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

Факультет вычислительной математики и кибернетики

Отчёт по лабораторной работе

Разбор и вычисление арифметических выражений

Выполнил:
студент ф-та ИИТММ гр. 0823-01

Глуздов А.Д.

Проверил:
к.т.н., ассистент каф. ПрИнж ИТММ.

Сиднев А.А.

Нижний Новгород
2016 г.

Содержание

Введение

В математике, физике практически каждая задача сводится к вычислительному процессу, к вычислению какого-либо выражения. Ученым не рационально тратить своё время на вычислительный этап. Кроме того, на сегодняшний день решаются масштабные задачи, где длина “того самого арифметического выражения” достигается таких величин, что на его решение “в столбик” не хватит и жизни. Поэтому задача вычисления арифметического выражения актуальна как никогда.

Постановка задачи

Реализовать класс, который

- принимает на вход строку, содержащую выражение
- выполняет её разбор
- выводит сообщение об ошибке при ее обнаружении
- выполняет вычисление значения выражения при заданных значениях переменных

Разбор и хранение выражения необходимо осуществлять в обратной польской записи. Необходимо реализовать тесты, содержащие различные типы выражений.

Руководство пользователя

Название класса:

Calculation

Публичные Методы

- **double Calculate**(string);

Основной метод класса для решения поставленной задачи. На вход принимает строку - арифметическое выражение в обычном виде. На выходе число - результат вычислений. В случае возникновения ошибки в ходе вычислительного процесса или по причине некорректности арифметического выражения выводит на экран ошибку.

- string **Parsing**(string);

Дополнительный метод класса, открытый пользователю для удобства, для решения дополнительных задач. На вход принимает строку - арифметическое выражение в обычном виде. На выходе строка - польская нотация данного арифметического выражения. В случае возникновения ошибки по причине некорректности арифметического выражения выводит на экран ошибку.

- **double Counting**(string);

Дополнительный метод класса, открытый пользователю для удобства, для решения дополнительных задач. На вход принимает строку - польская нотация арифметического выражения. На выходе число - результат вычислений. В случае возникновения ошибки в ходе вычислительного процесса выводит на экран ошибку.

- **void addVars**(**char**,**double**);

Метод класса, который добавляет переменную и ее значение в хранилище.

Описание

Для начала необходимо подключить Интерфейс Класса

#include "Calculation.h"

Чтобы выполнить вычисление арифметического выражения достаточно воспользоваться статическим методом **Calculate(*/арифм. выражение/*)**, который вернет число, либо выведет ошибку на экран.

Пример:

```
#include "Calculation.h"
#include <string>
int main() {
	Calculation calc;
	calc.addVars('x',2);
	return calc.Calculate("x+5"); // вернет 7
}
```

Руководство программиста

Описание структуры программы

Наша программа состоит из одной части; Класса **Calculation**.
Класс содержит ряд методов:

```
class Calculation{

    //Метод Calculate принимает выражение в виде строки и возвращает результат, в своей работе использует другие методы класса
   public double Calculate(string input)
   { ... }
    // Метод, достающий из хранилища переменные и их значения и вносит их в строку
   string SetVariables(string input);
   { ... }
    //Метод, преобразующий входную строку с выражением в постфиксную запись
   private string Parsing(string input)
   { ... }
     //Метод, вычисляющий значение выражения, уже преобразованного в постфиксную запись
   private double Counting(string input)
   { ... }

}
```

Это 4 основных метода класса, опишем их подробнее:

Calculate — метод общедоступный, ему передается выражение в виде строки, и он возвращает результат вычисления. Результат он получает используя другие методы.

SetVariables - приватный метод, достающий из хранилища переменные и их значения и вносит их в строку, заменяя тем самым сами переменные

Parsing — метод, которому основной метод Calculate передает выражение, и получвет его уже в постфиксной записи.

Counting — метод, который получая постфиксную запись выражения вычисляет его результат.

Также, помимо 4 основных методов, в классе еще 5 метода «обеспечения», это:

IsDelimeter — возвращает true, если проверяемый символ — разделитель

IsOperator — возвращает true, если проверяемый символ — оператор

GetPriority — метод возвращает приоритет проверяемого оператора, нужно для сортировки операторов

IsVariable - возвращает true, если проверяемый символ — переменная

Error - вспомогательная функция, которая в зависимости от числа выводит на экран текст соответствующей ошибки.

