**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра «Информационные Системы и Технологий (ИСиТ)»**

отчет

**по лабораторной работе №3**

**по дисциплине «Программирование»**

Тема: «Стек, прямая и обратная польская нотация»

|  |  |  |
| --- | --- | --- |
| Студент гр. 0324 |  | Зимацкий С. Н. |
| Преподаватель |  | Глущенко А.Г. |

Санкт-Петербург

2021

**Цель работы.**

получение практических навыков работы со стеками и очередями; изучение обратной и прямой польской нотации; проведение сравнительного анализа этих структур данных.

Необходимо написать программу, которая выполняет следующее:

1.   Реализует преобразование введенного выражения (если используются переменные, то пользователь должен их инициализировать). Ввод выражения должен быть реализовать двумя способами: с клавиатуры и \* с файла.

2.   Реализует проверку на корректность простого выражения и выражения, записанного в прямой и обратной польских нотациях (на выбор пользователя). Ввод выражения должен быть реализовать двумя способами: с клавиатуры и \* с файла.

3.   Реализует вычисления простого выражения и выражения, записанного в прямой и обратной польских нотациях (на выбор пользователя). Ввод выражения должен быть реализовать двумя способами: с клавиатуры и с файла.

\*4.   Генерирует несколько (на выбор пользователя) вариантов проверочной работы по польской нотации (прямой и обратной). Задание и ответы к ним необходимо вывести в отдельные файлы (ответы должны быть максимально подробными).

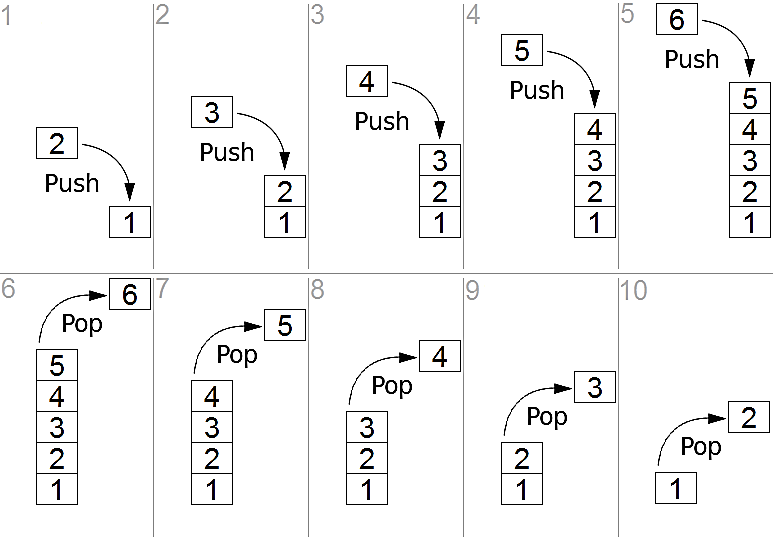
Программа должна выводить и описывать все промежуточные действия.

**Основные теоретические положения.**

## Стек

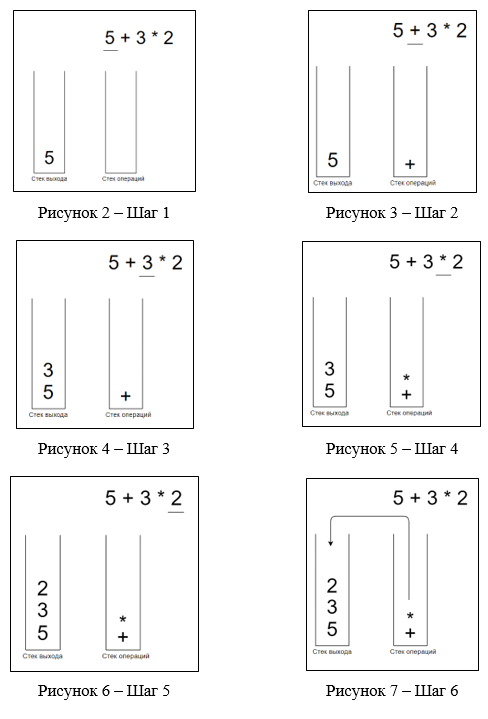
Стек – это частный случай однонаправленного списка, добавление элементов в который и выборка из которого выполняется с одного конца, называемого вершиной стека. Другие операции со стеком не определены. При выборке элемент исключается из стека. Говорят, что стек реализует принцип обслуживания LIFO (последним пришел – первым ушел).

