Федеральное государственное автономное образовательное учреждение  
высшего образования  
«Нижегородский государственный университет  
им. Н.И. Лобачевского»

Институт информационных технологий, математики и механики (ИИТММ)

Отчёт по лабораторной работе

**Вычисление арифметических выражений**

Выполнил:  
 студент ИИТММ гр. 0823-1  
 Красикова Е.А.

Проверил:

к.т.н., ассистент каф. ПрИнж  
 ИИТММ

Сиднев А.А.

Нижний Новгород  
2016г.

Содержание

[Введение 3](#_Toc470123956)

[Постановка учебно-практической задачи 3](#_Toc470123957)

[Руководство пользователя 3](#_Toc470123958)

[Руководство программиста 4](#_Toc470123959)

[Общее описание структуры программного комплекса. 4](#_Toc470123960)

[Описание структур данных. 4](#_Toc470123961)

[Описание алгоритмов. 5](#_Toc470123962)

[Алгоритм преобразования выражения из инфиксной формы в ОПН 5](#_Toc470123963)

[Алгоритм вычисления выражения, представленного в ОПН 6](#_Toc470123964)

[Заключение 7](#_Toc470123965)

[Список литературы 7](#_Toc470123966)

[Приложение 1. Arithm.h 7](#_Toc470123967)

[Приложение 2. Arithm.cpp 8](#_Toc470123968)

[Приложение 3. test.cpp 13](#_Toc470123969)

# Введение

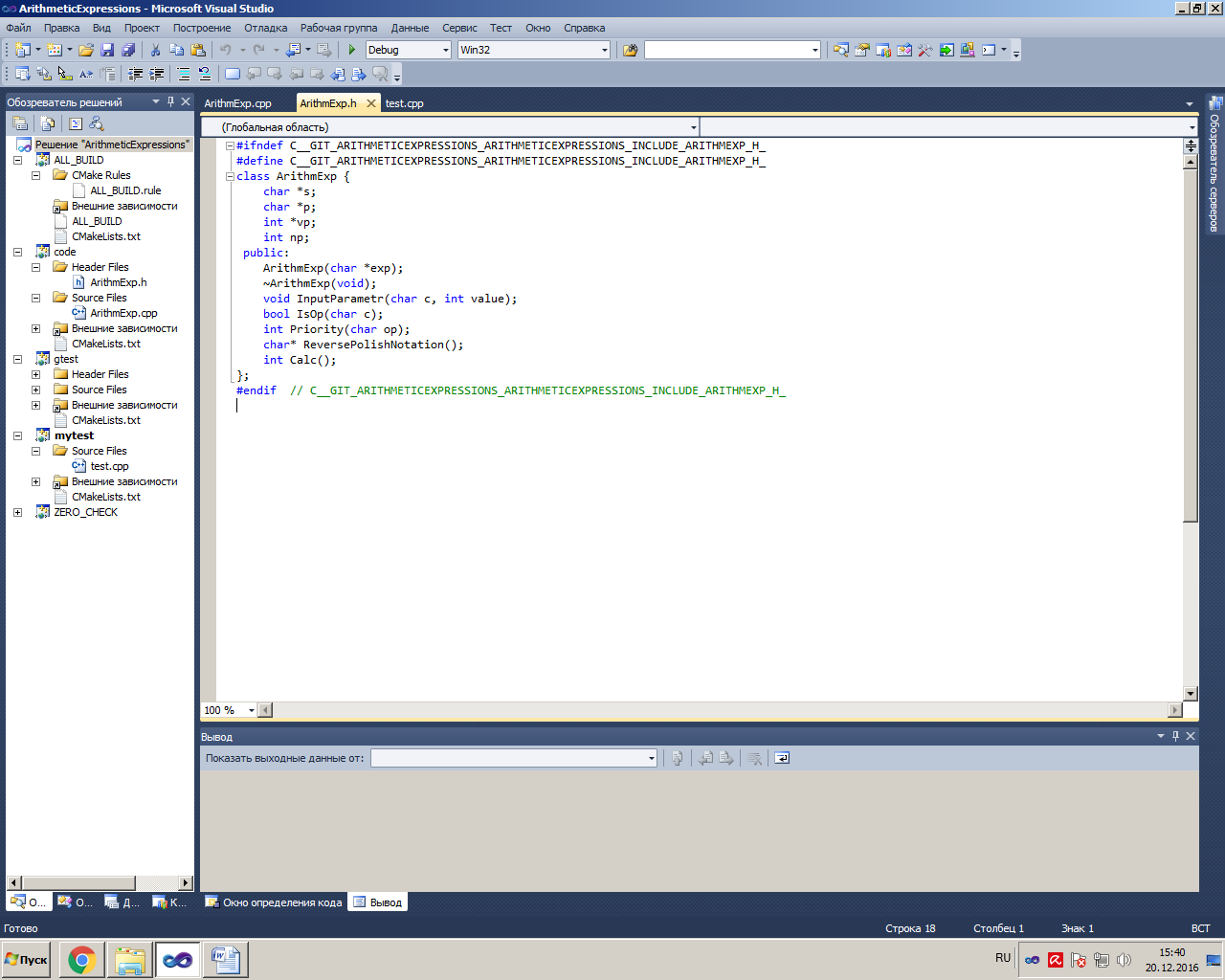
Одной из главных пpичин, лежащих в основе появления языков пpогpаммиpования высокого уpовня, явились вычислительные задачи, требующие больших объёмов рутинных вычислений. Поэтому к языкам пpогpаммиpования предъявлялись требования максимального приближения фоpмы записи вычислений к естественному языку математики. В этой связи одной из первых областей системного пpогpаммиpования сфоpмиpовалось исследование способов выpажений. В данной лабораторной работе будет исследован метод тpансляции с помощью обpатной польской записи , котоpую пpедложил польский математик Я. Лукашевич.

