Федеральное агентство по образованию Российской Федерации

Государственное образовательное учреждение

Высшего профессионального образования

Нижегородский государственный университет им. Н.И. Лобачевского

Институт информационных технологий математики механики

Польская запись

Отчет по лабораторной работе

Выполнил:

студент ИИТММ гр. 381706-2

Антипин А.С.\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Проверил:

ассистент каф. МОСТ, ИИТММ

Лебедев И.Г\_\_\_\_\_\_\_\_\_\_\_\_\_\_.

Нижний Новгород

2018 г.

**Содержание**

[Введение 3](#_Toc536821609)

[Постановка целей и задач 4](#_Toc536821610)

[Руководство пользователя 5](#_Toc536821611)

[Руководство программиста 6](#_Toc536821612)

[Описание структуры программы 6](#_Toc536821613)

[Описание структур данных 7](#_Toc536821614)

[Описание алгоритмов 8](#_Toc536821615)

[Заключение 10](#_Toc536821616)

[Литература 11](#_Toc536821617)

[Приложение 12](#_Toc536821618)

# Введение

Для визуального представления математических выражений мы привыкли использовать прямой порядок запись чисел и арифметических знаков, т.е. выражение a + b мы привыкли понимать так: необходимо к числу a прибавить число b. Но существует и другая форма записи числа, которая подразумевает под собой изначальную запись аргументов, а уже потом арифметических знаков. Такую запись принято называть обратной польской записью и выглядит она следующим образом: выражение a + b преобразуется в ab +. Более сложный пример приведен на рис. 1. По данным Википедии, данная форма записи используется в «организации байт-кода конфигураций прикладных решений системы 1С:Предприятие. Официального подтверждения компания 1С не дает, но пользователи данной системы на специализированных форумах приводят доказательства и алгоритмы, позволяющие декомпилировать исходные тексты.».



рис. 1 (Инфиксное выражение – слева и аналогичное ей постфиксная запись справа)

Для   
{\displaystyle (1+2)\times 4+3}преобразование из обычной записи в польскую обычно используют один стек, в нашем случае для преобразования и подсчета выражения будет использоваться два стека: один – целочисленных чисел, в котором будем хранить преобразованные из char числа (в нем окажется и результат вычислений), и стек символов – арифметических знаков.

# Постановка целей и задач

Основной целью лабораторной работы является создание структуры данных «Polish string» и реализация такого метода как: сосчитать выражение:

Для реализации алгоритмов будет использоваться классы Polish string, Calculator и структура Number.

Для проверки правильности работы этого класса будут написаны тесты с использованием фреймворка Google Test, а также тестовый образец программы, которая использует данный класс.

# Руководство пользователя

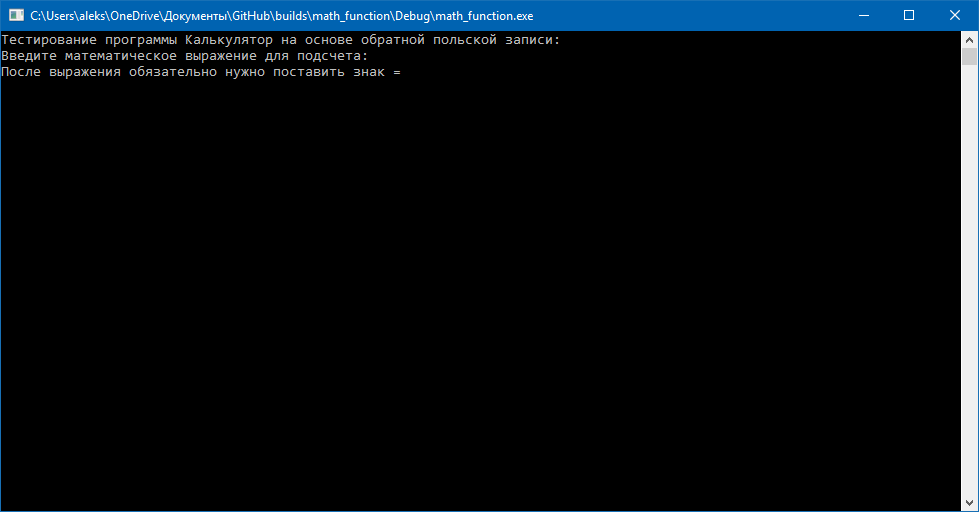
После запуска программы пользователя встречает консольное окно (рис. 2):

