Министерство образования и науки Российской Федерации

Федеральное государственное автономное образовательное   
учреждение высшего образования

Национальный исследовательский Нижегородский государственный университет им. Н.И. Лобачевского

Институт информационных технологий, математики и механики

**Отчет по лабораторной работе**

**«Вычисление арифметических выражений»**

**Выполнила**:

студентка группы 3822Б1ПМ1

Шарамыгина И.В.

**Проверил**:

преподаватель каф. МОСТ,

Волокитин В.Д.

Нижний Новгород

2023

Содержание

[Введение 3](#_Toc529541653)

[1. Постановка задачи 4](#_Toc529541654)

[2. Руководство пользователя 5](#_Toc529541655)

[3.1. Описание структуры программы 6](#_Toc529541656)

[3.2. Описание алгоритмов 6](#_Toc529541657)

[4. Результаты экспериментов 7](#_Toc529541658)

[Заключение 8](#_Toc529541659)

[Литература 9](#_Toc529541660)

[Приложение 10](#_Toc529541661)

# Введение

При расчетах арифметических выражений одна из часто встречаемых ошибок – ошибка в скобках. Чтобы избежать данную проблему, а также ускорить вычисления, были придуманы польская и обратная польская формы записи числа.

Отличительной особенностью обратной польской нотации является то, что все аргументы (или операнды) расположены перед знаком операции. В общем виде запись выглядит следующим образом:

Запись набора операций состоит из последовательности операндов и знаков операций. Операнды в выражении при письменной записи разделяются пробелами.

Выражение читается слева направо. Когда в выражении встречается знак операции, выполняется соответствующая операция над двумя последними встретившимися перед ним операндами в порядке их записи. Результат операции заменяет в выражении последовательность её операндов и её знак, после чего выражение вычисляется дальше по тому же правилу.

Результатом вычисления выражения становится результат последней вычисленной операции.

# Постановка задачи

Разработать пользовательское приложение – калькулятор для вычисления выражений с помощью обратной польской нотации. Выражения могут содержать константы и переменные. Для одного выражения реализовать возможность многократного вычисления с разными значениями переменных.

Для решения данной задачи реализовать:

- стек данных;

- алгоритм перевода в обратную польскую запись и вычисления выражений;

- пользовательское приложение;

- тесты для стека на основе Google Tests;

- тесты для алгоритмов перевода и вычисления на основе Google Tests;

# Руководство пользователя

1. При запуске приложения пользователю описываются правила ввода выражений и предлагается задать выражение для вычислений (см. рис. 1).

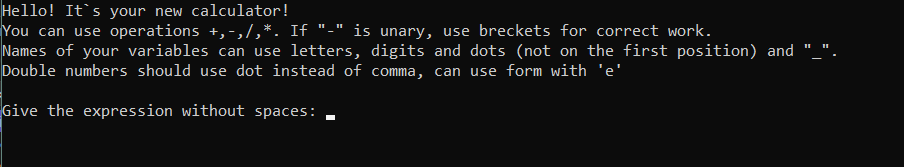


Рисунок 1. Руководство по введению выражений

1. При введении неверного выражение, пользователю будет выдана информация об ошибке, приведена лексема и комментарий, а также предложено повторное введение выражения (см. рис.2).

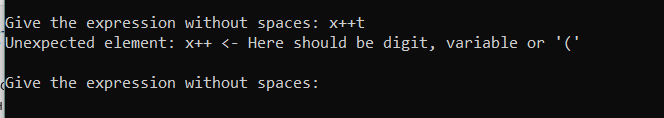


Рисунок 2. Вывод при введении неверного выражения

1. После введения корректного выражения пользователю будет предложено произвести вычисления, ввести новое выражение или выйти из программы (см. рис. 3). Необходимо указать цифру, соответствующую выбору.

