStart = nFound; nEnd = nFound;
			std::size_t i = sNUMBERs.find_first_of(sEXP[nFound - 1]);
			if (i != std::string::npos)
			{
				sFirstEXP = sEXP.substr(0, nStart);
				sMiddleEXP = "*"; // inserting multiplication (*)
				sLastEXP = sEXP.substr(nEnd, sEXP.length() - nEnd);
				sFinalEXP.clear();
				sFinalEXP.append(sFirstEXP);
				sFinalEXP.append(sMiddleEXP);
				sFinalEXP.append(sLastEXP);
				sEXP = sFinalEXP;
			}
		}
		nFound = sEXP.find_first_of(sFirstBrackets, nFound +1);
	}


	// inserting multiplication(*) between the number and second brackets )}]
	nFound = sEXP.find_first_of(sSecondBrackets);
	while (nFound != std::string::npos)
	{
		if (nFound != 0)
		{
			nStart = nFound; nEnd = nFound +1;
			std::size_t i = sNUMBERs.find_first_of(sEXP[nFound + 1]);
			if (i != std::string::npos)
			{
				sFirstEXP = sEXP.substr(0, nStart +1);
				sMiddleEXP = "*"; // inserting multiplication (*)
				sLastEXP = sEXP.substr(nEnd, sEXP.length() - nEnd);
				sFinalEXP.clear();
				sFinalEXP.append(sFirstEXP);
				sFinalEXP.append(sMiddleEXP);
				sFinalEXP.append(sLastEXP);
				sEXP = sFinalEXP;
			}
		}
		nFound = sEXP.find_first_of(sSecondBrackets, nFound + 1);
	}

	return sEXP;
}
```

3.5 Operators

The following code of function is used to calculate operators.

For example, “++” will make “+”

“+-“ will make “-“,

“-+” will make “-“,

and

“- -” will make “+”.

This happens and is necessary when removing brackets.

For example, expression 2-(-5+3) ➡ 2- -2 ➡ 2+2 ➡ 4.

```
std::string calculate_operators(std::string sEXP) {
	std::string sPlusMinus = "+-";
	std::string sMinusPlus = "-+";
	std::string sMinusMinus = "--";
	std::string sPlusPlus = "++";
	std::string sFirstEXP;
	std::string sMiddleEXP;
	std::string sLastEXP;
	std::string sFinalEXP;
	std::size_t nStart = 0;
	std::size_t nEnd = 0;
	std::size_t nFound = 0;

	// working for +-
	nFound = sEXP.find(sPlusMinus);
	if (nFound != std::string::npos)
	{
		sFirstEXP = sEXP.substr(0, nFound);
		sMiddleEXP = "-";
		sLastEXP = sEXP.substr(nFound + sPlusMinus.length(), sEXP.length() - (nFound +
			sPlusMinus.length()));
		sFinalEXP.append(sFirstEXP);
		sFinalEXP.append(sMiddleEXP);
		sFinalEXP.append(sLastEXP);
		return sFinalEXP;
	}

	// working for -+
	nFound = sEXP.find(sMinusPlus);
	if (nFound != std::string::npos)
	{
		sFirstEXP = sEXP.substr(0, nFound);
		sMiddleEXP = "-";
		sLastEXP = sEXP.substr(nFound + sMinusPlus.length(), sEXP.length() - (nFound +
			sMinusPlus.length()));
		sFinalEXP.append(sFirstEXP);
		sFinalEXP.append(sMiddleEXP);
		sFinalEXP.append(sLastEXP);
		return sFinalEXP;
	}

	// working for --
	nFound = sEXP.find(sMinusMinus);
	if (nFound != std::string::npos)
	{
		sFirstEXP = sEXP.substr(0, nFound);
		sMiddleEXP = "+";
		sLastEXP = sEXP.substr(nFound + sMinusMinus.length(), sEXP.length() - (nFound +
			sMinusMinus.length()));
		sFinalEXP.append(sFirstEXP);
		sFinalEXP.append(sMiddleEXP);
		sFinalEXP.append(sLastEXP);
		return sFinalEXP;
	}

	// working for ++
	nFound = sEXP.find(sPlusPlus);
	if (nFound != std::string::npos)
	{
		sFirstEXP = sEXP.substr(0, nFound);
		sMiddleEXP = "+";
		sLastEXP = sEXP.substr(nFound + sPlusPlus.length(), sEXP.length() - (nFound +
			sPlusPlus.length()));
		sFinalEXP.append(sFirstEXP);
		sFinalEXP.append(sMiddleEXP);
		sFinalEXP.append(sLastEXP);
		return sFinalEXP;
	}
	return sEXP;
}
```

3.6 First Bracket

The following code is used for the first type of bracket “()”.

It retrieves expression within the brackets and removes when the inner expression has completed its operation by BODMAS_JW function of this program.

```
std::string first_bracket(std::string sEXP) {
        std::string sOperators = "^r*/+-";
        std::string sMsg;
        std::string sFirstFind = "(";
        std::string sSecondFind = ")";
        std::string sFirstEXP;
        std::string sLastEXP;
        std::string sFinalEXP;
        std::string sBODMAS_JW;
        std::string sRetBODMAS_JW;
        std::size_t nStart = 0;
        std::size_t nEnd = 0;
        std::size_t nFound = sEXP.find(sFirstFind);

        while (nFound != std::string::npos)
        {
                nStart = nFound;
                //std::cout << "Found '" << sFirstFind << "' at " << nStart << "\n";
                nFound = sEXP.find(sFirstFind, nStart + 1);
        }

        if (nStart == 0) return sEXP;

        nFound = 0;
        nFound = sEXP.find(sSecondFind, nStart + 1);
        if (nFound != std::string::npos)
        {
                nEnd = nFound;
                //std::cout << "Found '" << sSecondFind << "' at " << nEnd << "\n";
        }
        else
        {
                sMsg = "Error on bracket close.\n";
                //std::cout << sMsg;
                return sMsg ;
        }

        if (nEnd == 0)
        {
                sMsg = "Error on bracket close.\n";
                //std::cout << sMsg;
                return sMsg ;
        }

        sBODMAS_JW = sEXP.substr(nStart + 1, nEnd - nStart -1);