# Постановка учебно-практической задачи

Необходимо разработать класс для разбора и   
вычисления арифметических выражений. Выражение может содержать скобки,   
операции +, -, \*, /, ^ (возведение в степень), || (модуль), константы и   
символьные переменные (строчные буквы латинского алфавита). Необходимо   
реализовать класс, который принимает на вход строку, содержащую выражение,   
выполняет её разбор, выводит сообщение об ошибке при её обнаружении,   
выполняет вычисление значения выражения при заданных значениях переменных.   
Разбор и хранение выражения должны осуществляться в обратной польской   
записи. Реализовать тесты, содержащие различные типы выражений.

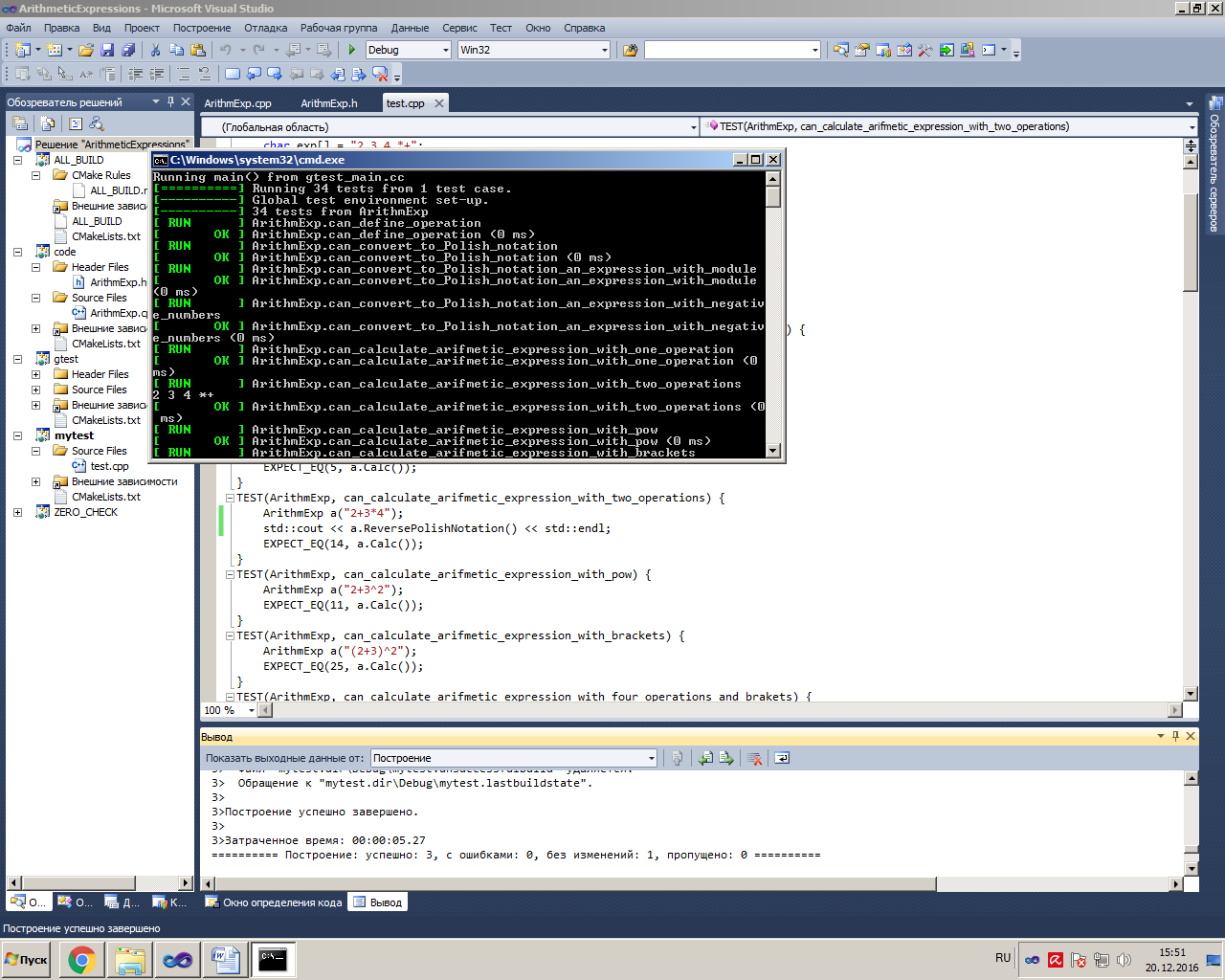
# Руководство пользователя

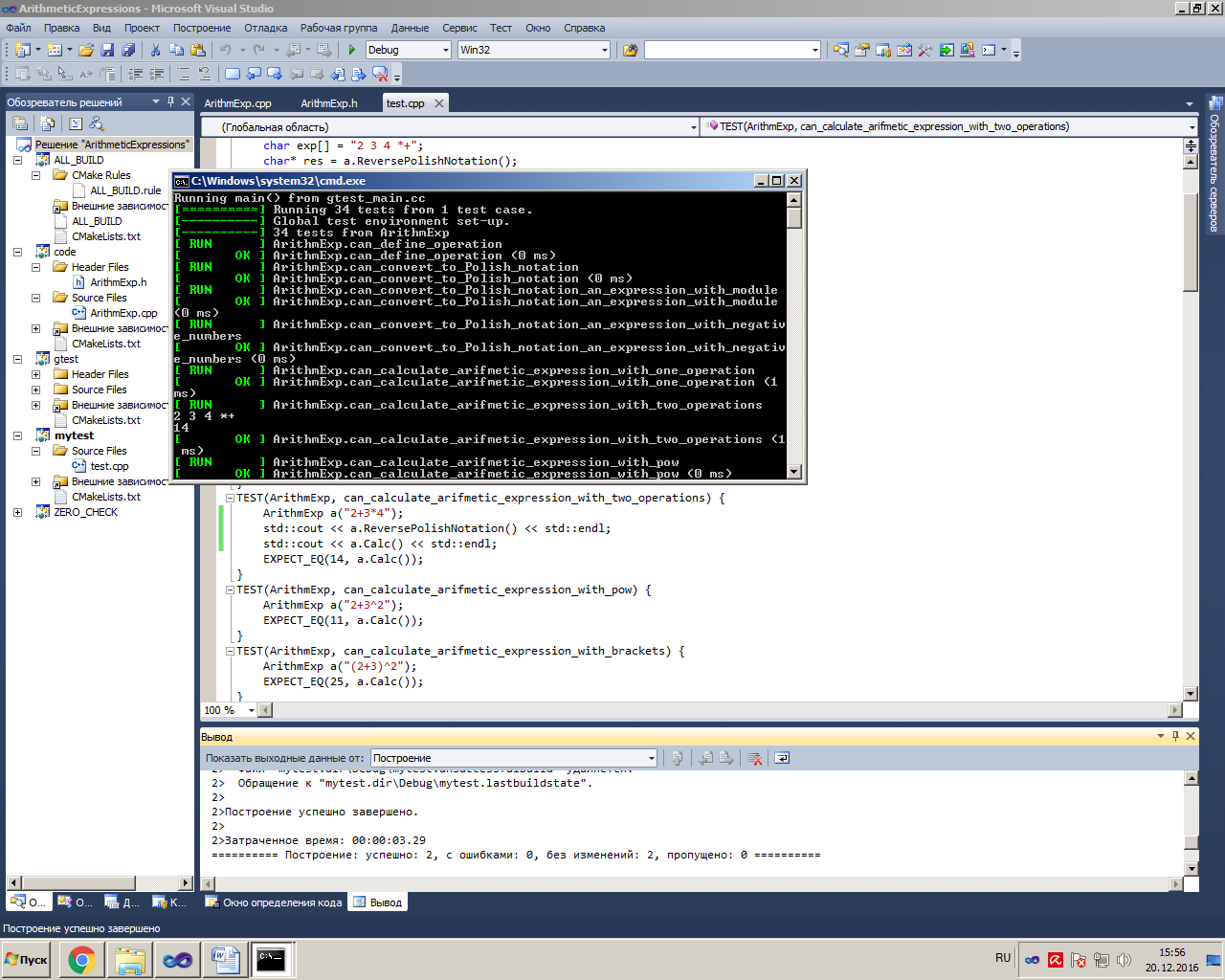
Интерфейс класса для разбора и вычисления арифметических выражений:



Пример работы с классом:

1. Ввод арифметического выражения: ArithmExp a("2+3\*4");
2. Перевод в обратную польскую нотацию:
3. С сохранением в переменную: сhar\* rpn = a.ReversePolishNotation();
4. С выводом на экран: std::cout << a.ReversePolishNotation() << std::endl;



1. Вычисление результата:
2. С сохранением в переменную: int res = a.Calc();
3. С выводом на экран: std::cout << a.Calc() << std::endl;

Пункт 2 можно опустить ( Метод Calc() сам вызывает метод ReversePolishNotation()).