Графически его удобно изобразить в виде вертикального списка, например, стопки книг, где для того чтобы воспользоваться одной из них и не нарушить установленный порядок, нужно поднять все книги, которые лежат выше нее, а положить книгу можно лишь поверх всех остальных.

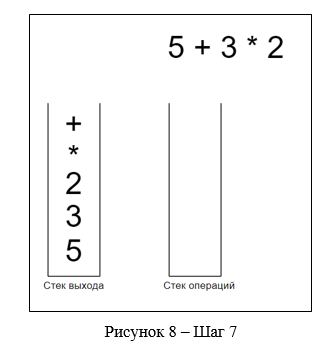


Стек чаще всего реализуется на основе обычных массивов, односвязных и двусвязных списков.

* **Алгоритм польской нотации:**



Финальное представление:



Обрабатывая строку слева направо мы проверяем текущий символ. Если этот символ число, то мы отправляем его в стек выхода. Если этот символ операция, необходимо проверить последний символ в стеке операций. Если стек операций пуст, отправляем символ в стек операций (см. рис. 3). Если в стеке есть операция, необходимо проверить их вес (больший вес должен находится сверху стека, т.к. эта операция более приоритетная). В случае, если вес текущего символа больше последнего, то мы просто отправляем символ в стек операций (см. рис. 5). В противном случае, необходимо удалить из стека операций меньший по весу символ и отправить в стек выхода. Если веса равны, то для корректности выражения нижележащую операцию удалить и отправить в стек выхода.

По окончанию строки, все символы из стека операций по одному отправляются в стек выхода. На этом преобразование закончено.

При прямой польской нотации обработка строки происходит справа налево.

## Обратная польская запись

Обратная польская запись (нотация) – форма записи математических и логических выражений, в которой операнды расположены перед знаками операций. Обратная польская запись имеет ряд преимуществ перед инфиксной записью при выражении алгебраических формул, одно из них то, что инфиксные операторы имеют приоритеты, которые произвольны и нежелательны.

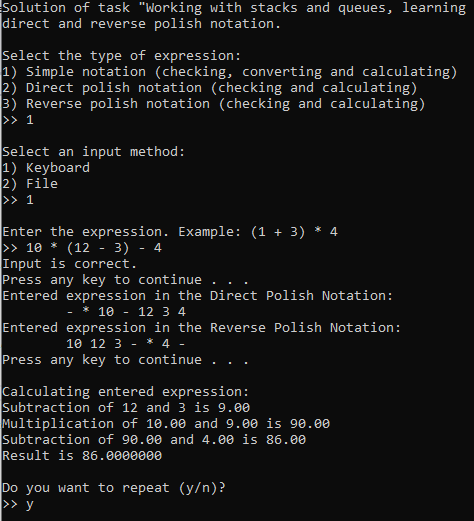
| **Простое выражение** | **Прямая польская запись** | **Обратная польская запись** |
| --- | --- | --- |
| X + 3 \* Y | + X \* 3 Y | X 3 Y \* + |
| (X + 3) \* Y | \* + X 3 Y | X 3 + Y \* |
| 1 + 2 | + 1 2 | 1 2 + |

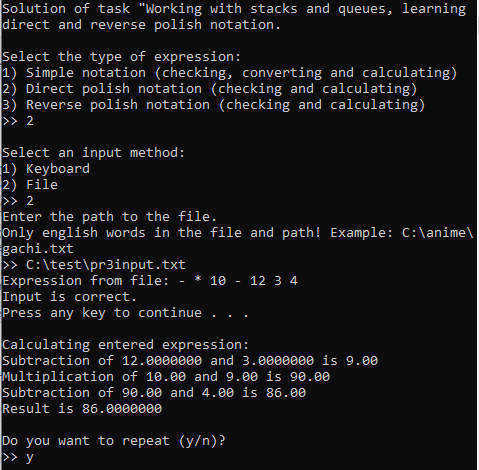
Рассмотрим пример: Обратная польская запись отлично подходит для вычисления выражений при помощи стека. Причем сам алгоритм достаточно прост. Необходимо просто прочитать обратную польскую запись слева направо. Если встречается операнд, то его нужно поместить в стек. Если встречается оператор, нужно выполнить заданную им операцию.