        //std::cout << "The BODMAS_JW is " << sBODMAS_JW << "\n";
        // Is it expression or numbers only?
       If it is a number only then remove the bracket and return

        if (sBODMAS_JW[0] == '-') // if signed expression
        {
                nFound = sBODMAS_JW.find_first_of(sOperators,1);
                // skip sign to check other operator
        }
        else
        {
                nFound = sBODMAS_JW.find_first_of(sOperators);
        }

        if (nFound == std::string::npos)
        {
                // numbers only found no expression, removing brackets
                sFirstEXP = sEXP.substr(0, nStart);
                sLastEXP = sEXP.substr(nEnd + 1, sEXP.length() - (nEnd + 1));
                sFinalEXP.append(sFirstEXP);
                sFinalEXP.append(sBODMAS_JW);
```

```
                sFinalEXP.append(sLastEXP);
                return sFinalEXP;
        }

        sRetBODMAS_JW = BODMAS_JW(sBODMAS_JW);
        sFirstEXP = sEXP.substr(0, nStart +1);
        sLastEXP = sEXP.substr(nEnd, sEXP.length() - nEnd);
        sFinalEXP.append(sFirstEXP);
        sFinalEXP.append(sRetBODMAS_JW);
        sFinalEXP.append(sLastEXP);
        //std::cout << "The final expression of first bracket is " << sFinalEXP << "\n";
        return sFinalEXP;

}
```

3.7 Second Bracket

The following code is used for the second type of bracket “{}”.

It retrieves expression within the brackets and removes when the inner expression has completed its operation by BODMAS_JW function of this program.

```
std::string second_bracket(std::string sEXP) {
        std::string sOperators = "^r*/+-";
        std::string sMsg;
        std::string sFirstFind = "{";
        std::string sSecondFind = "}";
        std::string sFirstEXP;
        std::string sLastEXP;
        std::string sFinalEXP;
        std::string sBODMAS_JW;
        std::string sRetBODMAS_JW;
        std::size_t nStart = 0;
        std::size_t nEnd = 0;
        std::size_t nFound = sEXP.find(sFirstFind);

        while (nFound != std::string::npos)
        {
                nStart = nFound;
                //std::cout << "Found '" << sFirstFind << "' at " << nStart << "\n";
                nFound = sEXP.find(sFirstFind, nStart + 1);
        }

        if (nStart == 0) return sEXP;

        nFound = 0;
        nFound = sEXP.find(sSecondFind, nStart + 1);
        if (nFound != std::string::npos)
        {
                nEnd = nFound;
                //std::cout << "Found '" << sSecondFind << "' at " << nEnd << "\n";
        }
        else
        {
                sMsg = "Error on bracket close.\n";
                //std::cout << sMsg;
                return sMsg;
        }

        if (nEnd == 0)
        {
                sMsg = "Error on bracket close.\n";
                //std::cout << sMsg;
                return sMsg;
        }

        sBODMAS_JW = sEXP.substr(nStart + 1, nEnd - nStart - 1);

        //std::cout << "The BODMAS_JW is " << sBODMAS_JW << "\n";
        // Is it expression or numbers only?
        If it is a number only then remove the bracket and return

        if (sBODMAS_JW[0] == '-') // if signed expression
        {
```

```
                nFound = sBODMAS_JW.find_first_of(sOperators, 1);
                // skip sign to check other operator
        }
        else
        {
                nFound = sBODMAS_JW.find_first_of(sOperators);
        }
        if (nFound == std::string::npos)
        {
                // numbers only found no expression, removing brackets
                sFirstEXP = sEXP.substr(0, nStart);
                sLastEXP = sEXP.substr(nEnd + 1, sEXP.length() - (nEnd + 1));
                sFinalEXP.append(sFirstEXP);
                sFinalEXP.append(sBODMAS_JW);
                sFinalEXP.append(sLastEXP);
                return sFinalEXP;
        }

        sRetBODMAS_JW = BODMAS_JW(sBODMAS_JW);
        sFirstEXP = sEXP.substr(0, nStart + 1);
        sLastEXP = sEXP.substr(nEnd, sEXP.length() - nEnd);
        sFinalEXP.append(sFirstEXP);
        sFinalEXP.append(sRetBODMAS_JW);
        sFinalEXP.append(sLastEXP);
        //std::cout << "The final expression of first bracket is " << sFinalEXP << "\n";
        return sFinalEXP;

}
```

3.8 Third Bracket

The following code is used for the third type of bracket “[]”.

It retrieves expression within the brackets and removes when the inner expression has completed its operation by BODMAS_JW function of this program.

