Федеральное государственное автономное образовательное учреждение

высшего образования

«Нижегородский государственный университет

им. Н.И. Лобачевского»

Институт информационных технологий математики и механики

Отчет по лабораторной работе

Вычисление арифметических выражений

Выполнил:

Студент института ИТММ

гр. 381703-1

Сизов К.Д

Проверил:

Ассистент кафедры МОСТ

Института ИТММ

Волокитин В.Д

Нижний Новгород

2018 г.

Оглавление

[Введение 3](#_Toc532051446)

[Постановка задачи 4](#_Toc532051447)

[Руководство пользователя 5](#_Toc532051448)

[Руководство программиста 6](#_Toc532051449)

[Описание структуры программы 6](#_Toc532051450)

[Описание структур данных 7](#_Toc532051451)

[Описание алгоритмов 8](#_Toc532051452)

[Заключение 9](#_Toc532051453)

[Литература 10](#_Toc532051454)

[Приложение 1 11](#_Toc532051455)

[Приложение 2 12](#_Toc532051456)

[Приложение 3 13](#_Toc532051457)

# Введение

Во многих отраслевых сферах деятельности в том или ином контексте возникает необходимость точно посчитать результат математического выражения. Например, в бухгалтерской сфере считают прибыль и убыль, а так же различные экономические коэффициенты (например ВВП), в физике, биологии, химии, строительстве, постоянно возникает потребность вычисления различных выражений. Расчеты производятся по формулам, подставляя конкретные числовые значения в определённую формулу мы получаем арифметическое выражение. Понятно, что считать результат в “ручную” или на калькуляторе это не практично, ведь выражение может содержать десятки операций, чисел, функций и переменных. Именно поэтому возникает необходимость создания программы, которая бы предоставляла возможность быстрой работы с арифметическими выражениями. Это программа должна иметь удобный, понятный и привычный синтаксис, и главное быстрое время работы. В данной лабораторной работе предлагается реализация программы данного типа на языке программирования C++. В качестве основного алгоритма вычисления арифметических выражений был реализован алгоритм обратной польской записи.

# Постановка задачи

Разработать программу, выполняющую вычисление арифметического выражения с вещественными числами. Выражение в качестве операндов может содержать переменные и вещественные числа. Допустимые операции известны: +, -, /, \*. Допускается наличие знака "-" в начале выражения или после открывающей скобки. Опционально - наличие математических функций (sin, соs, ln, exp, и т.д.) Программа должна выполнять предварительную проверку корректности выражения и сообщать пользователю вид ошибки и номера символов строки, в которых были найдены ошибки.

# Руководство пользователя

Для того, чтобы вычислить результат выражения, после запуска программы пользователю необходимо ввести в командную строку арифметическое выражение. Выражение должно быть введено с соблюдением синтаксиса и всех правил написания. В случае некорректного ввода программа предложит пользователю повторно произвести ввод выражения. Пользователю разрешается вводить числа, арифметические операции (+,-,\*,/), названия встроенных математических функций (sin, cos, log, exp), названия пользовательских переменных.

* *Числа*. Разрешается ввод как и целых так и дробных чисел. Для дробных чисел в качестве знака разделения целой и дробной части необходимо без пробелов ввести знак '.'.
* *Операция*. Разрешается ввод бинарных математических операций '+','-', '\*', '/'.
* *Функции*. После названия функции (sin, cos, log, exp) необходимо обязательно в круглых в скобках указать значение аргумента. Аргументом функции может вступать любое арифметическое выражение.
* *Переменные*. Название пользовательских переменных в выражении должно начинаться со строчной или заглавной буквы латинского алфавита, н внутри может содержать цифры (например 'a1' корректное название переменной). Не допускается в качестве названия переменной использовать название встроенной математической функции, в противном случае программа распознает такое название как функцию. Значение переменной задается пользователем единожды после ввода арифметического выражения. Допускается повторное использование переменной в других арифметических выражений в рамках одного сеанса работы программы, но не допускается её переопределение.
* *Скобки*. Допускается ввод скобок ‘(‘ и ‘)’. Количество открывающих скобок должно быть равно количеству закрывающих.
* *Унарный минус*. В выражении допускается наличие унарного минуса. Унарный минус может стоять в начале выражения (например '-5+2'), или после открывающейся скобки (например '-5\*(-2+1)'), или после бинарной математической операции (например ‘1/-5’).
* *Первый и последний*. Первый символ не может быть операцией или закрывающей скобкой, а последний не может быть унарным минусом, операцией или открывающей скобкой.
* *Количество скобок*. Количество открывающих скобок должно равняться количеству закрывающих.
* *Порядок следования*. Ниже приведена таблица, которая показывает какие знаки допускается ставить после определенного знака.