323 324 5 21 - + \*

1. Исходное состояние. Стек пуст.
2. Читаем значение 323. Это - число. Переносим в стек;
3. Читаем значение 324. Это - число. Переносим в стек;
4. Читаем значение 5. Это - число. Переносим в стек;
5. Читаем значение 21. Это - число. Переносим в стек;
6. Читаем значение -. Это оператор. Забираем числа 5 и 21 из стека. Вычитаем. Результат переносим в стек;
7. Читаем значение +. Это оператор. Забираем числа 324 и -16 из стека. Складываем. Результат отправляем в стек.
8. Читаем значение \*. Это оператор. Забираем числа 323 и 308 из стека. Умножаем. Результат отправляем в стек
9. Выражение прочитано. В стеке - одно значение - 631. Это и есть результат вычисления.

**Экспериментальные результаты.**





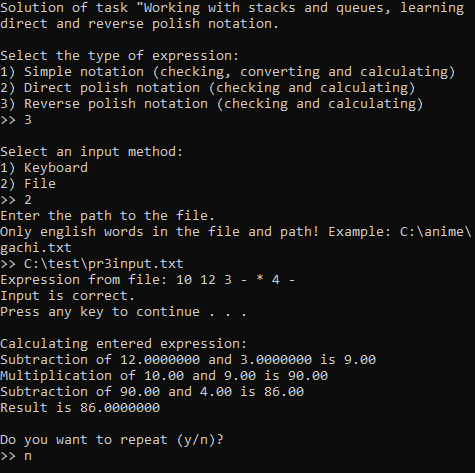


Рисунок 5 – Результат работы программы (полный код программы представлен в приложении А)

Приложение а  
полный код программы

#include <iostream>

#include <string>

#include <fstream>

#include <regex>

#include <iomanip> // for cout setprecision

//#include <Windows.h> // for set coursor coordinates

using namespace std;

typedef struct charStack\* pCharStack;

struct charStack {

char data;

pCharStack next;

};

typedef struct stringStack\* pStringStack;

struct stringStack {

string data;

pStringStack next;

};

typedef struct intStack\* pIntStack;

struct intStack {

int data;

pIntStack next;

};

typedef struct floatStack \* pFloatStack;

struct floatStack {

float data;

pFloatStack next;

};

void push(pCharStack \* stackp, char data);

void pop(pCharStack \* stackp);

int sizeOfStack(const pCharStack \* stackp);

void push(pStringStack \* stackp, string data);

void pop(pStringStack \* stackp);

int sizeOfStack(const pStringStack \* stackp);

void push(pIntStack \* stackp, int data);

void pop(pIntStack \* stackp);

int sizeOfStack(const pIntStack \* stackp);

void push(pFloatStack\* stackp, float data);

void pop(pFloatStack\* stackp);

int sizeOfStack(const pFloatStack \* stackp);

void removeSpacesFromString(string &input);

void expressionPrimaryCheck(string input);

void expressionSecondaryCheck(string input);

bool bracketCheck(string input);

bool polishExpressionCheck(string input);

// polish notations

string simpleToRevert(string input);

string simpleToDirect(string input);

float revertNotationCalculating(string input);

float directNotationCalculating(string input);

short operatorPriority(char input);

bool isOperator(char input);

bool isDigit(char input);

bool choiseNextAction();

void practicalWork3() {

do {

system("CLS");

cout << "Solution of task \"Working with stacks and queues, learning direct and reverse polish notation.\n";

bool inputIsCorrect = false;

short exprType;

string userEntered, input;

do {

// selecting the type of expression

do {

/\*cout << "\nSelect the type of expression:\n"

<< "1) Simple notation\n"

<< "2) Direct polish notation\n"

<< "3) Reverse polish notation\n>> ";\*/

cout << "\nSelect the type of expression:\n"