Описание структур данных

Calculation - основной класс, занимающийся вычислением арифметических выражений, а также приведением их к польской нотации.
Var Vars - поле класса **Calculation**; хранилище переменных и их значений.

Описание алгоритмов

В классе используется алгоритм Обратной польской записи.
Алгоритм Обратной польской нотации (ОПН) — форма записи математических выражений, в которой операнды расположены перед знаками операций. Также именуется как обратная польская запись, обратная бесскобочная запись (ОБЗ). © Wikipedia

В нашем классе мы приводим арифметическое выражение в постфиксною польскую нотацию в методе ***Parsing***, используя Стек.
Алгоритм

1. Обработка входного символа
    - Если на вход подан операнд, он помещается на вершину стека.
    - Если на вход подан знак операции, то соответствующая операция выполняется над требуемым количеством значений, извлеченных из стека, взятых в порядке добавления. Результат выполненной операции кладётся на вершину стека.
2. Если входной набор символов обработан не полностью, перейти к шагу 1.
3. После полной обработки входного набора символов результат вычисления выражения лежит на вершине стека.

Проект mytest

Описание структуры программы

Проект содержит функции, основанные на гугл тестах и тестирующие программу.

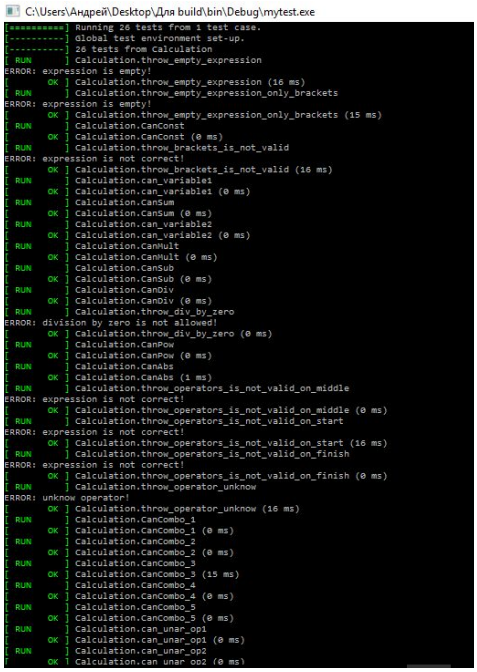
Описание алгоритмов

Каждый тест состоит из трёх частей:

1. Объявление переменных
2. Произведение действий над ними
3. Проверка, путём использования встроенных в gtest функций.

Заключение

В ходе лабораторной работы нам удалось реализовать класс для вычисления арифметических операций. Мы изучили и применили алгоритм обратной польской записи. В процессе мы использовали Google Tests для тестирования класса, написали 26 тестов, с которыми наш класс справился.



Литература

1. Столлингс, В. Структурная организация и архитектура компьютерных систем, 5-е изд.: Пер. с англ. — М.: Издательский дом «Вильямс», 2002. — 896 с.: ил. — Парал. тит. англ.
2. Johnson M. Superscalar Microprocessor Design. — Englewood Cliff, New Jersey: Prentice Hall, 1991.
3. 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. — 688 с. — (Классика Computer Science). — 4000 экз. — ISBN 5-318-00189-0

Приложения

Calculation.h

```
#pragma once
#include <string>
#include <stdexcept>
using namespace std;


class Calculation
{
public:
	Calculation();
	~Calculation();
	void addVars(char,double);
	double Calculate(string);
	string Parsing(string);
	double Counting(string);
	string SetVariables(string);
private:
	struct Var {
		char ch;
		double val;
		Var* next;
	};
	Var* Vars;
	bool IsDelimeter(char);
	bool IsOperator(char);
	int GetPriority(char);
	bool IsVariable(char);
	void Error(int);
};
```

```
template <typename T>

struct TNode{
  T data;
  TNode* next;
};

template <typename T>
class Stack{
  TNode<T>* top;
 public:
  Stack() {
    top = 0;
  }
  ~Stack();
  void push(T);
```

```
  T& gettop();
  bool pop();
  bool empty();
};

template <typename T>
Stack<T>::~Stack() {
  while (top != 0) {
    TNode<T>* buf = top->next;
    delete top;
    top = buf;
  }
}

template <typename T>
void Stack<T>::push(T x) {
  TNode<T>* node = new TNode<T>;
  node->data = x;
  node->next = top;
  top = node;
}

template <typename T>
T& Stack<T>::gettop() {
  if (top != 0)
    return top->data;
  else
    throw std::logic_error("Stack is empty!");
}

template <typename T>
bool Stack<T>::pop() {
  if (top != 0) {
    TNode<T>* s = top;
    top = top->next;
    delete s;
    return true;
  }
  else {
    return false;
  }
}

template <typename T>
bool Stack<T>::empty() {
  return !top;
}
```

