МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

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

**(ННГУ)**

**Институт информационных технологий, математики и механики**

Направление подготовки: Прикладная математика и информатика

Профиль подготовки: Общий профиль

**ОТЧЕТ**

по

лабораторной работе

на тему:

**«Вычисление арифметических выражений»**

**Выполнил(а):** студент(ка) группы 3821Б1ПМ1

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Соболь Л.А.

Подпись

**Научный руководитель:**

Преподаватель, к.ф.-м.н.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Подпись

Нижний Новгород  
2022

Оглавление:

[Введение 3](#_Toc122346206)

[1. Постановка учебно – практической задачи 4](#_Toc122346207)

[2. Руководство пользователя 5](#_Toc122346208)

[3. Руководство программиста 6](#_Toc122346209)

[3.1. Лексический анализ (токенизация) 6](#_Toc122346210)

[3.2. Преобразование в обратную польскую запись (постфиксная) 7](#_Toc122346211)

[3.3. Вычисление выражения 8](#_Toc122346212)

[3.4. Реализация Лабораторной работы 9](#_Toc122346213)

[Класс Stack 9](#_Toc122346214)

[Класс TPostfix 9](#_Toc122346215)

[Заключение 10](#_Toc122346216)

[Список литературы 11](#_Toc122346217)

[Приложение 12](#_Toc122346218)

[Приложение 1 12](#_Toc122346219)

[Приложение 2 13](#_Toc122346220)

[Приложение 3 19](#_Toc122346221)

# Введение

Вычисление арифметический выражений очень важная вещь в современном мире. Невозможно представить работу человека без калькулятора. Решение проблемы, на первый взгляд, не совсем очевидно. За всю история программирования была создано несколько способов представления выражений в удобный вид и дальнейший подсчёт. В ходе этой работы будет разобран самый удобный и понятный способ, с помощью алгоритма обратной Польской нотации.

Работа прежде всего будет интересна программистам, так как не содержит интерфейса для пользователя. Но даже взаимодействие с консолью удобнее, чем вводить цифры в домашний калькулятор, так как здесь присутствует приоритет операций и возможность пользоваться скобками и переменными.

Работа выполнена с помощью языка C++. В дальнейшем написанное можно дополнять, улучшать и использовать для себя, как личного помощника.

## Постановка учебно – практической задачи

Разработать класс для работы с арифметическими выражениями

Класс Арифметическое Выражение должен представлять следующие операции

- Получение, хранение и возврат исходной (инфиксной) формы

- Преобразование в постфиксную форму

- Возврат постфиксной формы

- Вычисление арифметического выражения при заданных значениях операндов

Условия и ограничения

- Операнды обозначаются буквами латинского алфавита или цифрами

- Исходное арифметическое выражение синтаксически корректно

- Список операций включает: + (сложение), - (вычитание), \* (умножение), / (деление)

- В выражении могут использоваться скобки вида ()

- Значение операндов представляет собой вещественные числа

## Руководство пользователя

Для работы с программой было бы неплохо знать C++. Но даже если знаний нет, то можно запустить проект и воспользоваться базовой работой с консолью.

string expression;

double res;

setlocale(LC\_ALL, "Russian");

cout << "Введите арифметическое выражение: ";

cin >> expression;

cout << expression << endl;

TPostfix postfix(expression);

cout << "Арифметическое выражение: " << postfix.GetInfix() << endl;

postfix.ToPostfix();

cout << "Постфиксная форма: " << postfix.GetPostfix() << endl;

res = postfix.Calculate(cin,cout);

cout << res << endl;

Это код, который стоит по умолчанию. Пользователя просят ввести арифметическое выражение. Далее выводится инфиксная и постфиксная запись. Функция Calculate() попросит ввести значения переменных, а затем произведет подсчет. Итог в конце.

ФОТО

Продублируем ограничения, наложенные на пользователя, чтобы программа работала корректно.

Условия и ограничения

- Операнды обозначаются буквами латинского алфавита или цифрами

- Исходное арифметическое выражение синтаксически корректно

- Список операций включает: + (сложение), - (вычитание), \* (умножение), / (деление)

- В выражении могут использоваться скобки вида ()

- Значение операндов представляет собой вещественные числа

## Руководство программиста

### Лексический анализ (токенизация)

Для реализации лексического анализа используется конечный автомат. Благодаря маленькому количеству операций, его можно не расписывать строго. Далее будут описаны правила введения выражения и простая таблица, по которой можно восстановить автомат при желании.