```
std::string third_bracket(std::string sEXP) {
        std::string sOperators = "^r*/+-";
        std::string sMsg;
        std::string sFirstFind = "[";
        std::string sSecondFind = "]";
        std::string sFirstEXP;
        std::string sLastEXP;
        std::string sFinalEXP;
        std::string sBODMAS_JW;
        std::string sRetBODMAS_JW;
        std::size_t nStart = 0;
        std::size_t nEnd = 0;
        std::size_t nFound = sEXP.find(sFirstFind);

        while (nFound != std::string::npos)
        {
                nStart = nFound;
                //std::cout << "Found '" << sFirstFind << "' at " << nStart << "\n";
                nFound = sEXP.find(sFirstFind, nStart + 1);
        }

        if (nStart == 0) return sEXP;

        nFound = 0;
        nFound = sEXP.find(sSecondFind, nStart + 1);
        if (nFound != std::string::npos)
        {
                nEnd = nFound;
                //std::cout << "Found '" << sSecondFind << "' at " << nEnd << "\n";
        }
        else
        {
                sMsg = "Error on bracket close.\n";
                //std::cout << sMsg;
                return sMsg;
        }
```

```
        if (nEnd == 0)
        {
                sMsg = "Error on bracket close.\n";
                //std::cout << sMsg;
                return sMsg;
        }

        sBODMAS_JW = sEXP.substr(nStart + 1, nEnd - nStart - 1);

        //std::cout << "The BODMAS_JW is " << sBODMAS_JW << "\n";
        // Is it expression or numbers only?
        If it is a number only then remove the bracket and return

        if (sBODMAS_JW[0] == '-') // if signed expression
        {
                nFound = sBODMAS_JW.find_first_of(sOperators, 1);
                // skip sign to check other operator
        }
        else
        {
                nFound = sBODMAS_JW.find_first_of(sOperators);
        }
        if (nFound == std::string::npos)
        {
                // numbers only found no expression, removing brackets
                sFirstEXP = sEXP.substr(0, nStart);
                sLastEXP = sEXP.substr(nEnd + 1, sEXP.length() - (nEnd + 1));
                sFinalEXP.append(sFirstEXP);
                sFinalEXP.append(sBODMAS_JW);
                sFinalEXP.append(sLastEXP);
                return sFinalEXP;
        }

        sRetBODMAS_JW = BODMAS_JW(sBODMAS_JW);
        sFirstEXP = sEXP.substr(0, nStart + 1);
        sLastEXP = sEXP.substr(nEnd, sEXP.length() - nEnd);
        sFinalEXP.append(sFirstEXP);
        sFinalEXP.append(sRetBODMAS_JW);
        sFinalEXP.append(sLastEXP);
        //std::cout << "The final expression of first bracket is " << sFinalEXP << "\n";
        return sFinalEXP;

}
```

3.9 Regular Expression Without Brackets

The following code is used to calculate regular expression without brackets.

It uses sub functions order_exponent_root(sExp), division_multiplication(sOER) and addition_subtraction(sDM).

```
std::string BODMAS JW(std::string sEXP) {
        std::string sOER;
        std::string sDM;
        std::string sAS;

        //std::cout << "BODMAS_JW Processing " << sEXP << "\n";
        sOER = order exponent_root(sEXP);
        if (sOER != sEXP)
        {
                //std::cout << "Order-EXPonent-Root has done " << sOER << "\n";
                return sOER;
        }
        sDM = division multiplication(sOER);
        if (sDM != sOER)
        {
                //std::cout << "Division-Multiplication has done " << sDM << "\n";
                return sDM;
        }
        sAS = addition_subtraction(sDM);
        if (sAS != sDM)
        {
```

```
                //std::cout << "Addition_Subtraction has done " << sAS << "\n";
                return sAS;
        }
        return sEXP;
}
```

3.10 Order/Power/Exponent

The following code is used to calculate "^" order/power/exponent and "r" root squire/cube.

It will process from left to right of the expression; whichever first.

And it will use one cycle for one "^ / r".

```
std::string order_exponent_root(std::string sEXP) {

        //std::cout << "Order EXPonent Root Processing " << sEXP << "\n";
        std::string sOperators = "^r*/+-";
        std::string sNUMBERs = "0123456789.";
        char sEXPonent = '^';
        char sRoot = 'r';
        float fFirstNum = 0;
        float fLastNum = 0;
        std::string sMsg;
        std::string sSearch = "r^";
        std::string sFirstNumStr = "            ";
        std::string sLastNumStr = "            ";
        std::string sResult;
        std::string sFirstEXP;
        std::string sLastEXP;
        std::string sFinalEXP;
        std::size_t  nMaxNum = 12;
        std::size_t nPos = 0;
        std::size_t nNum = 0;
        std::size_t nStart = 0;
        std::size_t nEnd = 0;
        std::size_t nFound = 0;
        bool bIsSigned = false;
        bool bStopCheckSign = false;

        nFound = sEXP.find_first_of(sSearch);
        // detecting invalid expresion
        if (nFound == 0)
        {
                // turn on cancel flag and return with error message
                bCancelFlag = true;
                sMsg.append("Invalid use of ");
                std::string ch;
                ch= sEXP[nFound];
                sMsg.append(ch);
                return sMsg;
        }

        if (nFound != std::string::npos)
        {
                if (sEXP[nFound] == sEXPonent) // starts exponents process
                {
                        nNum = 0;
                        for (int i = nFound -1; i > -1; --i) {
                                nPos = sOperators.find(sEXP[i]);
                                if (nPos != std::string::npos) break;
                                nPos = 0;
                                nPos = sNUMBERs.find(sEXP[i]);
                                if (nPos == std::string::npos) break;
                                if (nNum > nMaxNum)
                                {
                                        bCancelFlag = true;
                                        return "The exponent first maximum number dizits overflow.";
                                }
                                if (nPos > 0 || nPos == 0)
                                {
                                        nNum++;
                                        sFirstNumStr[nMaxNum - nNum] = (sEXP[i]);
```

```
                }
        }

        fFirstNum = std::stof(sFirstNumStr,nullptr);
        sFirstNumStr = std::to_string(fFirstNum);
        sFirstNumStr = remove_right_zeros(sFirstNumStr);
        //std::cout << "The exponent first number is " << sFirstNumStr << "\n";

        nNum = 0;
        for (std::size_t i = nFound; i < (sEXP.length()); ++i) {
                nPos = sOperators.find(sEXP[i+1]);
                if (nPos != std::string::npos)
                {
                        if (sOperators[nPos] == '-' && bStopCheckSign==false)
                        {