Calculation.cpp

```
#include "Calculation.h"
#include <string>
#include <cctype>
#include <iostream>
#include <math.h>
using namespace std;

Calculation::Calculation()
{
	Vars = new Var;
	Vars->next = NULL;
}
Calculation::~Calculation()
{
	delete Vars;
}
double Calculation::Calculate(string input)
{
    string output = SetVariables(input);
	output = Parsing(output);

    double result = Counting(output);
    return result;
}
void Calculation::addVars(char a, double v)
{
	Var* buf = Vars;
	int ex = 0;
	while (buf) {
		if (a == buf->ch) {
			ex = 1;
			break;
		}
		buf = buf->next;
	}

	if (ex)
		buf->val = v;
	else {
		Var* newvar = new Var;
		newvar->ch = a;
		newvar->val = v;
		newvar->next = Vars;
		Vars = newvar;
	}
}
string Calculation::SetVariables(string input) // Метод, предоставляющий пользователю
заполнить переменные числовыми константами (или другими переменными)
{
    for (int i = 0; i < input.length(); i++) //Для каждого символа в строке
    {
		if(IsVariable(input[i])){
			Var* buf = Vars;
			bool flag = true;
			while(flag)
			{
```

```
                        if(buf->ch == input[i]){
                                input[i] = buf->val + '0';
                                flag = false;
                        }
                        buf = buf->next;
                    }
                }
        }
        return input;
}
string Calculation::Parsing(string input)
{
        string output = ""; //Строка для хранения выражения
    Stack<char> operStack; //Стек для хранения операторов
      bool flag = false;
        for (int i = 0; i < input.length(); i++) // Проверка на наличие неизвестных
символов
    {
            if((!IsVariable(input[i])) && (!IsDelimeter(input[i])) &&
(!isdigit(input[i])) && (!IsOperator(input[i])))
                    Error(4);
        }

        for (int i = 0; i < input.length(); i++) //Для каждого символа в входной строке
    {

         //Разделители пропускаем
         if (IsDelimeter(input[i]))
              continue; //Переходим к следующему символу

         //Если символ - цифра, то считываем все число
         if (isdigit(input[i])) //Если цифра
         {
              //Читаем до разделителя или оператора, что бы получить число
              while (!IsDelimeter(input[i]) && !IsOperator(input[i]))
              {
                  output += input[i]; //Добавляем каждую цифру числа к нашей строке
                  i++; //Переходим к следующему символу

                          if (i == input.length()) break; //Если символ - последний,
то выходим из цикла
              }

              output += " "; //Дописываем после числа пробел в строку с выражением
              i--; //Возвращаемся на один символ назад, к символу перед разделителем
         }
         //Если символ - оператор
         if (IsOperator(input[i])) //Если оператор
         {
             if (input[i] == '(') //Если символ - открывающая скобка
                 operStack.push(input[i]); //Записываем её в стек
             else if (input[i] == ')') //Если символ - закрывающая скобка
             {
                 //Выписываем все операторы до открывающей скобки в строку
                 char s = operStack.gettop();
                           operStack.pop();

                 while (s != '(')
                 {
                     output += s;
```

```
                                output += " ";
                    s = operStack.gettop();
                                operStack.pop();
                }
            }else if (input[i] == '|' && !flag) //Если символ - открывающая скобка
                    {
                operStack.push(input[i]); //Записываем её в стек
                            flag = true;
                    }
            else if (input[i] == '|' && flag) //Если символ - закрывающая скобка
            {
                //Выписываем все операторы до открывающей скобки в строку
                char s = operStack.gettop();
                            operStack.pop();

                            while (s != '|')
                {
                    output += s;
                                output += " ";
                    s = operStack.gettop();
                                operStack.pop();
                }
                operStack.push(input[i]); //Если стек пуст, или же приоритет оператора
выше - добавляем операторов на вершину стека
                            flag = false;
            }
            else //Если любой другой оператор
            {
                            if (input.length() == i+1 || (IsOperator(input[i+1]) &&
(input[i+1] != '(') && (input[i+1] != ')') && (input[i+1] != '|') && !((input[i]=='-'
|| input[i]=='+') && (input[i+1]=='-' || input[i+1]=='+')) ))
                                Error(2);
                            if (!operStack.empty()) //Если в стеке есть элементы