<< "1) Simple notation (checking, converting and calculating)\n"

<< "2) Direct polish notation (checking and calculating)\n"

<< "3) Reverse polish notation (checking and calculating)\n>> ";

cin >> exprType;

if (exprType == 1 || exprType == 2 || exprType == 3) {

inputIsCorrect = true;

break;

}

else

cout << "Error: wrong input. Please repeat menu selection.\n";

} while (!inputIsCorrect);

// selecting an input method

inputIsCorrect = false;

short inputMethod;

do {

cout << "\nSelect an input method:\n"

<< "1) Keyboard\n"

<< "2) File\n>> ";

cin >> inputMethod;

if (inputMethod == 1) {

inputIsCorrect = true;

break;

}

else if (inputMethod == 2) {

inputIsCorrect = true;

break;

}

else

cout << "Error: wrong input. Please repeat menu selection.\n";

} while (!inputIsCorrect);

// entering the expression (or reading file)

inputIsCorrect = false;

if (inputMethod == 2) {

ifstream ifile;

string path;

cout << "\nEnter the path to the file. \n"

<< "Only english words in the file and path! Example: C:\\anime\\gachi.txt \n>> ";

cin.ignore(32767, '\n');

getline(cin, path);

ifile.open(path);

if (!ifile.is\_open()) {

cout << "Error opening file! Please restart the program! \n";

throw - 1;

}

else {

getline(ifile, userEntered);

ifile.close();

cout << "Expression from file: " << userEntered << endl;

}

}

else {

cin.ignore(32767, '\n');

cout << "\nEnter the expression. Example: ";

switch (exprType) {

case 1:

cout << "(1 + 3) \* 4";

break;

case 2:

cout << "\* + 1 3 4";

break;

case 3:

cout << "1 3 + 4 \*";

break;

}

cout << "\n>> ";

getline(cin, userEntered);

}

// checking the expression

input = userEntered;

try {

switch (exprType) {

case 1:

expressionPrimaryCheck(input);

removeSpacesFromString(input);

expressionSecondaryCheck(input);

break;

case 2:

polishExpressionCheck(input);

break;

case 3:

polishExpressionCheck(input);

break;

}

}

catch (const char\* msg) {

cout << msg;

continue;

}

inputIsCorrect = true;

if (inputIsCorrect) {

cout << "Input is correct.\n";

system("pause");

}

// conversion the expression into other types of notation

string convertedExpression;

switch (exprType) {

case 1:

cout << "Entered expression in the Direct Polish Notation: \n\t"

<< simpleToDirect(input) << endl;

convertedExpression = simpleToRevert(input);

cout << "Entered expression in the Reverse Polish Notation: \n\t"

<< convertedExpression << endl;

system("pause");

break;

case 2:

/\*cout << "Entered expression in the Simple Notation: \n\t"

<< endl;

cout << "Entered expression in the Reverse Polish Notation: \n\t"

<< simpleToRevert(input) << endl;\*/

break;

case 3:

/\*cout << "Entered expression in the Simple Notation: \n\t"

<< endl;

cout << "Entered expression in the Direct Polish Notation: \n\t"

<< endl;\*/

break;

}

cout << "\nCalculating entered expression: \n";

float result;

try {

switch (exprType) {

case 1:

result = revertNotationCalculating(convertedExpression);

break;

case 2:

result = directNotationCalculating(input);

break;

case 3:

result = revertNotationCalculating(input);

break;

}

}

catch (const char\* msg) {

inputIsCorrect = false;

cout << msg;

continue;

}

cout << "Result is ";

//// setting a console pointer

//HANDLE hConsole = GetStdHandle(STD\_OUTPUT\_HANDLE);

//CONSOLE\_SCREEN\_BUFFER\_INFO bi;

//GetConsoleScreenBufferInfo(hConsole, &bi);

//bi.dwCursorPosition.X += 10;

//SetConsoleCursorPosition(hConsole, bi.dwCursorPosition);

cout << fixed << setprecision(std::numeric\_limits<float>::digits10 + 1) << result << endl;

} while (!inputIsCorrect);

} while (choiseNextAction());

}

void removeSpacesFromString(string &input) {

bool isRemoved;

do {

isRemoved = false;

for (unsigned i = 0; i < input.length(); i++) {

if (input[i] == ' ') {

input.erase(i, 1);

isRemoved = true;

}

}

} while (isRemoved);

}

void expressionPrimaryCheck(string input) {

if (input.empty())

throw "Error! Empty input. \nPlease, repeat input.";

//searching incorrect symbols

regex mask("[^0-9\\s\\+\\-\\\*\\/\\(\\)]");

if (std::regex\_search(input, mask))

/\*

Error! Input contains wrong symbols.