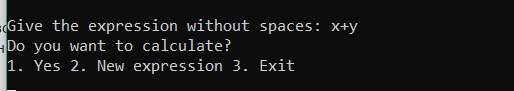


Рисунок 3. Вывод при вводе корректного выражения

1. При выборе произведение расчетов пользователю будет предложено ввести значение для каждой из переменных последовательно (см. рис.4), а затем показан результат. Вместе с результатом будет выведено повторное предложение о произведении расчетов, нового выражения или выходе из программы.

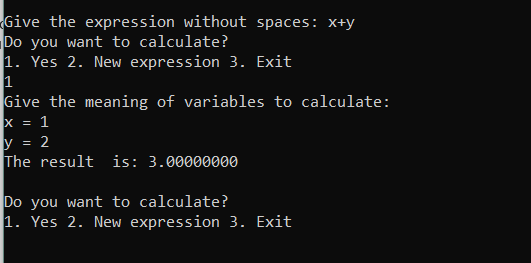


Рисунок 4. Произведение расчетов

1. При выборе ввода нового выражения появится соответствующая строка (см. рис. 5), при выборе выхода программа будет закрыта.

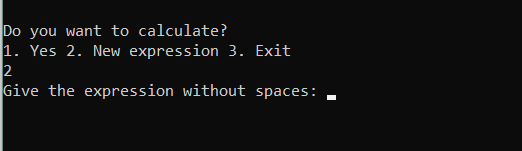


Рисунок 5. Ввод нового выражения

1. **Руководство программиста**

## Описание структуры программы

Программа содержит два класса TDynamicStack и Arithmetic.

**Класс TDynamicStack**

Шаблонный класс стек, private поля T\* data, size\_t index = 0 для хранения индекса текущего элемента и size\_t capacity для хранения размера стека. Для работы со стеком реализованы следующие методы:

void resize() – перевыделяет память, вызвается функцией push, если ind равен capacity.

void clear() – необходим для очистки stack(устанавливается индекс = 0, перевыделяется память, если очищаемый стек был больше размера по умолчанию)

bool isEmpty() – Проверка на пустоту. Вернет true, если стек пуст. Иначе вернет false.

void push(const T& elem) – добавляет элемент на вершину стека. Перевыделяется память, если стек полный

T pop() – извлекает верхний элемент из стека, с доступом к нему.

size\_t size() - возвращает количества элементов в стеке.

T top() - просмотривает первый элемент в стеке.

**Класс Arithmetic**

Класс содержит следующие закрытые поля: string infix – для хранения инфиксной формы, string postfix – для хранения обратной польской формы, string operands – строка содержащая все переменные без повторений через пробел, string\* variables – массив переменных, string\* SetOfLexems – массив лексем, int lexemSize – количество лексем, размер массива лексем и int variableSize – количество переменных, размер массива переменных. Для работы реализованы следующие методы:

void setInfix(string& str) – в созданном объекте меняет инфикс и остальные поля в соответствии с ним.

string getInfix() const – возвращает строку с инфиксной формой.

string getPostfix() const – возвращает строку с обратной польской записью.

string getOperands() const – возвращает строку со всеми переменными через пробел.

bool isOperation(const char& x) – возвращает является ли символ операцией (+,-,\*,/).

int priority(const char& operation) – возвращает приоритеты операций (+,-:1; \*,/:2; ~:3; иначе 0)

void makePostfix() – преобразует инфиксную форму в постфиксную.

double makeDouble(string number) – преобразует строку в число.

void Parser() – разбивает инфиксную форму на лексемы.

void infixCheck() – проверяет инфикс на наличие ошибок, бросает исключения при наличии таковых.

bool isOperand(const char& first\_symbol) – возращает True, если символ является буквой латинского алфавита или нижним подчеркиванием.

bool isDigit(const char& first\_symbol) – возращает True,если символ является цифрой или точкой.

double calculate() – запрашивает значения переменных и производит вычисления, возвращает результат.