Арифметическое выражение, состоит из чисел, переменных, операндов и скобок.

-Пробелы не обрабатываются.

**-**Числа имеют вид:

<знак><целая часть>< . ><дробная часть>

Вариант <знак>< . ><дробная часть> считается ложным.

Знаком является только “-“, его может не быть.

Дробная и целая часть – набор цифр.

-Допустимые операнды: +, -, \*, /

Переменные состоят из букв латинского алфавита, символов “\_” . Имя переменной не может содержать цифры.

Допускается использование только круглых скобок. Количество открытых и закрытых скобок одинаково. Они должны так, чтобы выражение было логически верным.

Все правила составления арифметических выражений можно легко представить в виде таблицы, по которой построена программа.

|  |  |
| --- | --- |
| **Лексема** | **Следующая лексема** |
| Операнд | Операция, ) |
| Операция | Операнд, ( |
| ( | Число, ( |
| ) | Операция, ) |

*Рисунок 1. Правило написания арифметического выражения.*

Так, если после лексемы идет та, которой нет в таблицы, программа будет выдавать ошибку. "Invalid expression"

### Преобразование в обратную польскую запись (постфиксная)

После выполнения функции Parse(), программа проверила и составила набор лексем (инфиксную запись). Данная запись не дает возможности произвести подсчет выражения.

Рассмотрим алгоритм преобразования в обратную Польскую запись. Это форма записи [математических](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%82%D0%B5%D0%BC%D0%B0%D1%82%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%BE%D0%B5_%D0%B2%D1%8B%D1%80%D0%B0%D0%B6%D0%B5%D0%BD%D0%B8%D0%B5) и [логических](https://ru.wikipedia.org/wiki/%D0%9B%D0%BE%D0%B3%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%BE%D0%B5_%D0%B2%D1%8B%D1%80%D0%B0%D0%B6%D0%B5%D0%BD%D0%B8%D0%B5) 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)) польского логика Я. Лукасевича (1958), который изучал свойства этой записи. Обратная польская запись имеет ряд преимуществ перед инфиксной записью при выражении алгебраических формул. Во-первых, любая формула может быть выражена без скобок. Во-вторых, она [удобна для вычисления формул в машинах со стеками](http://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%80%D0%B0%D1%82%D0%BD%D0%B0%D1%8F_%D0%BF%D0%BE%D0%BB%D1%8C%D1%81%D0%BA%D0%B0%D1%8F_%D0%B7%D0%B0%D0%BF%D0%B8%D1%81%D1%8C#.D0.92.D1.8B.D1.87.D0.B8.D1.81.D0.BB.D0.B5.D0.BD.D0.B8.D1.8F_.D0.BD.D0.B0_.D1.81.D1.82.D0.B5.D0.BA.D0.B5). В-третьих, инфиксные операторы имеют приоритеты, которые произвольны и нежелательны. Например, мы знаем, что ab+c значит (ab)+c, а не a(b+c), поскольку произвольно было определено, что умножение имеет приоритет над сложением. Но имеет ли приоритет сдвиг влево перед операцией И? Кто знает? Обратная польская запись позволяет устранить такие недоразумения. Правда в нашей работе третье преимущество рассматриваться не будет.

Рассмотрим реализацию алгоритма ниже. В программе использовался стек.

*Цикл по массиву лексем*

*Если лексема R это*

1. *Число( операнд): R -> Res*
2. *“(“ : R -> Stack*
3. *“)” : Проход по элементам (Element) стека пока не “(“:*

*Element -> R1 ;R1 -> Res*

1. *Операция: Проход по элементам (Element) стека пока не “(“:*

*Если приоритет (R) <= приоритет (Element):*

*Element -> Res*

*R -> Stack*

*Elem of Stack -> Res*

Так, выражение 12\*(3+4 \*5) -7)) будет преобразовано в 12 3 4 5 \* + 7 - \*.

Далее именно эта запись будет использоваться для подсчета выражения.