Таблица 1 - Правило следования знаков в выражении

|  |  |
| --- | --- |
| **Текущий символ** | **Допустимый следующий символ** |
| Число (или переменная) | Операция или ')' |
| Операция | Число или функция, или '(', или унарный минус |
| '(' | Число или '(', или унарный минус |
| ')' | Операция или ')' |
| Функция | '(' |
| Унарный минус | Число |

# Руководство программиста

## Описание структуры программы

Программа состоит из 3-ех модулей: главная программа (драйвер), Arithmetic, Stack

1. *Главный модуль.* Главный модуль содержит один файл main.cpp. В этом модуле содержится одна функция main(), которая является точкой входа для консольного приложения, которое в свою очередь и является основной программой. В функции размещен цикл, который обеспечивает возможность последовательного ввода арифметических выражений без перезапуска программы. Внутри цикла осуществляется непосредственный вызов функций, которые объявлены в модуле Arithmetic и предназначены для вычисления результата выражения. Помимо этого в цикле содержится конструкция try-catch, которая отвечает за обработку ошибок, возникающих в ходе выполнения обработки и вычисления арифметического выражения.
2. *Модуль Arithmetic.* Модуль Arithmetic состоит из двух файлов. Модуль предназначен для предоставления возможности хранения, обработки, вычисления арифметического выражения. В самом модуле используются стандартные библиотеки <vector>, <queue>, <map>, а так же пользовательская библиотека “stack.h” (см. Модуль Stack). Данные библиотеки нужны для полноценного использования объектов типов vector<T>, map<T>, queue<T>, TStack<T>.
   * *arithmetic.h –* заголовочный файл, содержащий объявление всех функции отвечающих за вычисление результат выражения, помимо этого в файле содержится определение класса Lexems, перечисления enum Token, и глобальной переменной variables.
   * *atithmetic.cpp* – файл с реализацией всех функций отвечающих за вычисление результата арифметического выражение (см. *Приложение 1*). В частности всех функций, которые были объявлены в файле arithmetic.h.
3. *Модуль Stack.* Модуль Stack состоит из одного файла. Модуль предназначен для предоставления возможности хранения и последующей обработки данных в структуре данных Stack. Сама же структура данных Stack необходимо при реализации алгоритмов обработки и вычисления арифметического выражения.
   * *stack.h* – заголовочный файл содержащий в себе объявление шаблонного класса TStack, а также реализацию всех его методов.

## Описание структур данных

1. *Перечисление Token.* Данное перечисление предназначен для хранения типа лексемы. Всего возможны следующие значения: NUMBER, SIN, COS, EXP, LOG, UNARY\_MINUS, LP = ‘(‘, RP = ‘)’, PLUS = '+', MINUS = '-', MUL = '\*', DIV = '/'. Важно отметить, что некоторым значениям этого причисления предписано соответствующие значение типа char, это было сделано с целью упрощения инициализации и возможности приведения типа к char.
2. *Класс Lexems.* Данный класс предназначен для хранения массива лексем. Изначально арифметическое выражение в программе представлено как объект типа string, но работа со строкой крайне неудобна для вычисления выражения, поэтому возникает необходимость создание класса, объект которого рассматривался бы как массив лексем, именно эту задачу и выполняет класс Lexems. Класс имеет два поля с модификатором доступа public (см. *Приложение 1).* Поле tokens представляет из собой массив элементов Token, обращаясь к i-ому элементу этого массива мы можем получить тип i-ой лексемы в арифметическом выражении. Поле values – очередь числовых значений стоящих в арифметическом выражении. При работе с числовым выражением необходимо хранить значения чисел стоящих в нем. Очередь values всегда заполняется числами из арифметического выражения слева направо. Помимо полей, класс содержит один метод – перегрузку оператора сравнения, этот метод необходим для внутреннего тестирования функционала программы.
3. *Шаблонный класс TStack.* Класс переназначен для хранения данных в структуре стек. Стек - тип данных, представляющий собой список элементов, организованных по принципу LIFO (первый вошел – последний вышел). Класс является шаблонным поэтому элементами стека могут быть объекты различных типов данных. Класс имеет 3 поля с модификатором доступа private (см. *Приложение 2*). В поле date хранятся данные которые мы организовываем по принципу стеку, в поле top\_ индекс последнего элемента, в поле maxSize\_ хранится максимальное количество элементов в стеке. Если количество элементов стека достигает максимального значения, происходит пере выделение памяти и максимальное количество элементов для стека увеличивается в два раза. В классе реализован конструктор по умолчанию, конструктор копирования, деструктор. В конструкторе по умолчанию происходит динамическое выделение памяти по maxSize\_ элементов заданного типа. Помимо конструкторов в классе TStack реализованы все необходимые функции для работы с ним, а именно: функция добавления элемента, функция удаления элемента и функция, которая возвращает элемент с вершины стека и удаляет его, а так же функции проверки на полноту и пустоту стека.