## Описание алгоритмов

* + 1. **Проверка строки, выражения в инфиксной форме**
* Заводиться строка, накапливающая ошибку.
* Отдельно проверяется 1 символ на корректность.
* Далее начиная с 1 и до предпоследнего символа циклом проходимся по строке
* Если встречаем скобки проверяем, стоят ли рядом арифметические операции, а также вычитаем для закрывающей и прибавляем для открывающей скобки для случаев, когда закрывающие поставят раньше, чем открывающие.
* Если встречаем знаки, проверяем, чтобы не стояло 2 знака подряд.
* Если встречаем символ (букву или нижнее подчеркивание), создаем строку и накапливаем имя, пока не встречаем скобку или операцию, затем проходимся по ней, проверяя корректность каждого символа.
* Если встречаем цифру, также накапливаем число в строке до того, как найдем скобку или операцию, затем проходимся по строке. Определяя количество экспонент и точек, а также их порядок, отсеиваем неправильные числа, затем проходимся и также проверяем каждый символ на корректность.
* Во всех случаях, встречая ошибку, которая накапливается в созданной изначально строке приписываем стрелочку и комментарий к ошибке. В случае с числами и именами переменных ошибка будет выведена не после конкретного символа, но после лексемы с ошибкой.
* Отдельно проверяем последний символ на корректность. Также проверяем деление на 0.
* Проходимся по циклу считая количество скобок и если они неравны в конце строки кидаем сообщение о том, что закрывающих не хватает (лишнее закрывающие обрабатываются в основном цикле до).
  + 1. **Преобразование строки в массив лексем**
* Заводим строку под имя, счетчик лексем и счетчик имен.
* Проходим по строке инфиксной формы.
* Если встречаем строку или операцию добавляем в массив лексем, меняя счетчик, отдельно обрабатываем унарный минус заменяя обычный знак минуса на тильду.
* Встречая букву, накапливаем имя переменной одновременно в массиве лексем и строке имени, проверяем, есть ли она в строке операндов и добавляем, если нет, меняем значение счетчиков.
* Встречая число, накапливаем его в массиве лексем.
* Итоговое количество переменных равно итоговому количеству имен, добавленных в строку операндов.
  + 1. **Преобразование массива лексем в постфиксную форму.**
* Создаем стек строк. Идем по массиву лексем.
* По первому символу проверяем, что перед нами
* Если встречаем открывающую скобку, кладем в стек. Закрывающую – выталкиваем в постфиксную форму все операции из стека через пробел.
* Если встречаем число или переменную – записываем в строку постфикса через пробел.
* Если встречаем операцию, проверяем ее приоритет и выталкиваем в строку все выражение одного с ней приоритета и выше через пробел.
* Унарный минус обрабатывается отдельно, так как обладает максимальным приоритетом, помещается в стек.
  + 1. **Вычисление выражения.**
* Создаем массив для значений переменных.
* С помощью цикла запрашиваем значения каждой из переменных.
* Создаем стек типа double для накапливания результата.
* Идем по строке постфикс, от пробела до пробела накапливаем значение.
* Если встречаем переменную, находим ее индекс в массиве переменных и добавляем в стек значение из созданного в начале массива с тем же индексом.
* Если встречаем константу, добавляем ее в стек.
* Если встречаем операцию берем два последних значение, где первое будет правым операндом для операции. Результат возвращаем в стек.
* Выполняя деление, делаем проверку деления на 0.
* Отдельно обрабатываем унарный минус, в связи с отсутствием левого операнда.
* Удаляем массив значений, возвращаем результат.
  + 1. **Преобразование строки в число.**
* Создаем переменные для хранения результата и знака.
* Если в строке есть точка и нет экспоненты, находим позицию точки. Проходя массивом по числу, собираем строки значений до точки и после. Вызываем функцию для полученных строк. Значение строки после точки прибавляем, выполняя деление на 10 в количестве раз по длине строки.
* Если встречаем экспоненту, аналогично накапливаем значения до и после нее, вызывая функцию к каждой из накопленных строк, затем возводим часть до экспоненты в нужную степень, учитывая знак.
* Если нет ни экспоненты, ни точки, приводим число посимвольно, умножая на 10 и накапливая результат. Также учитываем знак.
* Возвращаем полученный результат.