### Вычисление выражения

После функции ToPostfix() мы имеем набор лексем в наиболее удобной записи.

Проходимся по массиву этих лексем. Если нашли операцию, то считываем два оператора до неё и выполняем операцию. На их место ставим результат вычисления. Из-за того, что все процессы выполняются стеком, это сделать очень легко. Берем сверху, считаем, кидаем обратно наверх. После считывания всех лексем, результат выражения лежит в стеке.

12 3 4 5 \* + 7 - \*.

1. 4 5 \* 4\*5=20 новая запись 12 3 20 + 7 - \*
2. 3 20 + 3+20=23 новая запись 12 23 7 - \*
3. 23 7 - 23-7=16 новая запись 12 16 \*
4. 12\*16=192 в стеке 192

Исходное выражение 12\*(3+4 \*5) -7)). Ответ 192 верен.

### Реализация Лабораторной работы

Вся реализация выполнена в классе TPostfix. Так же для работы необходим стек, работающий на динамической памяти. Несколько слов про них ниже.

#### Класс Stack

Класс реализует базовый принцип стека First in Last out. Содержит все операции необходимые для работы с лабораторной. Можно взять и положить элемент в стек.

#### Класс TPostfix

Структура класса выглядит следующим образом:

string input\_expression

lexemType //тип лексем

vector<pair<lexemType, string>> infix; //набор лексем со значениями

vector<pair<lexemType, string>> postfix; // постфиксная запись

map<string, int> priority; //приоритет операций

map<string, double> operands; //набор переменных(variable) со значениями (по умолчанию в них нули)

Основной набор операций:

void Parse(); //разложить на лексемы

void ToPostfix();//сделать обратную польскую запись для подсчета

void GetValues(istream& input, ostream& output);

double Calculate(istream& input, ostream& output);// Ввод переменных, вычисление по постфиксной форме

Так же класс содержит конструктор, где идет считывание арифметического выражения. И два метода GetInfix(), GetPostfix(). Они выполняют возврат инфиксной и постфиксной записи соответсвенно.

Все методы можно подробнее изучить в приложении к работе.

# Заключение

Была разработана программа на языке C++ для вычисления арифметических выражений. Уникальность работы, в том, что выражения пользователь может вводить с приоритетом операций и переменными. Так же изучен алгоритм обратной польской записи. Вся реализация включена в классе TPostfix, который в дальнейшем можно дополнить и подключить в свои проекты.

# Список литературы

1. Hamblin C. L. Language and the Theory of Information : дис. – London School of Economics and Political Science (University of London), 1957.

# Приложение

## Приложение 1

|  |
| --- |
| stack.h – заголовочный файл класса Stack.h |
| #ifndef \_\_STACK\_H\_\_  #define \_\_STACK\_H\_\_  #include <vector>  #include <cstddef>  #include <stdexcept>  template <typename T>  class Stack  {  std::vector<T> mem;  size\_t sz; // кол-во элементов  public:  Stack()  {  sz = 0;  }  Stack(size\_t size) : Stack()  {  mem.reserve(size);  }  Stack(size\_t size, const T\* arr) : Stack(size)  {  for (int i = 0; i < size; i++)  mem.push\_back(arr[i]);  sz = size;  }  size\_t size() const noexcept  {  return sz;  }  bool empty()  {  return sz == 0;  }  void push(const T& x)  {  mem.push\_back(x);  sz++;  }  T top()  {  if(empty())  throw std::out\_of\_range("Can't get top from empty stack");  return mem[sz - 1];  }  void pop()  {  if (empty())  throw std::out\_of\_range("Can't do pop on empty stack");  mem.pop\_back();  sz--;  }  bool operator==(const Stack& st)  {  if (this == &st)  return true;  return st.mem == mem;  }  };  #endif |