# Руководство программиста

## Общее описание структуры программного комплекса.

Разработанная программа состоит из заголовочного файла «ArithmExp.h», содержащего интерфейс класса ArithmExp, файла исходного кода «ArithmExp.cpp», содержащего реализацию методов класса, и файла исходного кода «test.cpp» , содержащего тесты, проверяющие корректность работы методов класса.Также подключена библиотека Google Test.

Алгоритм работы с классом ArithmExp:

1. Создание объекта (ввод строки-выражения в качестве параметра конструктора-инициализатора).
2. Ввод значений параметров в качестве аргументов соответствующего метода.
3. Получение результата (вызов метода класса, вычисляющего выражение и присваивание полученного значения переменной, либо вывод значения на экран)

## Описание структур данных.

class ArithmExp { // Класс, осуществляющий разбор и вычисление арифметических выражений

char \*s; // Строка, в которой хранится арифметическое выражение

char \*p; // Строка, в которой хранятся имена параметров

int \*vp; // Массив значений параметров

int np; // Количество введенных параметров

public:

ArithmExp(char \*exp //Указатель на строку-выражение  
 ); // Конструктор-инициализатор

~ArithmExp(void); // Деструктор

void InputParametr(char c, // Имя параметра

int value // Значение параметра

); // Метод, осуществляющий ввод параметра

bool IsOp(char c); // Метод, определяющий, является ли символ знаком операции

int Priority(char op); // Метод, определяющий приоритет операции

char\* ReversePolishNotation(); // Метод, преобразующий исходное выражение в обратную   
 польскую запись (в случае некорректного выражения   
 вызывает исключение)

int Calc(); // Метод, выполняющий вычисления (в случае некорректного выражения   
 вызывает исключение)

};

## Описание алгоритмов.

Чтобы вычислить арифметическое выражение, введем промежуточный этап, на котором исходное выражение представим в виде обратной польской нотации. Обратная польская нотация (ОПН) — форма записи [математических](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%82%D0%B5%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0) и логических выражений, в которой [операнды](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D0%BD%D0%B4) расположены перед знаками [операций](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D1%8F_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)). Удобство обратной польской нотации заключается в том, что выражения, представленные в такой форме, очень легко вычислять, причём за линейное время.

### Алгоритм преобразования выражения из инфиксной формы в ОПН

В данной лабораторной работе используется алгоритм Дейкстры для преобразования выражений из инфиксной нотации в ОПН. Алгоритм основан на стеке. В преобразовании участвуют две текстовых переменных: входная и выходная строки. В процессе преобразования используется стек, хранящий ещё не добавленные к выходной строке операторы. Преобразующая программа читает входную строку последовательно символ за символом (символ — это не обязательно буква), выполняет на каждом шаге некоторые действия в зависимости от того, какой символ был прочитан.

В программе данный алгоритм дополнен проверкой выражения на корректность. Исключения, свидетельствующие об ошибке, будут вызваны в случаях:

1. Выражение содержит недопустимые символы
2. Количество открывающихся скобок (модулей) и не равно количеству закрывающихся
3. Не введены значения параметров

*Начало алгоритма:*

1. Совершаем обход строки, пока не встречен ноль
2. Проверяем, является ли первый элемент строки минусом. Если да, то делаем вывод, что он унарный, и записываем в выходную строку, после него записываем пробел как некоторое указание на унарный минус
3. Если минус встречен не на первой позиции, проверяем, является ли предыдущий элемент открывающейся скобкой (модулем). Если да, то записываем в выходную строку унарный минус
4. Если текущий элемент не является знаком операции, то
5. Если текущий элемент – строчная латинская буква, то
6. Если массив имен переменных пуст, то выдать сообщение об ошибке
7. Совершить обход массива имен параметров, в случае совпадения с текущим символом записать соответствующее значение из массива значений параметров в выходную строку
8. Если символ не найден в массиве имен переменных, выдать сообщение об ошибке
9. Если текущий элемент – цифра, то
10. Переписывать цифры из исходной строки в выходную пока не встретится знак операции. Если встретился недопустимый символ, выдать сообщение об ошибке
11. В противном случае, символ не является ни знаком операции, ни строчной латинской буквой, ни цифрой, следовательно, это недопустимый элемент и нужно выдать сообщение об ошибке (вызвать исключение)
12. Если стек пуст, или на вершине стека открывающаяся скобка (модуль), или текущий символ – открывающаяся скобка (модуль), то
13. Кладем символ в стек
14. Если это открывающаяся скобка (модуль) увеличиваем соответствующий счетчик на 1
15. Если текущий символ - закрывающаяся скобка, то
16. Уменьшаем счетчик скобок на 1
17. Пока на вершине стека не окажется открывающая скобка, выталкиваем содержимое стека в выходную строку
18. Удаляем открывающуюся скобку из стека
19. Если текущий символ – закрывающийся модуль, то
20. Уменьшаем счетчик модулей на 1
21. Пока на вершине стека не окажется модуль, выталкиваем содержимое стека в выходную строку
22. Удаляем модуль из стека
23. Записываем две черты модуля в выходную строку
24. Если текущий символ – знак возведения в степень, то добавляем его в стек
25. Если текущий элемент - другая операция, то
26. Выталкиваем из стека операции с большим либо равным приоритетом
27. Кладем на вершину стека текущую операцию
28. Если счетчик скобок (модулей) не равен нулю, вызвать исключение
29. Пока стек не пуст, выталкивать содержимое в выходную строку