                                        if (GetPriority(input[i]) <=
GetPriority(operStack.gettop()) ){ //И если приоритет нашего оператора меньше или
равен приоритету оператора на вершине стека
                                                output += operStack.gettop(); //То
добавляем последний оператор из стека в строку с выражением
                                                output += " ";
                                                operStack.pop();
                                        }
                operStack.push(input[i]); //Если стек пуст, или же приоритет оператора
выше - добавляем операторов на вершину стека
                            while(input.length() > i+1 && ((input[i+1] =='-' ||
input[i+1]=='+') && (operStack.gettop() =='-' || operStack.gettop() =='+')) ) //
Обработчик исключений типа --+++-+-+-
                            {
                                operStack.push(input[i]);
                                i++;
                            }
            }
        }
    }



    //Когда прошли по всем символам, выкидываем из стека все оставшиеся там операторы
в строку
       while (!operStack.empty()){
        output += operStack.gettop();
```

```
                output += " ";
                operStack.pop();

        }
    return output; //Возвращаем выражение в постфиксной записи
}

double Calculation::Counting(string input)
{
        double result = 0; //Результат
    Stack<double> temp; //Временный стек для решения

    for (int i = 0; i < input.length(); i++) //Для каждого символа в строке
    {
        //Если символ - цифра, то читаем все число и записываем на вершину стека
        if (isdigit(input[i]))
        {
            string a = "";

            while (!IsDelimeter(input[i]) && !IsOperator(input[i])) //Пока не
разделитель
            {
                a += input[i]; //Добавляем
                i++;
                            if (i == input.length()) break;
            }
            temp.push(stod(a)); //Записываем в стек
            i--;
        }
        else if (IsOperator(input[i])) //Если символ - оператор
        {
            //Берем два последних значения из стека
            double a = temp.gettop();
                    temp.pop();
                    if(!temp.empty())
                    {
                            double b = temp.gettop();
                            temp.pop();
                            switch (input[i]) //И производим над ними действие,
согласно оператору
                            {
                                    case '+': result = b + a; break;
                                    case '-': result = b - a; break;
                                    case '*': result = b * a; break;
                                    case '/': if(a==0) Error(3); result = b / a;  break;
                                    case '^': result = pow(b,a); break;
                            }
                            temp.push(result); //Результат вычисления записываем
обратно в стек
                    }
                    else if(input[i] == '|')
                            temp.push(abs(a));
                    else if(input[i] == '-')
                            temp.push(a*(-1));
                    else if(input[i] == '+')
                            temp.push(a);
                    else
                            Error(2);
```

```
        }
    }
      if(!temp.empty())
            return temp.gettop(); //Забираем результат всех вычислений из стека и возвращаем его
      else
            Error(1);
}

bool Calculation::IsDelimeter(char c)
{
      string str = " =";
    if ((str.find(c) != -1))
        return true;
    return false;
}
void Calculation::Error(const int key)
{
      string _return;
      switch (key)
      {
            case 1: _return = "ERROR: expression is empty!"; break;
            case 2: _return = "ERROR: expression is not correct!"; break;
            case 3: _return = "ERROR: division by zero is not allowed!"; break;
            case 4: _return = "ERROR: unknow operator!"; break;
            case 5: _return = "ERROR: unknow varibale!"; break;
            default: _return = "ERROR"; break;
      }
      cout << _return << endl;
      throw key;
}
bool Calculation::IsVariable(char c)
{
      string str = "qwertyuiopasdfghjklzxcvbnmQWERTYUIOPASDFGHJKLZXCVBNM";
    if ((str.find(c) != -1))
        return true;
    return false;
}
bool Calculation::IsOperator(char c)
{
      string str = "+-/*^()|";
    if ((str.find(c) != -1))
        return true;
    return false;
}
int Calculation::GetPriority(char s)
{
    switch (s)
    {
        case '|': return 0;
        case '(': return 1;
            case ')': return 2;
        case '+': return 3;
        case '-': return 4;
        case '*': return 5;
        case '/': return 6;
        case '^': return 7;
```

```
        default: return 8;
    }
}
```