Correct symbols: + - \* / ( ) \"space\" 0-9

Please, repeat input

\*/

throw "Error! Input contains wrong symbols. \nCorrect symbols: + - \* / ( ) \"space\" 0-9 \nPlease, repeat input.\n";

//searching incorrect sequence of characters

mask = "[0-9]\\s+[0-9]"; // "11 4" sequences

if (std::regex\_search(input, mask))

/\*

Error! Input contains wrong sequence of characters.

Correct example: (1 \* 4 + 3 \* 4) \* 4

Incorrect sequences example: (1 4 + 3 4) \* 4

Please, repeat input.

\*/

throw "Error! Input contains wrong sequence of characters. \nCorrect example: (1 + 3) \* 4\nIncorrect sequences example: +--\*/++--\*\*// 11 4 (space between numbers and characters)\nPlease, repeat input.\n";

}

void expressionSecondaryCheck(string input) {

//searching incorrect sequence of characters

regex mask("[\\+\\-\\\*\\/]{2}"); // "+--\*/++--\*\*//" sequences

if (std::regex\_search(input, mask))

/\*

Error! Input contains wrong sequence of characters.

Correct example: (1 + 3) \* 4

Incorrect sequences example: +--\* /++--\*\* //

Please, repeat input.

\*/

throw "Error! Input contains wrong sequence of characters. \nCorrect example: (x + 3) \* y\nIncorrect sequences example: +--\*/++--\*\*// \nPlease, repeat input.\n";

mask = "([0-9]\\()|(\\)[0-9])|(\\(\\))"; // "() 5( )5" sequences

if (std::regex\_search(input, mask))

/\*

Error! Input contains wrong sequence of characters.

Correct example: (1 + 3) \* 4

Incorrect sequences example: () 5( )5

Please, repeat input.

\*/

throw "Error! Input contains wrong sequence of characters. \nCorrect example: (1 + 3) \* 4\nIncorrect sequences example: () 5( )5 \nPlease, repeat input.\n";

//searching the operators in begin and end of the expression

mask = "^[\\+\\\*\\/]|[\\+\\-\\\*\\/]$";

if (std::regex\_search(input, mask))

/\*

Error! Input contains operators in begin or end of the expression.

Correct example: 1 + 3 \* 4

Incorrect example: \*1 + 3 \* 4-

Please, repeat input.

\*/

throw "Error! Input contains operators in begin or end of the expression. \nCorrect example: 1 + 3 \* 4\nIncorrect example: \*1 + 3 \* 4- \nPlease, repeat input.\n";

// searching only operators

mask = "^[\\s\\+\\-\\\*\\/\\(\\)]+$";

if (std::regex\_search(input, mask))

/\*

Error! Input contains only operators. Input must contains operands too.

Correct example: 1 + 3 \* 4

Incorrect example: ( \* )

Please, repeat input.

\*/

throw "Error! Input contains only operators. Input must contains operands too. \nCorrect example: 1 + 3 \* 4\nIncorrect example: ( \* ) \nPlease, repeat input.\n";

//searching only numbers

mask = "^[\\s0-9]+$";

if (std::regex\_search(input, mask))

/\*

Error! Input contains only operands. Input must contains operators too.

Correct example: 1 + 3 \* 4

Incorrect example: 23 31

Please, repeat input.

\*/

throw "Error! Input contains only operands. Input must contains operators too. \nCorrect example: 1 + 3 \* 4\nIncorrect example: 23 31 \nPlease, repeat input.\n";

//counting opened and closed brackets

if (!bracketCheck(input))

/\*

Error! The number of opening brackets does not match the number of closing brackets.