рис. 2 (вывод работы калькулятора для пользователя)

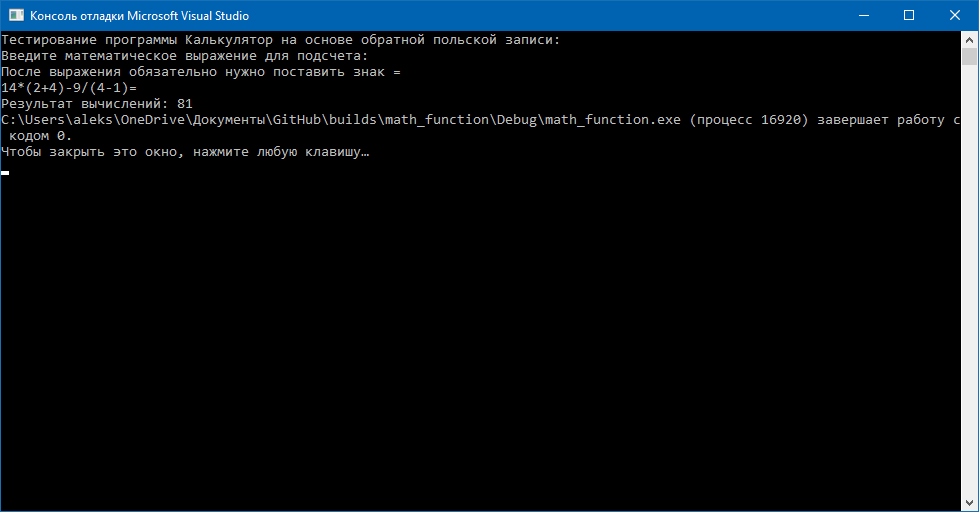
От пользователя требуется ввести какое-либо математическое выражение без пробелов и в конце поставить знак равно как признак его окончания. Затем нажать Enter, и программа сама подсчитает его и выведет результат на экран (рис. 3).

рис. 3 (вывод конечного результата вычислений программы)

# Руководство программиста

## Описание структуры программы

При создании объекта типа Calculator подразумевается, что максимальная длина строки, которую передают на вычисление не превышает 100 символов. Такого объема достаточно для вычисления несложных математических выражений, а при необходимости структуру можно легко изменить для более большого объема информации.

Для реализации алгоритмов будет создано два класса Calculator и Polish\_string, а также структура Number.

Проверку правильности работы алгоритмов будет осуществлять фреймворк Google Test.

**Polish\_string.cpp**

**Структура Number**

Древо классов

**Класс gtest**

**test\_main.cpp**

**Polish\_string\_test.cpp**

**Проект для пользователя**

**main.cpp**

**Polish\_string.h**

**Класс Polish\_string**

**Polish\_record.h**

**Polish\_record.cpp**

**Класс Calculator**

**Calculator\_test.cpp**

**Структура Number:**

Эта структура временно хранит либо текущее число, либо текущий арифметический знак, взятые из строки, переданной пользователем.

**Класс Polish\_string:**

Этот класс включает в себя методы обработки строки переданной пользователем для ее дальнейшего преобразования и вычисления результата.

**Класс Calculator:**

йКласс, в котором реализованы методы вычисления и обработки текущих символов, которые поступили из строки.

**Класс gtest:**

Класс gtest реализует тестирование всех классов, по средствам фреймворка Google Test. Тесты пишутся для каждого метода классов, каждого ветвления этих методов и для всех возможных исключений этих методов.

**Проект main:**

В данном проекте реализован примет использования калькулятора, в котором пользователь может ввести математическое выражение и получить результат вычислений.