## Описание алгоритмов

В программе были реализованы следующие алгоритм:

* *Алгоритм преобразования строки (string) в массив лексем (Lexems)*. Данный алгоритм реализован в функции convertToArrayLexem, которая объявлена в файле arithmetic.h. Алгоритм выглядит следующим образом:
  1. Вход: строка символов.
  2. Создаем результирующий массив лексем Lexems, в который будет записываться результат. i = 0.
  3. Пока i меньше длины строки, получаем i-ый символ строки.
  4. Если i-ый символ – пробельный символ, пропускаем его, инкрементируем i и переходим к шагу 2.
  5. Если i-ый символ – цифра, тогда начиная с i-ой позиции считываем из строки число, добавляем это число в Lexems.values, в Lexems.tokens добавляем NUMBER, инкрементируем i и переходим к шагу 2.
  6. Если i-ый символ – операция или скобка, тогда создаем элемент tok типа Token и инициализируем используя преобразование типов (из char в Token) для i-ого элемента строки. Проверяем является ли i-ый символ строки унарным минусом.
     1. Если i-ый символ – унарный минус, тогда меняем значение tok на UNARY\_MINUS.
     2. Добавляем tok в Lexems.tokens, инкрементируем i и переходим к шагу 2.
  7. Если i-ый символ – буква, тогда считываем буквенно-численную комбинацию символов и записываем их в строковую переменную name.
     1. Если name совпадает с названием математической функции, тогда добавляем в Lexem.tokens соответствующий данной функции элемент типа Token, инкрементируем i и переходим к шагу 2.
     2. Если переменная name есть в глобальном списке, тогда добавляем её значение в Lexems.values, а в Lexems.tokens добавляем NUMBER, инкрементируем i и переходим к шагу 2.
     3. Если переменной name нет в списке глобальных переменных, тогда просим у пользователя задать значение переменной это значение добавляем в Lexems.values. Саму переменную сохраняем в глобальный список. В Lexems.tokens добавляем NUMBER переходим к шагу 2.
  8. В строке не корректный символ – функция выбрасывает исключение с сообщением об ошибке.
  9. Выход: массив лексем.
* *Алгоритм проверки арифметического выражения на корректность*. Данный алгоритм реализован в функции isCorrect, которая объявлена в файле arithmetic.h. Алгоритм выглядит следующем образом.
  1. Вход: массив лексем.
  2. Проверяем на корректность первую и последнею лексему.
  3. Устанавливаем счетчик скобок на ноль (counterBracket = 0) и i = 0.
  4. Пока i + 1 меньше длины массива лексем, получаем i-ый элемент.
  5. Проверяем наличие исключительных (некорректных) случаев (см*. Руководство пользователя*) для этого смотрим на значение i-ого и i+1-ого элемента.
     1. Если была найденна исключтельная (некорректная) последовательность лексем, выбрасываем исключение с сообщение об ошибке.
  6. Если i-ая лексема открывающая скобка, тогда декремнтируем счетчик скобок.
  7. Если i-ая лексема закрывающая скобка, тогда инкрементируем счетчик скобок.
  8. Инкрементируем i и переходим к шагу 3.
  9. Если счетчик скобок не равен 0, тогда выбрасываем исключение с сообщение об ошибке.
* *Алгоритм перевода арифметического выражения в обратную польскую запись.* Данный алгоритм реализован в функции convertToRpn, которая объявлена в файле arithmetic.h. Алгоритм выглядит следующим образом.
  1. Вход: массив лексем.
  2. Создаем результирующий массив лексем, в который будет записываться выражение в ОПЗ (обратная польская запись), создаем пустой стек элементов типа Token, i = 0.
  3. Пока i меньше длины входного массива лексем, получаем i-ый элемент.
  4. Если i-ая лексема является числом, добавляем ее (значение в поле value, тип в tokens) в результирующий массив лексем, инкрементируем i и переходим к шагу 2.
  5. Если i–ая лексема – функция, помещаем соответствующие значение типа Token в стек, инкрементируем i и переходим к шагу 2.
  6. Если i-ая лексема открывающая скобка, помещаем LP в стек, инкрементируем i и переходим к шагу 2.
  7. Если i-ая лексема – закрывающая скобка. Пока верхним элементом стека не станет открывающая скобка, выталкиваем элементы из стека в результирующий массив (в его поле tokens). Удаляем закрывающую скобку, инкрементируем i и переходим к шагу 2.
  8. Если i-ая лексема – бинарная операция. Пока на вершине стека функция или операция с приоритетом выше или равной приоритету i-ой операции, выталкиваем верхний элемент стека в результирующий массив. Помещаем значение типа Token, равное значению i-ой лексемы, в стек, инкрементируем i и переходим к шагу 2.
  9. Выталкиваем из стека все оставшиеся элементы в поле tokens результирующего массива.
  10. Выход: массив лексем – арифметическое выражение записанное в формате ОПЗ.
* *Алгоритм вычисления результата выражения по его обратной польской записи.* Данный алгоритм реализован в функции computeResult, которая объявлена в файле arithmetic.h.
  1. Вход: массив лексем в формате ОПЗ.
  2. Создаем стек для операндов (double), i = 0.
  3. Пока i мешьше длины входного массива лексем, получаем i-ый элемент.
  4. Если i-ая лексема – число, добавляем значение числа в стек, инкрементируем i и переходим к шагу 3.
  5. Если i-ая лексема – функция или унарный минус, тогда достаем с вершины стека число – аргумент, и вычисляем значение функции (умножаем на -1 в случае унарного минуса) от полученного аргумента. Вычисленное значение функции добавляем в стек, инкрементируем i и переходим к шагу 3.
  6. Если i-ая лексема – операция, достаем с вершины стека два операнда, и применяем к ним бинарную операцию, результат выполнения операции кладем в стек, инкрементируем i и переходим к шагу 3.
  7. Выход: число с вершины стека – результат арифметического выражения.
* *Алгоритм вычисления арифметического выражения.* Данной алгоритм суть последовательное применение других алгоритмов и фактическая его реализация присутвует в файле main.cpp главного модуля программы. Алгоритм позволяет по строке с арифметическим выражением вычислить его значение.
  1. Из командной строки получаем строку с арифметическим выражением.
  2. Переводим входную строку в массив лексем.
  3. Полученный массив лексем проверяем на корректность.
     1. Если было поймано исключение переходим к шагу 0.
  4. Переводим корректный массив лексем в массив лексем записанный в обратной польской записи.
  5. По обратной польской записи вычисляем результат
  6. Выход: результат арифметического выражения.