Correct example: (1 + (3 - 1)) \* 4

Incorrect example: (1 + (3 - 4) \* 4

Please, repeat input.

\*/

throw "Error! The number of opening brackets does not match the number of closing brackets. \nCorrect example: (1 + (3 - 1)) \* 4\nIncorrect example: (1 + (3 - 4) \* 4 \nPlease, repeat input.\n";

}

bool bracketCheck(string input) {

int openedBracket = 0, closedBracket = 0;

for (unsigned i = 0; i < input.length(); i++) {

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

openedBracket++;

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

closedBracket++;

}

if (openedBracket == closedBracket)

return true;

else

return false;

}

bool polishExpressionCheck(string input) {

if (input.empty())

throw "Error! Empty input. \nPlease, repeat input.";

//searching incorrect symbols

regex mask("[^0-9\\s\\+\\-\\\*\\/]");

if (std::regex\_search(input, mask))

/\*

Error! Input contains wrong symbols.

Correct symbols: + - \* / \"space\" 0-9

Brackets are incorrect!

Please, repeat input

\*/

throw "Error! Input contains wrong symbols. \nCorrect symbols: + - \* / \"space\" 0-9 \nBrackets are incorrect! \nPlease, repeat input.\n";

// searching only operators

mask = "^[\\s\\+\\-\\\*\\/]+$";

if (std::regex\_search(input, mask))

/\*

Error! Input contains only operators. Input must contains operands too.

Correct example: 1 + 3 \* 4

Incorrect example: ( \* )

Please, repeat input.

\*/

throw "Error! Input contains only operators. Input must contains operands too. \nCorrect example: 1 + 3 \* 4\nIncorrect example: ( \* ) \nPlease, repeat input.\n";

//searching only numbers

mask = "^[\\s0-9]+$";

if (std::regex\_search(input, mask))

/\*

Error! Input contains only operands. Input must contains operators too.

Correct example: 1 + 3 \* 4

Incorrect example: 23 31

Please, repeat input.

\*/

throw "Error! Input contains only operands. Input must contains operators too. \nCorrect example: 1 + 3 \* 4\nIncorrect example: 23 31 \nPlease, repeat input.\n";

//counting operators and operands

int operators = 0, operands = 0, digits = 0;

for (int i = 0; i < input.size(); i++) {

if (isDigit(input[i])) {

digits++;

continue;

}

if (input[i] == ' ')

if (digits > 0) {

operands++;

digits = 0;

}

if (isOperator(input[i])) {

//if (input[i] != '-')

operators++;

//counting operands

if (digits > 0) {

operands++;

digits = 0;

}

continue;

}

}

if (operators > operands)

/\*

Error! Input contains operators more than operands!

This program not working correctly with negative numbers and operators besause i don't get paid for this job.

\*/

throw "Error! Input contains operators more than operands! \nThis program not working correctly with negative numbers and operators besause i don't get paid for this job.";

}

string simpleToRevert(string input) {

string result;

pCharStack stackP = NULL;

for (int i = 0; i < input.length(); i++) {

if (input[i] == '(') {

push(&stackP, input[i]);

continue;

}

while (!isOperator(input[i])) {

result += input[i];

i++;

if (i >= input.length()) {

result += ' ';

break;

}

if (isOperator(input[i])) {

result += ' ';

//i--;

break;

}

}

if (i >= input.length()) break;

if (input[i] == ')') {

char pos = ' ';

while (pos != '(') {

pos = stackP->data;

pop(&stackP);

if (pos != '(') {

result += pos;

result += ' ';

}

}

continue;

}

if (sizeOfStack(&stackP) != 0)

while (operatorPriority(stackP->data) >= operatorPriority(input[i])) {

result += stackP->data;

result += ' ';

pop(&stackP);

if (sizeOfStack(&stackP) == 0)

break;

}

if (sizeOfStack(&stackP) == 0) {

push(&stackP, input[i]);

continue;

}

if (operatorPriority(stackP->data) < operatorPriority(input[i])) {

push(&stackP, input[i]);

continue;

}

}

while (sizeOfStack(&stackP) != 0) {

result += stackP->data;

result += ' ';

pop(&stackP);