## Описание структур данных

**Структура Number:**

char\* number – указатель на выделяемую память под текущий символ;

int len – длина текущего символа;

**Класс Polish\_string:**

char\* str – указатель на память выделяемую под строку, переданную пользователем;

int index – индекс текущего элемента в строке;

int char\_count – количество символов в строке;

char\* servicechar – массив математических выражений, которые могут использоваться;

**Класс Calculator:**

Polich\_string str – переменная типа Polish\_string;

Stack<int>\* numbers – стек под числа в строке;

Stack<char>\* s\_char – стек под математические выражения;

Exceptions\_from\_polinom\_and\_calculator exception – переменная для вызова исключений;

**Описание методов:**

|  |  |
| --- | --- |
| Метод: | Описание: |
| Number::Number() | Конструктор по умолчанию для Number. |
| Polich\_string::Polich\_string() | Конструктор по умолчанию для класса Polich\_string. |
| int Polich\_string::GetIndex() | Метод, который позволяет вернуть индекс рассматриваемой строки. |
| void Polich\_string::WrightString(char\* string) | Метод, который принимает строку. |
| Number Polich\_string::GetNextChar() | Метод, который возвращает следующий символ в формате Number для класса Calculator. |
| bool Polich\_string::IsServiceChar() | Метод, который определяет является ли текущий символ математических знаков. |
| Calculator::Calculator(char\* string) | Конструктор с параметром, который принимает строку из символов. |
| int Calculator::Calculate() | Основной метод, который производит вычисления. |
| int Calculator::AddToStacks(Number n) | Метод, который обеспечивает заполнение стеков новыми символами. |
| int Calculator::CharToInt(Number st) | Метод, который по принятой строке возвращает числовое значение. |
| int Calculator::Priority() | Метод, который возвращает приоритет поступившей операции. |
| bool Calculator::WhatsMore() | Метод, который дает разрешение на выполнение операции по ее приоритету. |

## Описание алгоритмов

**Подробное описание некоторых методов**

Превращение строки в целочисленное значение:

* Заводятся две целочисленные переменные, одна для результата, вторая для исключения, если оно возникнет.
* В цикле от 0 до длины данного символа, которая передается в Number, из каждого элемента строки вычитается символ «0», тем самым можно узнать какое это число. Затем оно умножается на 10 в степени текущей итерации цикла. После чего это число складывается с предыдущим своим значением
* Если какое-то число оказывается большим 9, или меньше 0 на какой-то итерации цикла до умножения на 10 в степени, то вызывается исключение, т.к. этот символ не является числом, а кокай-либо другой символ.

Подсчет результата:

* Создаются две целочисленные переменны, которые будут отвечать за аргументы в текущей операции, а также создается булева переменная для разрешения или запрета выполнения операции на определенной итерации цикла;
* Вводится бесконечный цикл, который закончится, когда в целочисленном стеке останется только одно число, а в знаковом все закончатся.
* Получение следующего символа, распределение этого символа в нужный стек и определение приоритета математических выражений.
* Если можно выполнить математическое действие, то:
  + Если встречается открывающая скобка, то создается еще одна строка с элементами, расположенными в этой скобке, создается дополнительная переменная типа «Calculator», от которой рекурсивно вызывается метод подсчета текущего выражения, затем общий цикл одет на следующий круг.
  + Если же встречается какое-либо другое математическое выражение, то из числового стека берутся два верхних числа, над ними выполняется действие и результат кладется в числовой стек.
* После того, как в числовом стеке останется только одно число, оно возвращается.

# Заключение

Подводя итоги можно сказать, что все поставленные цели работы были выполнены, а именно созданы методы представления строки в математическое выражение и его подсчет. Все тесты к данной работе бы успешно пройдены, что подтверждает правильность ее работы и возможность применения в реальных задачах.