# Заключение

По результату выполнения лабораторной работы была написана программа, которая вычисляет результат арифметического выражения, задаваемого пользователем. Программа удовлетворяет постановке задачи и работает с соблюдением всех критериев. Были реализованы все необходимые алгоритмы, включая алгоритм записи арифметического выражения в формате обратной польской записи и алгоритм вычисления результата выражения по его польской записи. Так же был реализован шаблонный класс TStack, который реализует структуры данных стек, которая в свою очередь была необходима для реализации алгоритмов вычисления выражения. Весь функционал программы, включая функционал класса TStack, был протестирован с помощью библиотеки для модульного тестирования Google C++ Testing Framework.

# Литература

1. Бьёрн Страуструп. Язык программирования C++ — 3-е изд. — СПб.; М.: Невский диалект — Бином, 1999. — 991 с.
2. Обратная польская нотация

[http://algolist.manual.ru/maths/misc/revpn.php]

# Приложение 1

**arithmetic.h**

int getPriority(const Token& tok); // функция возвращает приоритет операции

double stringToDouble(string str, size\_t\* pos); // функция начиная с позиции pos извлекает из строки str число

Lexems convertToArrayLexem(string str); // функция преобразовывает строку в массив лексем

bool isCorrect(const Lexems& lexemsArray); // функция проверяет арифметическое выражение на корректность

Lexems convertToRpn(const Lexems& lexemArray); // функция преобразует массив лексем в обратную польскую запись

double computeResult(const Lexems& rpn); // функция вычисляет результат выражения по обратной пользой записи

# Приложение 2

**arithmetic.h**

class Lexems

{

public:

queue<double> values;

vector<Token> tokens;

bool operator==(const Lexems& lex) const ;

};

# Приложение 3

**stack.h**

template <class ValType>

class TStack

{

private:

ValType\* data; // массив элементов стека

int top\_; // индекс последнего элемента

int maxSize\_; // максимальное количество элементов

public:

TStack(int maxSize); // конструктор инициализации

TStack(); // конструктор по умолчанию

TStack(const TStack& s); // конструктор копирования

~TStack(); // деструктор

bool isFull(); // проверка на полноту

bool isEmpty(); // проверка на пустоту

void push(const ValType& v); // добавить в конец

ValType pop(); // удалить с конца

ValType getTop(); // вернуть с вершины и удалить

};