                                bIsSigned = true;
                                continue; // turn on signed flag for last number
                        }
                        else
                        {
                                break;
                        }
                }
                bStopCheckSign = true;
                nPos = 0;
                nPos = sNUMBERs.find(sEXP[i+1]);
                if (nPos == std::string::npos) break;
                if (nNum > nMaxNum)
                {
                        bCancelFlag = true;
                        return "The exponent last maximum number dizits overflow.";
                }
                if (nPos > 0 || nPos == 0)
                {
                        sLastNumStr[nNum] = (sEXP[i+1]);
                        nNum++;
                }
        }

        fLastNum = std::stof(sLastNumStr,nullptr);
        if (bIsSigned == true) fLastNum = fLastNum * -1;
        // add a sign(-) on first number
        sLastNumStr = std::to_string(fLastNum);
        sLastNumStr = remove_right_zeros(sLastNumStr);
        //std::cout << "The exponent last number is " << sLastNumStr << "\n";

        sResult = std::to_string((float)std::pow(fFirstNum, fLastNum));
        sResult = remove_right_zeros(sResult);
        nStart = nFound - sFirstNumStr.length();
        nEnd = nStart + sFirstNumStr.length() + sLastNumStr.length() + 1;
        // +1 becuase of + or - sign is there
        sFirstEXP = sEXP.substr(0, nStart);
        sLastEXP = sEXP.substr(nEnd, sEXP.length() - nEnd);
        sFinalEXP.append(sFirstEXP);
        sFinalEXP.append(sResult);
        sFinalEXP.append(sLastEXP);

        return sFinalEXP;
}

if (sEXP[nFound] == sRoot) // starts root process
{
        nNum = 0;
        for (int i = nFound - 1; i > -1; --i) {
                nPos = sOperators.find(sEXP[i]);
                if (nPos != std::string::npos) break;
                nPos = 0;
                nPos = sNUMBERs.find(sEXP[i]);
                if (nPos == std::string::npos) break;
                if (nNum > nMaxNum)
                {
                        bCancelFlag = true;
                        return "The root first maximum number dizits overflow.";
                }
                if (nPos > 0 || nPos == 0)
                {
                        nNum++;
```

```
                                sFirstNumStr[nMaxNum - nNum] = (sEXP[i]);
                        }
                }

                fFirstNum = std::stof(sFirstNumStr, nullptr);
                sFirstNumStr = std::to_string(fFirstNum);
                sFirstNumStr = remove_right_zeros(sFirstNumStr);
                //std::cout << "The root first number is " << sFirstNumStr << "\n";

                nNum = 0;
                for (std::size_t i = nFound; i < (sEXP.length()); ++i) {
                        nPos = sOperators.find(sEXP[i + 1]);
                        if (nPos != std::string::npos)
                        {
                                if (sOperators[nPos] == '-' && bStopCheckSign==false)
                                {
                                        bIsSigned = true;
                                        // turn on signed flag for last number
                                        continue;
                                }
                                else
                                {
                                        break;
                                }
                        }
                        bStopCheckSign = true;
                        nPos = 0;
                        nPos = sNUMBERs.find(sEXP[i + 1]);
                        if (nPos == std::string::npos) break;
                        if (nNum > nMaxNum)
                        {
                                bCancelFlag = true;
                                return "The root last maximum number dizits overflow.";
                        }
                        if (nPos > 0 || nPos == 0)
                        {
                                sLastNumStr[nNum] = (sEXP[i + 1]);
                                nNum++;
                        }
                }

                fLastNum = std::stof(sLastNumStr, nullptr);
                if (bIsSigned == true) fLastNum = fLastNum * -1;
                 // add a sign(-) on first number
                sLastNumStr = std::to_string(fLastNum);
                sLastNumStr = remove_right_zeros(sLastNumStr);
                //std::cout << "The root last number is " << sLastNumStr << "\n";

                switch ((std::size_t)fLastNum)
                {
                case 2:
                        sResult = std::to_string((float)std::sqrt(fFirstNum));
                        break;
                case 3:
                        sResult = std::to_string((float)std::cbrt(fFirstNum));
                        break;
                default:
                        break;
                }
                sResult = remove_right_zeros(sResult);
                nStart = nFound - sFirstNumStr.length();
                nEnd = nStart + sFirstNumStr.length() + sLastNumStr.length() + 1;
               // +1 becuase of ^ or r sign is there
                sFirstEXP = sEXP.substr(0, nStart);
                sLastEXP = sEXP.substr(nEnd, sEXP.length() - nEnd);
                sFinalEXP.append(sFirstEXP);
                sFinalEXP.append(sResult);
                sFinalEXP.append(sLastEXP);

                return sFinalEXP;
        }
    }


    return sEXP;

}
```

3.11 Division/Multiplication

The following code is used to calculate “/” division and “*” multiplication.

It will process from left to right of the expression; whichever first.

And it will use one cycle for one “/ or *”.