Test.cpp

```
#include <gtest/gtest.h>
#include <cstdlib>
#include "Calculation.h"
#include <stack>
#include <string>
#include <cctype>
#include <iostream>
#include <math.h>
TEST(Calculation, throw_empty_expression)
{
	Calculation calculation;
	ASSERT_ANY_THROW(calculation.Calculate(""));
}
TEST(Calculation, throw_empty_expression_only_brackets)
{
	Calculation calculation;
	ASSERT_ANY_THROW(calculation.Calculate("()"));
}
TEST(Calculation, CanConst)
{
	Calculation calculation;
	EXPECT_EQ(calculation.Calculate("1"),1);
}
TEST(Calculation, throw_brackets_is_not_valid)
{
	Calculation calculation;
	ASSERT_ANY_THROW(calculation.Calculate("((1)"));
}
TEST(Calculation, can_variable1)
{
	Calculation calc;
	calc.addVars('x',2);
	EXPECT_EQ(calc.Calculate("x"),2);
}
TEST(Calculation, CanSum)
{
	Calculation calculation;
	EXPECT_EQ(calculation.Calculate("1+1"),2);
}
TEST(Calculation, can_variable2)
{
	Calculation calc;
	calc.addVars('x',2);
	calc.addVars('y',3);
	EXPECT_EQ(calc.Calculate("x+y"),5);
}
TEST(Calculation, CanMult)
{
```

```
	Calculation calculation;
	EXPECT_EQ(calculation.Calculate("2*2"),4);
}
TEST(Calculation, CanSub)
{
	Calculation calculation;
	EXPECT_EQ(calculation.Calculate("3-2"),1);
}
TEST(Calculation, CanDiv)
{
	Calculation calculation;
	EXPECT_EQ(calculation.Calculate("2/2"),1);
}
TEST(Calculation, throw_div_by_zero)
{
	Calculation calculation;
	ASSERT_ANY_THROW(calculation.Calculate("3/0"));
}
TEST(Calculation, CanPow)
{
	Calculation calculation;
	EXPECT_EQ(calculation.Calculate("2^3"),8);
}
TEST(Calculation, CanAbs)
{
	Calculation calculation;
	EXPECT_EQ(calculation.Calculate("|2-4|"),2);
}
TEST(Calculation, throw_operators_is_not_valid_on_middle)
{
	Calculation calculation;
	ASSERT_ANY_THROW(calculation.Calculate("3*+3"));
}
TEST(Calculation, throw_operators_is_not_valid_on_start)
{
	Calculation calculation;
	ASSERT_ANY_THROW(calculation.Calculate("*3"));
}
TEST(Calculation, throw_operators_is_not_valid_on_finish)
{
	Calculation calculation;
	ASSERT_ANY_THROW(calculation.Calculate("3*"));
}
TEST(Calculation, throw_operator_unknow)
{
	Calculation calculation;
	ASSERT_ANY_THROW(calculation.Calculate("3%2"));
}
TEST(Calculation, CanCombo_1)
{
	Calculation calculation;
	EXPECT_EQ(calculation.Calculate("2^(2+1)+3*2"),14);
}
TEST(Calculation, CanCombo_2)
{
	Calculation calculation;
	EXPECT_EQ(calculation.Calculate("(4^2-1+1)/(2+2)^2"),1);
}
```

```
TEST(Calculation, CanCombo_3)
{
	Calculation calculation;
	EXPECT_EQ(calculation.Calculate("(1+3)*(3-1)^2"),16);
}
TEST(Calculation, CanCombo_4)
{
	Calculation calculation;
	EXPECT_EQ(calculation.Calculate("2+2*(3^2)^2"),164);
}
TEST(Calculation, CanCombo_5)
{
	Calculation calculation;
	EXPECT_EQ(calculation.Calculate("15/5 + (2+12)/7"),5);
}
TEST(Calculation, can_unar_op1)
{
	Calculation calculation;
	EXPECT_EQ(calculation.Calculate("-1"),-1);
}
TEST(Calculation, can_unar_op2)
{
	Calculation calculation;
	EXPECT_EQ(calculation.Calculate("+1"),1);
}
TEST(Calculation, can_unar_combo1)
{
	Calculation calculation;
	EXPECT_EQ(calculation.Calculate("-+-1"),1);
}
TEST(Calculation, CanCombo_6)
{
	Calculation calculation;
	EXPECT_EQ(calculation.Calculate("2^3^2"),64);
}



int main(int ac, char* av[])
{
  testing::InitGoogleTest(&ac, av);
  int _return = RUN_ALL_TESTS();
  system("pause");
  return _return;
}
```