# Литература

* Учебные материалы к учебному курсу «Методы программирования» - Гергель В.П.
* https://ru.wikipedia.org/wiki/Обратная\_польская\_запись

# Приложение

**Polish\_string.h:**

#pragma once

#include <iostream>

struct Number

{

public:

char\* number;

int len;

Number();

};

class Polich\_string

{

private:

char\* str;

int index;

int char\_count;

char\* servicechar;

public:

Polich\_string();

int GetIndex();

void WrightString(char\* string);

Number GetNextChar();

bool IsServiceChar();

};

**Polish\_string.cpp:**

#include "Polish\_string.h"

Polich\_string::Polich\_string()

{

str = new char[100];

index = 0;

char\_count = 0;

servicechar = { "+-\*/^()=" };

}

int Polich\_string::GetIndex()

{

return index;

}

void Polich\_string::WrightString(char\* string)

{

for (int i = 0; i < 100; i++)

{

if (string[i - 1] == '=' && i > 0)

break;

str[i] = string[i];

char\_count++;

}

}

Number Polich\_string::GetNextChar()

{

Number tmp;

int j = 0;

if (IsServiceChar() == true && index != char\_count)

{

tmp.number[0] = str[index];

index++;

tmp.len++;

}

else

while (!IsServiceChar() && index != char\_count)

{

tmp.number[j] = str[index];

index++;

tmp.len++;

j++;

}

return tmp;

}

bool Polich\_string::IsServiceChar()

{

bool type = false;

for (int i = 0; i < 8; i++)

if (str[index] == servicechar[i])

return true;

return type;

}

Number::Number()

{

number = new char[100];

len = 0;

}

**Polish\_record.h:**

#pragma once

#include "Polish\_string.h"

#include "Stack.h"

#include "Exception\_class.h"

class Calculator

{

private:

Polich\_string str;

Stack<int>\* numbers;

Stack<char>\* s\_char;

Exceptions\_from\_polinom\_and\_calculator exception;

public:

Calculator(char\* string);

int Calculate();

int AddToStacks(Number n);

int CharToInt(Number st);

int Priority();

bool WhatsMore();

};

**Polish\_record.cpp:**

#include "Polish\_record.h"

#include <cmath>

Calculator::Calculator(char\* string)

{

str.WrightString(string);

numbers = new Stack<int>(100);

s\_char = new Stack<char>(100);

}

int Calculator::Calculate()

{

int a = 0, b = 0;

bool priority\_of\_simbols = true;

while (1)

{

Number tmp = str.GetNextChar();

if (numbers->GetIndex() == 1 && tmp.len == 0)

break;

int type = AddToStacks(tmp);

if (s\_char->GetIndex() > 1)

priority\_of\_simbols = WhatsMore();

if ((s\_char->GetIndex() > 0 && Priority() == 4) || (numbers->GetIndex() >= 2 && type > 0 && priority\_of\_simbols == true))

{

if (s\_char->GetWithoutDelete() == '(')

{

char\* inside\_string = new char[100];

int inside\_char\_count = 0;

int inside\_brecket\_count = 0;

while (1)

{

tmp = str.GetNextChar();

if (tmp.number[0] == '(')

inside\_brecket\_count++;

if (tmp.number[0] == ')' && inside\_brecket\_count == 0)

break;

else if (tmp.number[0] == ')')

inside\_brecket\_count--;

int i;

for (i = 0; i < tmp.len; i++)

inside\_string[i + inside\_char\_count] = tmp.number[i];

inside\_char\_count += tmp.len;

}

inside\_string[inside\_char\_count] = '=';

Calculator calcul(inside\_string);

numbers->Put(calcul.Calculate());

s\_char->Get();

continue;

}

char c = s\_char->Get();

if (s\_char->GetWithoutDelete() == '+')

{

a = numbers->Get();

b = numbers->Get();

numbers->Put(a + b);

s\_char->Get();

}

else if (s\_char->GetWithoutDelete() == '-')

{

a = numbers->Get();

b = numbers->Get();

numbers->Put(b - a);

s\_char->Get();

}

else if (s\_char->GetWithoutDelete() == '\*')

{

a = numbers->Get();

b = numbers->Get();

numbers->Put(a \* b);

s\_char->Get();

}

else if (s\_char->GetWithoutDelete() == '/')

{

a = numbers->Get();

b = numbers->Get();

numbers->Put(b / a);

s\_char->Get();

}

else if (s\_char->GetWithoutDelete() == '^')

{

a = numbers->Get();

b = numbers->Get();

numbers->Put(pow(b, a));

s\_char->Get();