```
std::string division_multiplication(std::string sEXP) {

	//std::cout << "division multiplication Processing " << sEXP << "\n";
	std::string sOperators = "^r*/+-";
	std::string sNUMBERs = "0123456789.";
	char sDivision = '/';
	char sMultiplication = '*';
	float fFirstNum = 0;
	float fLastNum = 0;
	std::string sSearch = "*/";
	std::string sFirstNumStr = "            ";
	std::string sLastNumStr = "            ";
	std::string sResult;
	std::string sFirstEXP;
	std::string sLastEXP;
	std::string sFinalEXP;
	std::string sMsg;
	std::size_t  nMaxNum = 12;
	std::size_t nPos = 0;
	std::size_t nNum = 0;
	std::size_t nStart = 0;
	std::size_t nEnd = 0;
	std::size_t nFound = 0;
	bool bIsSigned = false;
	bool bStopCheckSign = false;

	nFound = sEXP.find_first_of(sSearch);
	// detecting invalid expresion
	if (nFound == 0)
	{
		// turn on cancel flag and return with error message
		bCancelFlag = true;
		sMsg.append("Invalid use of ");
		std::string ch;
		ch = sEXP[nFound];
		sMsg.append(ch);
		return sMsg;
	}
	if (nFound != std::string::npos)
	{
		if (sEXP[nFound] == sDivision) // starts division process
		{
			nNum = 0;
			for (int i = nFound - 1; i > -1; --i) {
				nPos = sOperators.find(sEXP[i]);
				if (nPos != std::string::npos) break;
				nPos = 0;
				nPos = sNUMBERs.find(sEXP[i]);
				if (nPos == std::string::npos) break;
				if (nNum > nMaxNum)
				{
					bCancelFlag = true;
					return "The division first maximum number dizits overflow.";
				}
				if (nPos > 0 || nPos == 0)
				{
					nNum++;
					sFirstNumStr[nMaxNum - nNum] = (sEXP[i]);
				}
			}


			fFirstNum = std::stof(sFirstNumStr, nullptr);
			sFirstNumStr = std::to_string(fFirstNum);
			sFirstNumStr = remove_right_zeros(sFirstNumStr);
```

```
        //std::cout << "The division first number is " << sFirstNumStr << "\n";

        nNum = 0;
        for (std::size_t i = nFound; i < (sEXP.length()); ++i) {
                nPos = sOperators.find(sEXP[i + 1]);
                if (nPos != std::string::npos)
                {
                        if (sOperators[nPos] == '-' && bStopCheckSign == false)
                        {
                                bIsSigned = true;
                                // turn on signed flag for last number
                                continue;
                        }
                        else
                        {
                                break;
                        }
                }
                bStopCheckSign = true;
                nPos = 0;
                nPos = sNUMBERs.find(sEXP[i + 1]);
                if (nPos == std::string::npos) break;
                if (nNum > nMaxNum)
                {
                        bCancelFlag = true;
                        return "The division last maximum number dizits overflow.";
                }
                if (nPos > 0 || nPos == 0)
                {
                        sLastNumStr[nNum] = (sEXP[i + 1]);
                        nNum++;
                }
        }

        fLastNum = std::stof(sLastNumStr, nullptr);
        if (bIsSigned == true) fLastNum = fLastNum * -1;
        // add a sign(-) on first number
        sLastNumStr = std::to_string(fLastNum);
        sLastNumStr = remove_right_zeros(sLastNumStr);
        //std::cout << "The division last number is " << sLastNumStr << "\n";

        sResult = std::to_string(fFirstNum/fLastNum);
        sResult = remove_right_zeros(sResult);
        nStart = nFound - sFirstNumStr.length();
        nEnd = nStart + sFirstNumStr.length() + sLastNumStr.length() + 1;
        // +1 becuase of + or - sign is there
        sFirstEXP = sEXP.substr(0, nStart);
        sLastEXP = sEXP.substr(nEnd, sEXP.length() - nEnd);
        sFinalEXP.append(sFirstEXP);
        sFinalEXP.append(sResult);
        sFinalEXP.append(sLastEXP);

        return sFinalEXP;
}

if (sEXP[nFound] == sMultiplication) // starts root process
{
        nNum = 0;
        for (int i = nFound - 1; i > -1; --i) {
                nPos = sOperators.find(sEXP[i]);
                if (nPos != std::string::npos) break;
                nPos = 0;
                nPos = sNUMBERs.find(sEXP[i]);
                if (nPos == std::string::npos) break;
                if (nNum > nMaxNum)
                {
                        bCancelFlag = true;
                        return "The multiplication first maximum number dizits
                         overflow.";
                }
                if (nPos > 0 || nPos == 0)
                {
                        nNum++;
                        sFirstNumStr[nMaxNum - nNum] = (sEXP[i]);
                }
        }
```

```
                        fFirstNum = std::stof(sFirstNumStr, nullptr);
                        sFirstNumStr = std::to_string(fFirstNum);
                        sFirstNumStr = remove_right_zeros(sFirstNumStr);
                        //std::cout << "The multiplication first number is " << sFirstNumStr <<
                        "\n";
                        nNum = 0;
                        for (std::size_t i = nFound; i < (sEXP.length()); ++i) {
                                nPos = sOperators.find(sEXP[i + 1]);
                                if (nPos != std::string::npos)
                                {
                                        if (sOperators[nPos] == '-'  && bStopCheckSign == false)
                                        {
                                                bIsSigned = true;
                                                // turn on signed flag for last number
                                                continue;
                                        }
                                        else
                                        {
                                                break;
                                        }
                                }
                                bStopCheckSign = true;
                                nPos = 0;
                                nPos = sNUMBERs.find(sEXP[i + 1]);
                                if (nPos == std::string::npos) break;
                                if (nNum > nMaxNum)
                                {
                                        bCancelFlag = true;
                                        return "The multiplication last maximum number dizits
                                        overflow.";
                                }
                                if (nPos > 0 || nPos == 0)
                                {
                                        sLastNumStr[nNum] = (sEXP[i + 1]);
                                        nNum++;
                                }
                        }