# Результаты экспериментов

Приложение может считать выражения с унарным минусом (см. рис. 6), принимать числа с плавающей запятой с записью через экспоненту (см. рис. 7), принимать различные значения переменных для одного выражения и показывать результат (см. рис. 8), принимать выражения с длинными названиями переменных(см. рис. 9), а также не запрашивать повторно значение переменной в случае ее многократного повторения в коде (см. рис. 10).

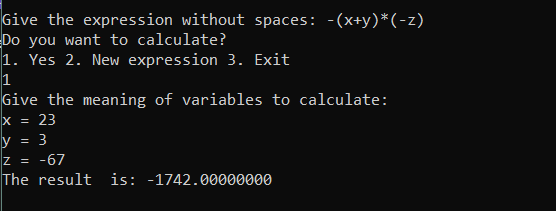


Рисунок 6. Вычисление выражений с унарным минусом

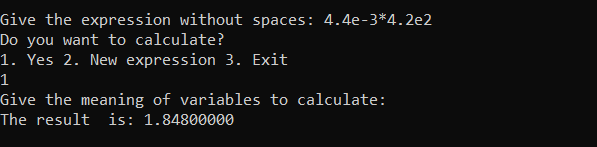


Рисунок 7. Вычисление выражений с числами с плавающей запятой

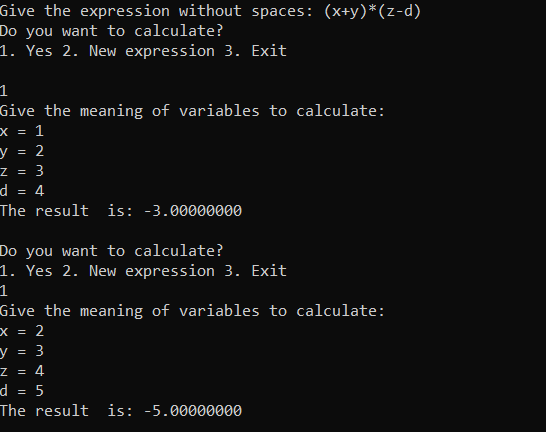


Рисунок 8. Множественное вычисления результата для разных значений переменных

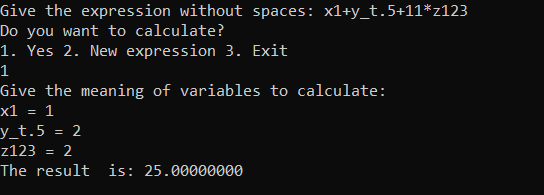


Рисунок 9. Выражение с разнообразными значениями переменных

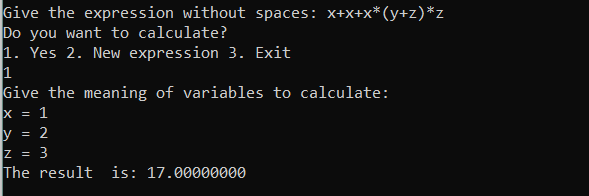


Рисунок 10. Выражение с многократно повторяющимися переменными

# Заключение

Результатом данной работы стало приложение для вычисления разнообразных арифметических выражений с помощью постфиксной (обратной польской) формы записи. Были разработаны класс динамического стека и класс арифметика, а также алгоритмы перевода строк в числа с плавающей запятой, проверке исходного выражения, перевода в лексемы, постфиксную запись, а также вычисление выражений.

Результаты экспериментов показали корректную работу приложения с различными выражениями.