}

s\_char->Put(c);

}

}

return numbers->Get();

}

int Calculator::AddToStacks(Number n)

{

if (n.number[0] == '+')

{

s\_char->Put('+');

return 1;

}

else if (n.number[0] == '-')

{

s\_char->Put('-');

return 1;

}

else if (n.number[0] == '\*')

{

s\_char->Put('\*');

return 1;

}

else if (n.number[0] == '/')

{

s\_char->Put('/');

return 1;

}

else if (n.number[0] == '^')

{

s\_char->Put('^');

return 1;

}

else if (n.number[0] == '(')

{

s\_char->Put('(');

return 1;

}

else if (n.number[0] == ')')

{

s\_char->Put(')');

return 1;

}

else if (n.number[0] == '=')

{

s\_char->Put('=');

return 2;

}

else

{

if (n.len != 0)

{

numbers->Put(CharToInt(n));

return 0;

}

else

return 3;

}

}

int Calculator::CharToInt(Number st)

{

int M = 0;

int N = 0;

for (int i = 0; i < st.len; i++)

{

M += (st.number[i] - '0') \* pow(10, st.len - i - 1);

N = st.number[i] - '0';

if (N > 9 || N < 0)

exception.except\_throw(106);

}

return M;

}

int Calculator::Priority()

{

if (s\_char->GetWithoutDelete() == '=')

return 0;

if (s\_char->GetWithoutDelete() == '+' || s\_char->GetWithoutDelete() == '-')

return 1;

if (s\_char->GetWithoutDelete() == '\*' || s\_char->GetWithoutDelete() == '/')

return 2;

if (s\_char->GetWithoutDelete() == '^')

return 3;

if (s\_char->GetWithoutDelete() == '(' || s\_char->GetWithoutDelete() == ')')

return 4;

}

bool Calculator::WhatsMore()

{

char ch = s\_char->Get();

int prior = Priority();

s\_char->Put(ch);

if (prior >= Priority())

return true;

else

return false;

}

**Polish\_string.cpp:**

#include <gtest.h>

#include "Polish\_string.h"

TEST(Polish\_string, can\_create\_some\_string)

{

ASSERT\_NO\_THROW(Polich\_string PS);

}

TEST(Polish\_string, is\_servise\_char\_true)

{

Polich\_string PS;

PS.WrightString("\*");

EXPECT\_EQ(1, PS.IsServiceChar());

}

TEST(Polish\_string, is\_servise\_char\_false)

{

Polich\_string PS;

PS.WrightString("0");

EXPECT\_EQ(0, PS.IsServiceChar());

}

TEST(Polish\_string, can\_return\_next\_service\_char)

{

Polich\_string PS;

Number A;

PS.WrightString("\*");

A = PS.GetNextChar();

EXPECT\_EQ('\*', A.number[0]);

}

TEST(Polish\_string, can\_return\_next\_char\_number)

{

Polich\_string PS;

Number A;

PS.WrightString("100");

A = PS.GetNextChar();

EXPECT\_EQ('1', A.number[0]);

EXPECT\_EQ('0', A.number[1]);

EXPECT\_EQ('0', A.number[2]);

}

TEST(Polish\_string, can\_return\_next\_char\_from\_some\_other\_chars)

{

Polich\_string PS;

Number A;

PS.WrightString("\*100+25");

PS.GetNextChar();

A = PS.GetNextChar();

EXPECT\_EQ('1', A.number[0]);

EXPECT\_EQ('0', A.number[1]);

EXPECT\_EQ('0', A.number[2]);

}

TEST(Polish\_string, can\_return\_next\_char\_from\_some\_other\_chars\_2)

{

Polich\_string PS;

Number A;

PS.WrightString("\*100+25");

PS.GetNextChar();

PS.GetNextChar();

A = PS.GetNextChar();

EXPECT\_EQ('+', A.number[0]);

}

TEST(Polish\_string, can\_return\_wright\_index)

{

Polich\_string PS;

PS.WrightString("\*100+25");

PS.GetNextChar();

PS.GetNextChar();