                        fLastNum = std::stof(sLastNumStr, nullptr);
                        if (bIsSigned == true) fLastNum = fLastNum * -1;
                        // add a sign(-) on first number
                        sLastNumStr = std::to_string(fLastNum);
                        sLastNumStr = remove_right_zeros(sLastNumStr);
                        //std::cout << "The multiplication last number is " << sLastNumStr << "\n";

                        sResult = std::to_string(fFirstNum * fLastNum);
                        sResult = remove_right_zeros(sResult);
                        nStart = nFound - sFirstNumStr.length();
                        nEnd = nStart + sFirstNumStr.length() + sLastNumStr.length() + 1;
                       // +1 becuase of ^ or r sign is there
                        sFirstEXP = sEXP.substr(0, nStart);
                        sLastEXP = sEXP.substr(nEnd, sEXP.length() - nEnd);
                        sFinalEXP.append(sFirstEXP);
                        sFinalEXP.append(sResult);
                        sFinalEXP.append(sLastEXP);

                        return sFinalEXP;
                }
        }


        return sEXP;
}
```

3.12 Addition/Subtraction

The following code is used to calculate “+” addition and “-” subtraction.

It will process from left to right of the expression; whichever first.

It will use one cycle for one “+ or -”.

```
std::string addition_subtraction(std::string sEXP) {

	//std::cout << "addition subtraction Processing " << sEXP << "\n";
	std::string sOperators = "^r*/+-";
	std::string sNUMBERs = "0123456789.";
	char sAddition = '+';
	char sSubtraction = '-';
	float fFirstNum = 0;
	float fLastNum = 0;
	std::string sSearch = "+-";
	std::string sFirstNumStr = "            ";
	std::string sLastNumStr = "            ";
	std::string sResult;
	std::string sFirstEXP;
	std::string sLastEXP;
	std::string sFinalEXP;
	std::size_t  nMaxNum = 12;
	std::size_t nPos = 0;
	std::size_t nNum = 0;
	std::size_t nStart = 0;
	std::size_t nEnd = 0;
	std::size_t nFound = 0;
	bool bIsSigned = false;

	nFound = sEXP.find_first_of(sSearch);
		// detecting signed expresion
	if (nFound == 0)
	{
		// turn on sized expression flag
		if (sEXP[nFound] == sSubtraction) bIsSigned = true;
		nFound = sEXP.find_first_of(sSearch,1); // repeat search skip sign(-)
	}
	if (nFound != std::string::npos)
	{
		if (sEXP[nFound] == sAddition) // starts addition process
		{
			nNum = 0;
			for (int i = nFound - 1; i > -1; --i) {
				nPos = sOperators.find(sEXP[i]);
				if (nPos != std::string::npos) break;
				nPos = 0;
				nPos = sNUMBERs.find(sEXP[i]);
				if (nPos == std::string::npos) break;
				if (nNum > nMaxNum)
				{
					bCancelFlag = true;
					return "The addition first maximum number dizits overflow.";
				}
				if (nPos > 0 || nPos == 0)
				{
					nNum++;
					sFirstNumStr[nMaxNum - nNum] = (sEXP[i]);
				}
			}

			fFirstNum = std::stof(sFirstNumStr, nullptr);
			if (bIsSigned == true) fFirstNum = fFirstNum * -1;
			// add a sign(-) on first number
			sFirstNumStr = std::to_string(fFirstNum);
			sFirstNumStr = remove_right_zeros(sFirstNumStr);
			//std::cout << "The addition first number is " << sFirstNumStr << "\n";

			nNum = 0;
			for (std::size_t i = nFound; i < (sEXP.length()); ++i) {
				nPos = sOperators.find(sEXP[i + 1]);
				if (nPos != std::string::npos) break;
				nPos = 0;
				nPos = sNUMBERs.find(sEXP[i + 1]);
				if (nPos == std::string::npos) break;
				if (nNum > nMaxNum)
				{
					bCancelFlag = true;
					return "The addition last maximum number dizits overflow.";
				}
				if (nPos > 0 || nPos == 0)
				{
					sLastNumStr[nNum] = (sEXP[i + 1]);
					nNum++;
```

```
                }
        }

        fLastNum = std::stof(sLastNumStr, nullptr);
        sLastNumStr = std::to_string(fLastNum);
        sLastNumStr = remove_right_zeros(sLastNumStr);
        //std::cout << "The addition last number is " << sLastNumStr << "\n";

        sResult = std::to_string(fFirstNum + fLastNum);
        sResult = remove_right_zeros(sResult);
        nStart = nFound - sFirstNumStr.length();
        nEnd = nStart + sFirstNumStr.length() + sLastNumStr.length() + 1;
        // +1 becuase of + or - sign is there
        sFirstEXP = sEXP.substr(0, nStart);
        sLastEXP = sEXP.substr(nEnd, sEXP.length() - nEnd);
        sFinalEXP.append(sFirstEXP);
        sFinalEXP.append(sResult);
        sFinalEXP.append(sLastEXP);

        return sFinalEXP;
}

if (sEXP[nFound] == sSubtraction) // starts subtraction process
{
        nNum = 0;
        for (int i = nFound - 1; i > -1; --i) {
                nPos = sOperators.find(sEXP[i]);
                if (nPos != std::string::npos) break;
                nPos = 0;
                nPos = sNUMBERs.find(sEXP[i]);
                if (nPos == std::string::npos) break;
                if (nNum > nMaxNum)
                {
                        bCancelFlag = true;
                        return "The subtraction first maximum number dizits
                        overflow.";
                }
                if (nPos > 0 || nPos == 0)
                {
                        nNum++;
                        sFirstNumStr[nMaxNum - nNum] = (sEXP[i]);
                }
        }