## Приложение 2

|  |
| --- |
| posfix.h – заголовочный файл класса ArithmeticExpression |
| #ifndef \_\_POSTFIX\_H\_\_  #define \_\_POSTFIX\_H\_\_  #include <iostream>  #include <string>  #include <map>  #include "stack.h"  using namespace std;  class ArithmeticExpression {  // приоритет операций  static map<char, int> priority; // все разрешенные операции односимвольные  // типы лексем  enum lType {number, variable, operation, begin, end, null}; // begin и end - открывающая и закрывающая скобка  string text;  vector<pair<lType, string>> infix; // набор пар (тип\_лексемы, текст\_лексемы)  vector<pair<lType, string>> postfix; // набор пар (тип\_лексемы, текст\_лексемы)  map<string, double> operands;  // проверка символов  static bool isDigit(char c); // 0 ... 9  static bool isLetter(char c); // a ... z, A ... Z, \_  static bool isOperation(char c); // +, -, \*, /  static bool isMinus(char c); // -  static bool isPoint(char c); // -  static bool isBegin(char c); // (  static bool isEnd(char c); // )  static void deleteAll(string& str, char toDelete) {  int spaces = 0;  for(int i = 0; i < str.size(); i++) {  str[i - spaces] = str[i];  if(str[i] == toDelete)  spaces++;  }  str.erase(str.size() - spaces, spaces);  }  void parse(); // текст -> набор лексем  void toPostfix();  void readOperands(istream& input, ostream& output);  public:  explicit ArithmeticExpression(const string& text);  string getInfix() const { return text; }  string getPostfix() const {  string postfixStr;  for(auto& p: postfix) {  postfixStr += p.second;  }  return postfixStr;  }  double calculate(istream& input = cin, ostream& output = cout); // Ввод переменных, вычисление по постфиксной форме  };  #endif |