EXPECT\_EQ(4, PS.GetIndex());

}

**Calculator.cpp:**

#include <gtest.h>

#include "Polish\_record.h"

TEST(calculator, can\_create\_calculator)

{

char\* string = { "100" };

ASSERT\_NO\_THROW(Calculator calc(string));

}

TEST(calculator, can\_create\_calculator\_1)

{

char\* string = { "100" };

ASSERT\_NO\_THROW(Calculator calc(string));

}

TEST(calculator, can\_transformate\_char\_to\_int)

{

char\* string = { "100=" };

Calculator calc(string);

Polich\_string str;

str.WrightString(string);

Number num = str.GetNextChar();

EXPECT\_EQ(calc.CharToInt(num), 100);

}

TEST(calculator, can\_add\_to\_stack)

{

Number sign;

sign.number = { "+" };

sign.len = 1;

Number num;

num.number = { "100" };

num.len = 3;

Calculator calc\_sign("+");

ASSERT\_NO\_THROW(calc\_sign.AddToStacks(sign));

ASSERT\_NO\_THROW(calc\_sign.AddToStacks(num));

}

TEST(calculator, can\_calculate\_something)

{

char\* string = { "100+50=" };

Calculator calc(string);

EXPECT\_EQ(calc.Calculate(), 150);

}

TEST(calculator, can\_calculate\_something\_hard)

{

char\* string = { "100+50\*3=" };

Calculator calc(string);

EXPECT\_EQ(calc.Calculate(), 250);

}

TEST(calculator, can\_calculate\_something\_hard\_2)

{

char\* string = { "100\*3+50=" };

Calculator calc(string);

EXPECT\_EQ(calc.Calculate(), 350);

}

TEST(calculator, can\_calculate\_something\_hard\_3)

{

char\* string = { "149/2=" };

Calculator calc(string);

EXPECT\_EQ(calc.Calculate(), 74);

}

TEST(calculator, can\_calculate\_something\_very\_hard)

{

char\* string = { "100\*(2+3)=" };

Calculator calc(string);

EXPECT\_EQ(calc.Calculate(), 500);

}

TEST(calculator, can\_calculate\_something\_very\_very\_hard)

{

char\* string = { "100\*(2\*(3+1))=" };

Calculator calc(string);

EXPECT\_EQ(calc.Calculate(), 800);

}

TEST(calculator, can\_calculate\_something\_very\_very\_hard\_2)

{

char\* string = { "10\*(2\*(3+1)+4)=" };

Calculator calc(string);

EXPECT\_EQ(calc.Calculate(), 120);

}

TEST(calculator, can\_raise\_to\_a\_power)

{

char\* string = { "5^3=" };

Calculator calc(string);

EXPECT\_EQ(calc.Calculate(), 125);

}

TEST(calculator, can\_raise\_to\_a\_hard\_power)

{

char\* string = { "5^(2+1)=" };

Calculator calc(string);

EXPECT\_EQ(calc.Calculate(), 125);

}

TEST(calculator, can\_throw\_if\_whrong\_record)

{

char\* string = { "S^(2+1)=" };

Calculator calc(string);

ASSERT\_ANY\_THROW(calc.Calculate());

}

TEST(calculator, can\_throw\_if\_whrong\_record\_2)

{

char\* string = { "5^(;+1)=" };

Calculator calc(string);

ASSERT\_ANY\_THROW(calc.Calculate());

}

TEST(calculator, can\_throw\_if\_record\_without\_equally)

{

char\* string = { "2+1" };

Calculator calc(string);

ASSERT\_ANY\_THROW(calc.Calculate());

}

**main.cpp:**

#include <iostream>

#include "Polish\_record.h"

int main()

{

setlocale(LC\_ALL, "RUS");

std::cout << "Тестирование программы Калькулятор на основе обратной польской записи:\n";

std::cout << "Введите математическое выражение для подсчета:\nПосле выражения обязательно нужно поставить знак =\n";

char\* string = new char[100];

std::cin >> string;

Calculator calculator(string);

std::cout << "Результат вычислений: " << calculator.Calculate();

return 0;

}