        fFirstNum = std::stof(sFirstNumStr, nullptr);
        if (bIsSigned == true) fFirstNum = fFirstNum * -1;
        // add a sign(-) on first number
        sFirstNumStr = std::to_string(fFirstNum);
        sFirstNumStr = remove_right_zeros(sFirstNumStr);
        //std::cout << "The subtraction first number is " << sFirstNumStr << "\n";

        nNum = 0;
        for (std::size_t i = nFound; i < (sEXP.length()); ++i) {
                nPos = sOperators.find(sEXP[i + 1]);
                if (nPos != std::string::npos) break;
                nPos = 0;
                nPos = sNUMBERs.find(sEXP[i + 1]);
                if (nPos == std::string::npos) break;
                if (nNum > nMaxNum)
                {
                        bCancelFlag = true;
                        return "The subtraction last maximum number dizits
                        overflow.";
                }
                if (nPos > 0 || nPos == 0)
                {
                        sLastNumStr[nNum] = (sEXP[i + 1]);
                        nNum++;
                }
        }

        fLastNum = std::stof(sLastNumStr, nullptr);
        sLastNumStr = std::to_string(fLastNum);
        sLastNumStr = remove_right_zeros(sLastNumStr);
        //std::cout << "The subtraction last number is " << sLastNumStr << "\n";

        sResult = std::to_string(fFirstNum - fLastNum);
        sResult = remove_right_zeros(sResult);
```

```
                        nStart = nFound - sFirstNumStr.length();
                        nEnd = nStart + sFirstNumStr.length() + sLastNumStr.length() + 1;
                        // +1 becuase of ^ or r sign is there
                        sFirstEXP = sEXP.substr(0, nStart);
                        sLastEXP = sEXP.substr(nEnd, sEXP.length() - nEnd);
                        sFinalEXP.append(sFirstEXP);
                        sFinalEXP.append(sResult);
                        sFinalEXP.append(sLastEXP);

                        return sFinalEXP;
                }
        }

        return sEXP;
}
```

3.13 Remove Right Zeros

The following code is used to removing floating numbers right zeros to help process the user's expression and displays the number without right zeros.

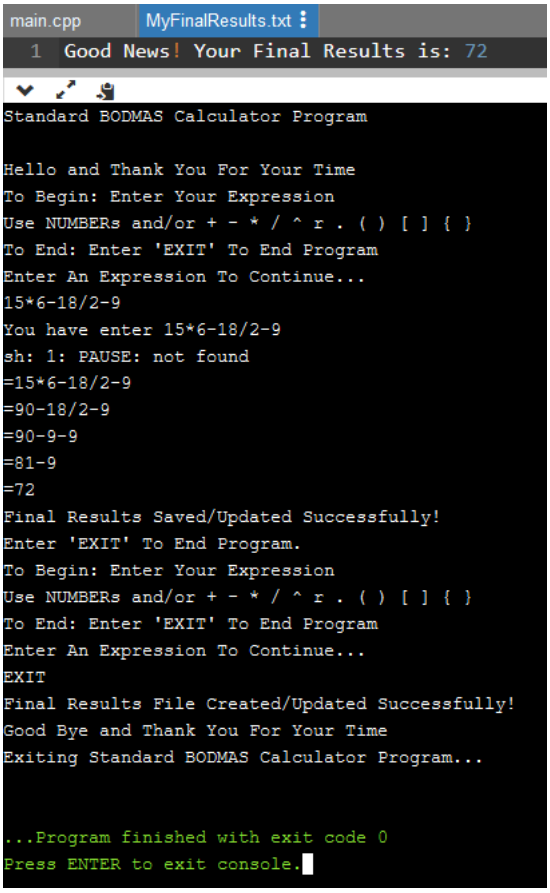
```
std::string remove_right_zeros(std::string sNUMBER) {

        std::string sTerminateDizits = "123456789.";
        std::size_t nFound = 0;