*Конец алгоритма*

Обработка операции возведения в степень осуществляется отдельно от других операций ввиду её правоассоциативности.

Чтобы установить, является ли модуль открывающимся, проверялись следующие условия:

1. Счетчик модулей равен 0
2. Предыдущий символ был открывающимся модулем (и был добавлен в стек)
3. Предыдущий символ – арифметическая операция (+, -, \*, /, ^)

### Алгоритм вычисления выражения, представленного в ОПН

Автоматизация вычисления выражений в обратной польской нотации основана на использовании стека.

*Начало алгоритма*

1. Совершаем обход ОПН, пока не встретится ноль
2. Если текущий символ не знак операции, то переводим операнд из символьного представление в число типа int и кладем его в стек
3. Если текущий символ – знак операции, то
4. Достаем из стека нужное количество операндов
5. Выполняем соответствующую операцию над ними
6. Результат кладем на вершину стека
7. Если в стеке остался один элемент, то это результат
8. В противном случае, выражение некорректно и нужно вызвать исключение

*Конец алгоритма*

Контроль корректности выражения осуществляется путем проверки следующих условий:

1. Указатель на ОПН не равен нулю (метод, преобразующий перевод выражения из инфиксной формы в ОПН не выявил ошибку)
2. В процессе вычислений не встретилась попытка деления на ноль
3. После всех вычислений в стеке остался один элемент

# Заключение

Разработан класс для разбора и вычисления арифметических выражений, использующий алгоритм перевода выражения в обратную польскую нотацию и вычисление выражения в форме ОПН. Класс вычисляет корректные выражения и вызывает исключение в случае недопустимых выражений. Реализованы необходимые тесты.

# Список литературы

1. Т. Пратт, М. Зелковиц. Языки программирования: разработка и реализация = Terrence W. Pratt, Marvin V. Zelkowitz. Programming Languages: Design and Implementation. — 4-е издание. — Питер, 2002.
2. Бьёрн Страуструп.  Язык программирования C++ = The C++ Programming Language / Пер. с англ. — 3-е изд. — СПб.; М.: Невский диалект — Бином, 1999.

# Приложение 1. Arithm.h

#ifndef C\_\_GIT\_ARITHMETICEXPRESSIONS\_ARITHMETICEXPRESSIONS\_INCLUDE\_ARITHMEXP\_H\_

#define C\_\_GIT\_ARITHMETICEXPRESSIONS\_ARITHMETICEXPRESSIONS\_INCLUDE\_ARITHMEXP\_H\_

class ArithmExp {

char \*s;

char \*p;

int \*vp;

int np;

public:

ArithmExp(char \*exp);

~ArithmExp(void);

void InputParametr(char c, int value);

bool IsOp(char c);

int Priority(char op);

char\* ReversePolishNotation();

int Calc();

};

#endif // C\_\_GIT\_ARITHMETICEXPRESSIONS\_ARITHMETICEXPRESSIONS\_INCLUDE\_ARITHMEXP\_H\_

# Приложение 2. Arithm.cpp

#include "ArithmExp.h"

#include <stack>

#include "math.h"

#include <iostream>

ArithmExp::ArithmExp(char \*exp) {

int n;

for (n=0; exp[n] != 0; n++) {}

s = new char[n+1];

for (int i=0; i < n+1; i++)

s[i] = exp[i];

p = 0;

vp = 0;

np = 0;

}

ArithmExp::~ArithmExp(void) {

delete []s;

delete []p;

delete []vp;

}

void ArithmExp::InputParametr(char c, int value) {

np++;

p = (char\*)realloc(p, (np+1)\*sizeof(char));

p[np-1] = c;

p[np] = 0;

vp = (int\*)realloc(vp, np\*sizeof(int));

vp[np-1] = value;

}

bool ArithmExp::IsOp(char c) {

if ( (c == '+') || (c == '-') || (c == '\*') || (c == '/') || (c == '^') || (c == '|') || ( c == '(') || (c == ')') )

return true;

else

return false;