}

// delete last ' ' symbol

result.erase(result.length() - 1);

return result;

}

string simpleToDirect(string input) {

string result = input;

int size = input.size();

//flip the string

char \* temp = new char[size + 1];

for (int i = 0, j = size - 1; i < size, j >= 0; i++, j--) {

temp[i] = input[j];

}

temp[size] = '\0';

result = temp;

//brackets flipping

for (int i = 0; i < size; i++) {

if (result[i] == '(' || result[i] == ')') {

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

result[i] = ')';

else

result[i] = '(';

i++;

}

}

result = simpleToRevert(result);

size = result.size();

delete temp;

temp = new char[size + 1];

//flip the string again

for (int i = 0, j = size - 1; i < size, j >= 0; i++, j--) {

temp[i] = result[j];

}

temp[size] = '\0';

result = temp;

delete temp;

return result;

}

float revertNotationCalculating(string input) {

pFloatStack stackP = NULL;

for (int i = 0; i < input.length(); i++) {

if (input[i] == ' ') continue;

if (isDigit(input[i])) {

//parsing the number to tmp and then converting to int

string tmp;

do {

tmp += input[i];

i++;

if (i >= input.length()) break;

if (input[i] == ' ') break;

} while (isDigit(input[i]));

float number = stof(tmp);

push(&stackP, number);

}

else {

float secondNumber = stackP->data;

pop(&stackP);

//checking to existing next number

if (!(stackP->data)) throw "Error! Excess number in the expression.";

float firstNumber = stackP->data;

pop(&stackP);

//calculating and printing

float operationResult;

switch (input[i]) {

case '+':

operationResult = firstNumber + secondNumber;

cout << "Sum of ";

break;

case '-':

operationResult = firstNumber - secondNumber;

cout << "Subtraction of ";

break;

case '\*':

operationResult = firstNumber \* secondNumber;

cout << "Multiplication of ";

break;

case '/':

operationResult = firstNumber / secondNumber;

cout << "Divide of ";

break;

}

push(&stackP, operationResult);

cout << firstNumber << " and " << secondNumber << " is " << fixed << setprecision(2) << stackP->data << endl;

}

}

return stackP->data;

}

float directNotationCalculating(string input) {

pFloatStack stackP = NULL;

int size = input.size();

for (int i = size - 1; i >= 0; i--) {

if (input[i] == ' ') continue;

if (isDigit(input[i])) {

//parsing the number to tmp and then converting to int

string inverseTmpNumber;

do {

inverseTmpNumber += input[i];

i--;

if (i >= input.length()) break;

if (input[i] == ' ') break;

} while (isDigit(input[i]));

// flipping the number back

char \* tmpNumber = new char[inverseTmpNumber.size() + 1];

for (int k = 0, j = inverseTmpNumber.size() - 1; k < size, j >= 0; k++, j--) {

tmpNumber[k] = inverseTmpNumber[j];

}

tmpNumber[size] = '\0';

float number = stof(tmpNumber);

push(&stackP, number);

}

else {

float firstNumber = stackP->data;

pop(&stackP);

//checking to existing next number

if (!(stackP->data)) throw "Error! Excess number in the expression.";

float secondNumber = stackP->data;

pop(&stackP);

//calculating and printing

float operationResult;

switch (input[i]) {

case '+':

operationResult = firstNumber + secondNumber;

cout << "Sum of ";

break;

case '-':

operationResult = firstNumber - secondNumber;

cout << "Subtraction of ";

break;

case '\*':

operationResult = firstNumber \* secondNumber;

cout << "Multiplication of ";

break;

case '/':

operationResult = firstNumber / secondNumber;

cout << "Divide of ";

break;

}

push(&stackP, operationResult);

cout << firstNumber << " and " << secondNumber << " is " << fixed << setprecision(2) << stackP->data << endl;

}

}

return stackP->data;

}

short operatorPriority(char input) {

switch (input) {

case '(':

return 1;

case '+': case '-':

return 2;

case '\*': case '/':

return 3;

default:

return 0;

}

}

bool isOperator(char input) {

if (input == '+'

|| input == '-'

|| input == '\*'

|| input == '/'

|| input == ')'

|| input == '(')

return true;

else

return false;

}