        for (int i = sNUMBER.length(); i > -1; --i)
        {
                nFound = sTerminateDizits.find_first_of(sNUMBER[i]);
                if (nFound != std::string::npos)
                {
                        if (sNUMBER[i] == '.') sNUMBER[i] = '\0';
                        break;
                }
                else
                {
                        if (sNUMBER[i] == '0') sNUMBER[i] = '\0';
                }
        }
        nFound = sNUMBER.find('\0');
        if (nFound != std::string::npos)
        {
                sNUMBER.erase(nFound);
        }
        return sNUMBER;
}
```

4. Execution of Source Code

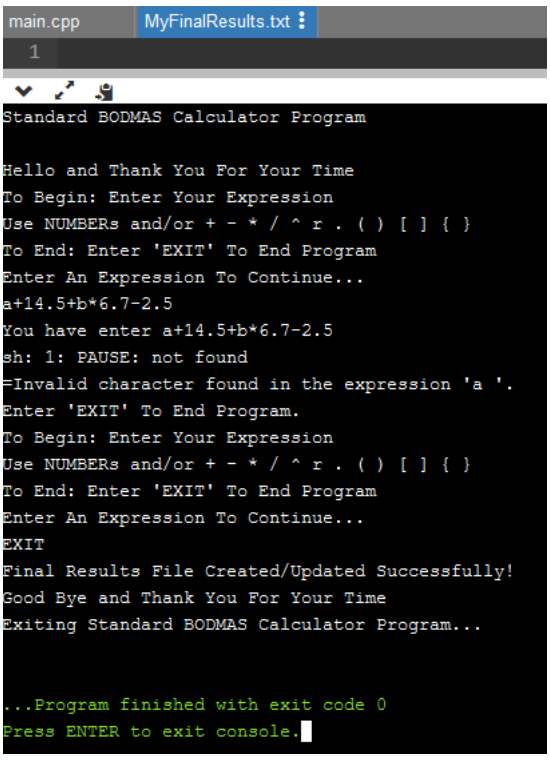
See execution screenshots using the following sample expressions:

- 15*6-18/2-9
- a+14.5+b*6.7-2.5 // invalid expression
- 40*7-[24-{16-(3^2-4+5+64r3+9/-6*2)}]
- 3.5^2.5+64.5r3

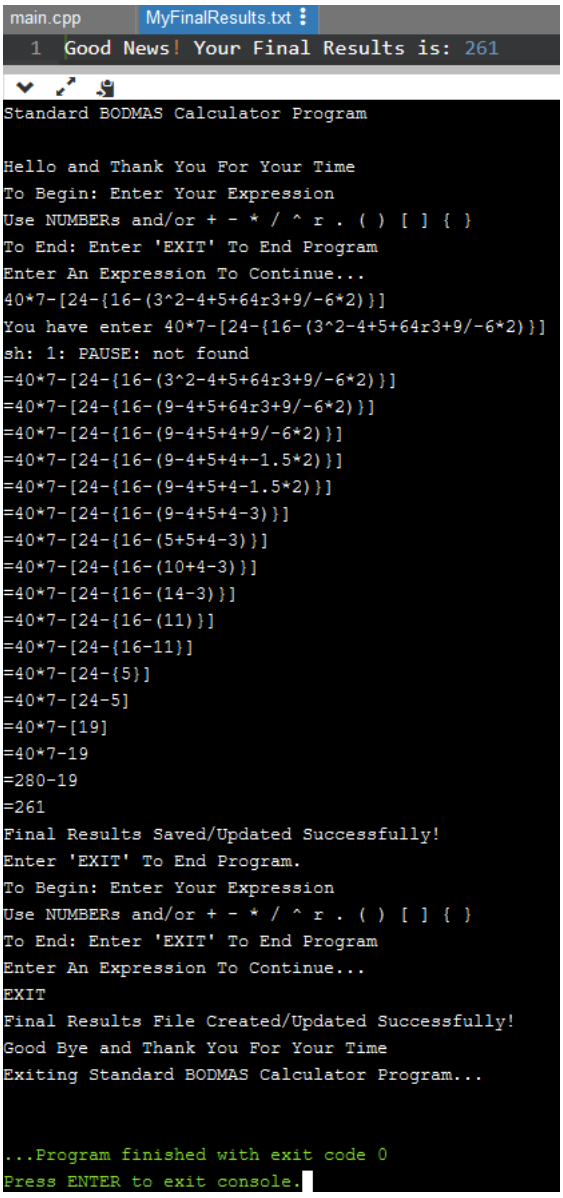
4.1 Execution #1 - 15*6-18/2-9



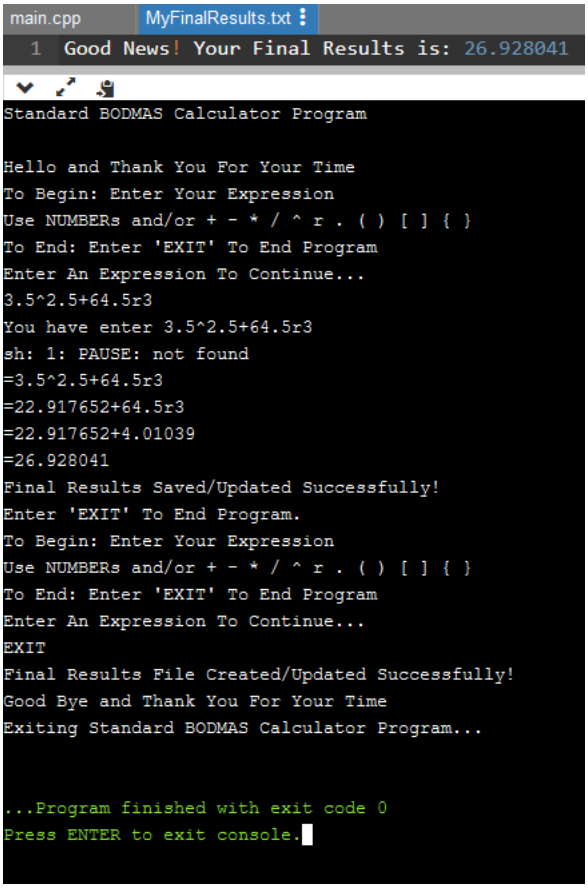
4.2 Execution #2 - a14.5+b*6.7-2.5



4.4 Execution #3 - 40*7-[24-{16-(3^-2-4+5+64r3+9/-3-6*-2)}]



4.5 Execution #4 - 3.5^2.5+64.5r3



References

[**1**] **Wikipedia contributors**. "Order of operations." Wikipedia, The Free Encyclopedia. Wikipedia, The Free Encyclopedia, 18 Apr. 2021. Web. 18 Apr. 2021.

[**2**] **Software Engineering Group**, Reverse Polish Notation, http://seg.ee.upatras.gr/JavaCalculator/Reverse%20Polish%20Notation.htm, Web. 18 Apr. 2021.

[**3**] **Cplusplus.com**, C++ Language, https://www.cplusplus.com/doc/tutorial, Web. 18 Apr. 2021.

[**4**] **Cppsecrets.com**, Implement BODMAS using C++, https://cppsecrets.com/users/1069910011710710110211111410111810111457536410310997105108469911110 9/Implement-BODMAS-using-C00.php, Web. 18 Apr. 2021.

[**5**] **OnlineGDB.com**, Online_C++_compiler, https://www.onlinegdb.com/online_c++_compiler, Web. 18 Apr. 2021.

[**6**] **Github.com**, Research-Programming-Work, https://github.com/jeremyjwsc/Research-Programming-Work, Web. 18 Apr. 2021.