}

int ArithmExp::Priority(char op) {

switch (op) {

case '+':

case '-': return 1; break;

case '\*':

case '/': return 2; break;

case '^':

case '|': return 3; break;

}

}

char\* ArithmExp::ReversePolishNotation() {

std::stack<char> st;

int n, k = 0;

int f = 0, br = 0;

bool flag = 0, fl = 0;

for (n=0; s[n] != 0; n++) {}

char \*res = 0;

for (int i=0; s[i] != 0; i++) {

fl = 0;

if ( (i == 0) && (s[i] == '-') ) {

res = (char\*)realloc(res, (k+2)\*sizeof(char));

res[k++] = '-';

res[k++] = ' ';

continue;

}

if (i != 0) {

if ( ( (s[i-1] == '(') && (s[i] == '-') ) || ( (s[i-1] == '|') && (s[i] == '-') ) ) {

res = (char\*)realloc(res, (k+2)\*sizeof(char));

res[k++] = '-';

res[k++] = ' ';

continue;

}

}

if (!IsOp(s[i])) {

if ( ( (s[i] > 96) && (s[i] < 123) ) ) {

if (p == 0) {

std::cout << "Input parametr" << std::endl;

return 0;

}

for (int j=0; p[j] != 0; j++) {

if (s[i] == p[j]) {

fl = 1;

int ns = 0;

if (vp[j] < 0) {

res = (char\*)realloc(res, (k+2)\*sizeof(char));

res[k++] = '-';

res[k++] = ' ';

}

for (int tmp=abs(vp[j]); tmp != 0; ns++, tmp=tmp/10) {}

int tmp = abs(vp[j]);

for (int l=ns-1; l >= 0; l--) {

res = (char\*)realloc(res, (k+1)\*sizeof(char));

res[k++] = tmp / pow(10.0, l) + '0';

tmp = tmp % (int)pow(10.0, l);

}

res = (char\*)realloc(res, (k+1)\*sizeof(char));

res[k++] = ' ';

break;

} else {

continue;

}

}

if (fl == 0) {

std::cout << "Input parametr" << std::endl;

return 0;

}

} else {

if ( (s[i] > 47) && (s[i] < 58) ) {

for (; ( (!IsOp(s[i])) && (s[i] != 0) ); i++) {

if (!( (s[i] > 47) && (s[i] < 58) ) )

throw 1;

res = (char\*)realloc(res, (k+1)\*sizeof(char));

res[k++] = s[i];

}

res = (char\*)realloc(res, (k+1)\*sizeof(char));

res[k++] = ' ';

i--;

} else {

throw 1;

}

}

} else {

if ((st.empty()) || (s[i] == '(') || ((st.top() == '(') && (s[i] != ')')) || ((s[i] == '|') && (f == 0)) || ((st.top() == '|') && (s[i] != '|')) || ((s[i] == '|') && (((s[i-1] == '|') && (flag == 1)) || ((s[i-1] == '+') || (s[i-1] == '-') || (s[i-1] == '\*') || (s[i-1] == '/') || (s[i-1] == '^'))))) {

st.push(s[i]);

flag = 1;

if (s[i] == '|')

f++;

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

br++;

} else {

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

br--;

while (st.top() != '(') {

res = (char\*)realloc(res, (k+1)\*sizeof(char));

res[k++] = st.top();

st.pop();

}

st.pop();

} else {

if ( (s[i] == '|') && (f > 0) ) {

flag = 0;

while (st.top() != '|') {

res = (char\*)realloc(res, (k+1)\*sizeof(char));

res[k++] = st.top();

st.pop();

}

st.pop();

res = (char\*)realloc(res, (k+2)\*sizeof(char));

res[k++] = '|';

res[k++] = '|';

f--;

} else {

if (s[i] == '^') {

st.push(s[i]);

} else {

if (Priority(st.top()) < Priority(s[i])) {

st.push(s[i]);

} else {

res = (char\*)realloc(res, (k+1)\*sizeof(char));

res[k++] = st.top();

st.pop();

st.push(s[i]);

}

}

}

}

}

}

}

if ((br != 0) || (f != 0))

throw 1;

while (!st.empty()) {

res = (char\*)realloc(res, (k+1)\*sizeof(char));

res[k++] = st.top();

st.pop();

}

res[k] = 0;

return res;

}

int ArithmExp::Calc() {

char \*rpn = ReversePolishNotation();

std::stack<int> st;

int n;

int res;

if (rpn == 0)

throw 1;

for (int i=0; rpn[i] != 0; i++) {

if (!IsOp(rpn[i])) {

int operand = 0;

for (n=0; rpn[i+n] != ' '; n++) {}

for (int j=0; rpn[i] != ' '; i++, j++)

operand+=((rpn[i]-'0')\*pow(10.0, n-1-j));

st.push(operand);