# Приложение

double Arithmetic::makeDouble(string str)

{

double res = 0.0, sign = 1.0;

if (str.find('.') != -1 && str.find('e') == -1) {//there is dot

double beforepoint = 0, afterpoint = 0;

string beforepoint\_s, afterpoint\_s;

size\_t p = str.find('.');//position of the dot

for (size\_t i = 0; i < p; i++) { beforepoint\_s += str[i]; }

for (size\_t i = p + 1; i < str.size(); i++) { afterpoint\_s += str[i]; }

beforepoint = makeDouble(beforepoint\_s);

afterpoint = makeDouble(afterpoint\_s);

afterpoint /= pow(10,afterpoint\_s.size());

res = beforepoint + afterpoint;

}

else if (str.find('e') !=-1) {//there is e

double beforeE = 0.0, afterE = 0.0;

string beforeE\_s, afterE\_s;

size\_t p = str.find('e');

for (size\_t i = 0; i < p; i++) { beforeE\_s += str[i]; }

for (size\_t i = p + 1; i < str.size(); i++) { afterE\_s += str[i]; }

beforeE = makeDouble(beforeE\_s);

afterE = makeDouble(afterE\_s);

if (afterE > 0) res = beforeE \* pow(10.0, afterE);

else {

res = beforeE;

for (size\_t i = 0; i < (-1.0 \* afterE); i++) { res /= 10; }

}

}

else { //no dot, no e

if (str[0] == '~' || str[0] == '-') {

for (size\_t i = 1; i < str.size(); i++) {

res = res \* 10.0 + static\_cast<double>(str[i] - '0');

}

res \*= -1.0;

}

else {

for (size\_t i = 0; i < str.size(); i++) {

res = res \* 10.0 + static\_cast<double>(str[i] - '0');

}

}

}

return res;

}

double Arithmetic::calculate()

{

double\* variableMeanings = new double[variableSize];

cout << "Give the meaning of variables to calculate:" << endl;

for (int i = 0; i < variableSize; i++)

{

string str;

cout << variables[i] << " = ";

cin >> str;

variableMeanings[i]=makeDouble(str);

}

TDynamicStack<double> st;

double eps = pow(10, -20);

for (size\_t i = 0; i < postfix.size(); i++)

{

string name;

while (postfix[i] != ' ')

name.push\_back(postfix[i++]);

if (isOperand(name[0]))//is variable

{

int ind = -1;

int j = 0;

while (ind < 0)

if (variables[j] == name)

ind = j++;

else j++;

st.push(variableMeanings[ind]);

}

if (isOperation(name[0]))

{

double val1, val2;

val2 = st.pop();

val1 = st.pop();

switch (name[0])

{

case '+':

st.push(val1 + val2);

break;

case '-':

st.push(val1 - val2);

break;

case '\*':

st.push(val1 \* val2);

break;

case '/':

if (abs(val2) < eps)

throw runtime\_error ("There is division by zero, it isn't possible");

else

st.push(val1 / val2);

break;

}

}

if (name[0] == '~')

{

double val;

val = st.pop();

st.push(val \* (-1));

}

if (isDigit(name[0]))//is a number

{

double val = makeDouble(name);

st.push(val);

}

}

delete[] variableMeanings;

return st.pop();

}

void Arithmetic::makePostfix()

{

TDynamicStack<string> st;

for (int i = 0; i < lexemSize; i++)

{

switch (SetOfLexems[i][0]) {

case '(':

{

st.push(SetOfLexems[i]);

break;

}

case ')':

{

while (st.top()[0] != '(')

postfix += st.pop() + " ";

st.pop();

break;

}

case '+':case '-':

{

while ((!st.isEmpty()) && (priority(st.top()[0]) > 0))

postfix += st.pop() + " ";

st.push(SetOfLexems[i]);

break;

}

case '\*': case'/':

{

while ((!st.isEmpty()) && (priority(st.top()[0]) > 1))

postfix += st.pop() + " ";

st.push(SetOfLexems[i]);

break;

}

case '~':

{

st.push(SetOfLexems[i]);

break;

}

default:

{

postfix += SetOfLexems[i] + " ";

}

}

}

while(!(st.isEmpty()))

postfix += st.pop()+ " ";

}