} else {

if (rpn[i] == '^') {

int r;

double l;

if (!st.empty()) {

r = st.top();

st.pop();

} else {

throw 1;

}

if (!st.empty()) {

l = st.top();

st.pop();

} else {

throw 1;

}

st.push(pow(l, r));

}

if (rpn[i] == '+') {

int r, l;

if (!st.empty()) {

r = st.top();

st.pop();

} else {

throw 1;

}

if (!st.empty()) {

l = st.top();

st.pop();

} else {

throw 1;

}

st.push(l+r);

}

if (rpn[i] == '-') {

if (rpn[i+1] == ' ') {

i+=2;

double operand = 0;

for (n=0; rpn[i+n] != ' '; n++) {}

for (int j=0; rpn[i] != ' '; i++, j++)

operand+=((rpn[i]-'0')\*pow(10.0, n-1-j));

st.push(-operand);

} else {

int r, l;

if (!st.empty()) {

r = st.top();

st.pop();

} else {

throw 1;

}

if (!st.empty()) {

l = st.top();

st.pop();

} else {

throw 1;

}

st.push(l-r);

}

}

if (rpn[i] == '\*') {

int r, l;

if (!st.empty()) {

r = st.top();

st.pop();

} else {

throw 1;

}

if (!st.empty()) {

l = st.top();

st.pop();

} else {

throw 1;

}

st.push(l\*r);

}

if (rpn[i] == '/') {

int r, l;

if (!st.empty()) {

r = st.top();

st.pop();

} else {

throw 1;

}

if (!st.empty()) {

l = st.top();

st.pop();

} else {

throw 1;

}

if (r == 0)

throw 1;

else

st.push(l/r);

}

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

int tmp;

if (!st.empty()) {

tmp = st.top();

st.pop();

} else {

throw 1;

}

st.push(abs(tmp));

i++;

}

}

}

res = st.top();

st.pop();

if (st.empty()) {

return res;

} else {

std::cout << "error expression";

throw 1;

}

}

# Приложение 3. test.cpp

#include < gtest/gtest.h>

#include "ArithmExp.h"

#include <iostream>

TEST(ArithmExp, can\_define\_operation) {

ArithmExp a("2+3\*4");

EXPECT\_TRUE(a.IsOp('+'));

}

TEST(ArithmExp, can\_convert\_to\_Polish\_notation) {

ArithmExp a("2+3\*4");

char exp[] = "2 3 4 \*+";

char\* res = a.ReversePolishNotation();

for (int i=0; exp[i] != 0; i++)

EXPECT\_EQ(exp[i], res[i]);

}

TEST(ArithmExp, can\_convert\_to\_Polish\_notation\_an\_expression\_with\_module) {

ArithmExp a("|2+3|\*4");

char exp[] = "2 3 +||4 \*";

char\* res = a.ReversePolishNotation();

for (int i=0; exp[i] != 0; i++)

EXPECT\_EQ(exp[i], res[i]);

}

TEST(ArithmExp, can\_convert\_to\_Polish\_notation\_an\_expression\_with\_negative\_numbers) {

ArithmExp a("2+(-53)\*4");

char exp[] = "2 - 53 4 \*+";

char\* res = a.ReversePolishNotation();

for (int i=0; exp[i] != 0; i++)

EXPECT\_EQ(exp[i], res[i]);

}

TEST(ArithmExp, can\_calculate\_arifmetic\_expression\_with\_one\_operation) {

ArithmExp a("2+3");

EXPECT\_EQ(5, a.Calc());

}

TEST(ArithmExp, can\_calculate\_arifmetic\_expression\_with\_two\_operations) {

ArithmExp a("2+3\*4");

std::cout << a.ReversePolishNotation() << std::endl;

std::cout << a.Calc() << std::endl;

EXPECT\_EQ(14, a.Calc());

}

TEST(ArithmExp, can\_calculate\_arifmetic\_expression\_with\_pow) {

ArithmExp a("2+3^2");

EXPECT\_EQ(11, a.Calc());

}

TEST(ArithmExp, can\_calculate\_arifmetic\_expression\_with\_brackets) {

ArithmExp a("(2+3)^2");

EXPECT\_EQ(25, a.Calc());

}

TEST(ArithmExp, can\_calculate\_arifmetic\_expression\_with\_four\_operations\_and\_brakets) {

ArithmExp a("23+(2+3^2)\*3");

EXPECT\_EQ(56, a.Calc());

}

TEST(ArithmExp, can\_calculate\_arifmetic\_expression\_with\_module) {

ArithmExp a("2+|123|\*2");

EXPECT\_EQ(248, a.Calc());

}

TEST(ArithmExp, can\_calculate\_arifmetic\_expression\_with\_div) {

ArithmExp a("2+48/2");

EXPECT\_EQ(26, a.Calc());

}

TEST(ArithmExp, can\_calculate\_arifmetic\_expression\_with\_an\_negative\_number\_1) {

ArithmExp a("2+3\*(-4)");

EXPECT\_EQ(-10, a.Calc());

}

TEST(ArithmExp, can\_calculate\_arifmetic\_expression\_with\_an\_negative\_number\_2) {

ArithmExp a("-3\*4+2");

EXPECT\_EQ(-10, a.Calc());

}

TEST(ArithmExp, can\_calculate\_arifmetic\_expression\_with\_an\_negative\_number\_3) {

ArithmExp a("5\*4+(-2\*3+1)");

EXPECT\_EQ(15, a.Calc());

}

TEST(ArithmExp, can\_calculate\_arifmetic\_expression\_with\_an\_negative\_number\_in\_module) {

ArithmExp a("5\*4+(|-2|\*3+1)");

EXPECT\_EQ(27, a.Calc());

}

TEST(ArithmExp, can\_calculate\_arifmetic\_expression\_with\_expression\_in\_module) {

ArithmExp a("5\*4+|-2\*3+1|");

EXPECT\_EQ(25, a.Calc());

}

TEST(ArithmExp, can\_calculate\_arifmetic\_expression\_with\_nested\_modules\_1) {

ArithmExp a("5\*4+|3+|-2|\*3+1|");

EXPECT\_EQ(30, a.Calc());

}

TEST(ArithmExp, can\_calculate\_arifmetic\_expression\_with\_nested\_modules\_2) {

ArithmExp a("5\*4+||-2|\*3+1|");

EXPECT\_EQ(27, a.Calc());

}

TEST(ArithmExp, can\_calculate\_arifmetic\_expression\_with\_nested\_modules\_3) {

ArithmExp a("||3|+|-7|+|18||");

EXPECT\_EQ(28, a.Calc());

}

TEST(ArithmExp, can\_calculate\_arifmetic\_expression\_with\_right\_associative\_operation) {

ArithmExp a("2^2^3");

EXPECT\_EQ(256, a.Calc());

}

TEST(ArithmExp, can\_calculate\_arifmetic\_expression\_with\_parametr) {

ArithmExp a("a+1");

a.InputParametr('a', 5);

EXPECT\_EQ(6, a.Calc());

}

TEST(ArithmExp, can\_convert\_to\_Polish\_notation\_twelve) {

ArithmExp a("a+1");

a.InputParametr('a', 155);

EXPECT\_EQ(156, a.Calc());

}

TEST(ArithmExp, throw\_when\_there\_are\_not\_enough\_parametrs) {

ArithmExp a("a+b");

a.InputParametr('a', 155);

EXPECT\_ANY\_THROW(a.Calc());

}

TEST(ArithmExp, can\_calculate\_arifmetic\_expression\_with\_parametr\_with\_negative\_value) {

ArithmExp a("a+b");

a.InputParametr('a', -52);

a.InputParametr('b', 57);

EXPECT\_EQ(5, a.Calc());

}

TEST(ArithmExp, can\_calculate\_arifmetic\_expression\_with\_parametr\_with\_negative\_value\_in\_module) {

ArithmExp a("|a|+|b|");

a.InputParametr('a', -52);

a.InputParametr('b', 57);

EXPECT\_EQ(109, a.Calc());

}

TEST(ArithmExp, throw\_when\_there\_are\_not\_enough\_brackets) {

ArithmExp a("(2+3");

EXPECT\_ANY\_THROW(a.Calc());

}

TEST(ArithmExp, throw\_when\_there\_are\_not\_enough\_module) {

ArithmExp a("|2+3");

EXPECT\_ANY\_THROW(a.Calc());

}

TEST(ArithmExp, throw\_when\_the\_expression\_is\_incorrect\_1) {

ArithmExp a("2+3(5)\*7");

EXPECT\_ANY\_THROW(a.Calc());

}

TEST(ArithmExp, throw\_when\_the\_expression\_is\_incorrect\_2) {

ArithmExp a("2+8+||");

EXPECT\_ANY\_THROW(a.Calc());

}

TEST(ArithmExp, throw\_when\_the\_expression\_is\_incorrect\_3) {

ArithmExp a("2+8+()");

EXPECT\_ANY\_THROW(a.Calc());

}

TEST(ArithmExp, throw\_when\_there\_are\_not\_enough\_operands) {

ArithmExp a("2+");

EXPECT\_ANY\_THROW(a.Calc());

}

TEST(ArithmExp, throw\_when\_there\_is\_no\_operands) {

ArithmExp a("+");

EXPECT\_ANY\_THROW(a.Calc());

}

TEST(ArithmExp, throw\_when\_there\_is\_div\_by\_zero) {

ArithmExp a("2+8/0");

EXPECT\_ANY\_THROW(a.Calc());

}

TEST(ArithmExp, throw\_when\_the\_expression\_contains\_invalid\_characters) {

ArithmExp a("2+3%8");

EXPECT\_ANY\_THROW(a.Calc());

}