|  |
| --- |
| postfix.cpp – реализация класса ArithmeticExpression |
| #include "postfix.h"  map<char, int> ArithmeticExpression::priority = { {'+', 1}, {'-', 1}, {'\*', 2}, {'/', 2} };  ArithmeticExpression::ArithmeticExpression(const string& text)  {  if(text.empty())  throw invalid\_argument("Creating arithmetic expression from an empty string");  this->text = text;  deleteAll(this->text, ' ');  parse();  toPostfix();  }  void ArithmeticExpression::parse()  {  lType t = null; // тип текущей лексемы  int b = 0; // индекс начала текущей лексемы  int allBracketsAreClosed = 0;  for(int i = 0; i < text.size(); i++) {;  char c = text[i];  if(c == ' ')  continue;  switch(t) {  case (number):  if(isPoint(c) || isDigit(c))  continue;  // считывание числа окончено  infix.emplace\_back(number, text.substr(b, i - b));  if(isEnd(c)) {  b = i;  t = end;  } else if(isOperation(c)) {  b = i;  t = operation;  } else {  throw invalid\_argument("Invalid expression");  }  break;  case (variable):  if(isLetter(c) || isDigit(c))  continue;  // считывание переменной окончено  infix.emplace\_back(variable, text.substr(b, i - b));  operands.insert({text.substr(b,i - b), 0.0});  if(isEnd(c)) {  b = i;  t = end;  } else if(isOperation(c)) {  b = i;  t = operation;  } else {  throw invalid\_argument("Invalid expression");  }  break;  case (operation):  // считывание операции окончено  infix.emplace\_back(operation, text.substr(b, i - b));  if(isBegin(c)) {  b = i;  t = begin;  } else if(isDigit(c) || isPoint(c) || isMinus(c)) {  b = i;  t = number;  } else if(isLetter(c)) {  b = i;  t = variable;  } else {  throw invalid\_argument("Invalid expression");  }  break;  case (begin):  // считывание открывающей скобки окончено  allBracketsAreClosed++;  infix.emplace\_back(begin, text.substr(b, i - b));  if(isBegin(c)) {  b = i;  t = begin;  } else if(isDigit(c) || isPoint(c) || isMinus(c)) {  b = i;  t = number;  } else if(isLetter(c)) {  b = i;  t = variable;  } else {  throw invalid\_argument("Invalid expression");  }  break;  case (end):  // считывание закрывающей скобки окончено  allBracketsAreClosed--;  infix.emplace\_back(end, text.substr(b, i - b));  if(isEnd(c)) {  b = i;  t = end;  } else if(isOperation(c)) {  b = i;  t = operation;  } else {  throw invalid\_argument("Invalid expression");  }  break;  case (null):  // начало выражения  if(isBegin(c)) {  b = i;  t = begin;  } else if(isDigit(c) || isPoint(c) || isMinus(c)) {  b = i;  t = number;  } else if(isLetter(c)) {  b = i;  t = variable;  } else {  throw invalid\_argument("Invalid expression");  }  break;  }  }  infix.emplace\_back(t, text.substr(b, text.size() - b));  if(t == variable)  operands.insert({text.substr(b,text.size() - b), 0.0});  if(t == end)  allBracketsAreClosed--;  if(t == begin)  allBracketsAreClosed--;  if(allBracketsAreClosed != 0)  throw invalid\_argument("Invalid expression: troubles with brackets");  if((t != end && t != variable && t != number)) {  throw invalid\_argument("Invalid expression: invalid ending");  }  } // текст -> набор лексем  void ArithmeticExpression::toPostfix()  {  Stack<pair<lType, string>> st;  pair<lType, string> stackItem;  for (auto& lexem : infix) {  switch (lexem.first) {  case begin:  st.push(lexem);  break;  case end:  stackItem = st.top();  st.pop();  while (stackItem.first != begin) {  postfix.emplace\_back(stackItem);  stackItem = st.top();  st.pop();  }  break;  case operation:  while (!st.empty()) {  stackItem = st.top();  st.pop();  if (priority[lexem.second[0]] <= priority[stackItem.second[0]])  postfix.emplace\_back(stackItem);  else {  st.push(stackItem);  break;  }  }  st.push(lexem);  break;  default:  postfix.emplace\_back(lexem);  }  }  while (!st.empty()) {  stackItem = st.top();  st.pop();  postfix.emplace\_back(stackItem);  }  }  double ArithmeticExpression::calculate(istream& input, ostream& output)  {  readOperands(input, output);  double left, right; // операнды  Stack<double> st;  for(auto& lexem: postfix) {  switch (lexem.second[0]) {  case '+':  right = st.top();  st.pop();  left = st.top();  st.pop();  st.push(left + right);  break;  case '-':  right = st.top();  st.pop();  left = st.top();  st.pop();  st.push(left - right);  break;  case '\*':  right = st.top();  st.pop();  left = st.top();  st.pop();  st.push(left \* right);  break;  case '/':  right = st.top();  st.pop();  left = st.top();  st.pop();  st.push(left / right);  break;  default:  if(lexem.first == variable)  st.push(operands[lexem.second]);  else  st.push(stod(lexem.second));  }  }  return st.top();  }  void ArithmeticExpression::readOperands(istream& input, ostream& output) {  output << "Enter values:"<< endl;  for(auto& o: operands) {  output << o.first << " = ";  input >> o.second;  }  }  bool ArithmeticExpression::isDigit(char c) {  return '0' <= c && c <= '9';  }  bool ArithmeticExpression::isLetter(char c) {  return 'a' <= c && c <= 'z' || 'A' <= c && c <= 'Z' || c == '\_';  }  bool ArithmeticExpression::isOperation(char c) {  return c == '+' || c == '-' || c == '\*' || c == '/';  }  bool ArithmeticExpression::isMinus(char c) {  return c == '-';  }  bool ArithmeticExpression::isBegin(char c) {  return c == '(';  }  bool ArithmeticExpression::isEnd(char c) {  return c == ')';  }  bool ArithmeticExpression::isPoint(char c) {  return c == '.';  } |

## Приложение 3

Ниже представлены примеры использования класса ArithmeticExpression, вместе с изображениями диалогового окна.

Пример 1:

|  |
| --- |
| #include <iostream>  #include <string>  #include "postfix.h"  using namespace std;  int main()  {  string str;  setlocale(LC\_ALL, "Russian");  cout << "Введите арифметическое выражение: ";  cin >> str;  cout << "Вы ввели: " << str << endl;  ArithmeticExpression expression(str);  cout << "Обратная польская запись: "<< expression.getPostfix() << endl;  double res = expression.calculate();  cout << "Результат: "<< res << endl;  return 0;  } |
|  |

Пример 2:

|  |
| --- |
| #include <iostream>  #include <string>  #include "postfix.h"  using namespace std;  int main()  {  setlocale(LC\_ALL, "Russian");  string str = "2 \* x + 3 \* y";  ArithmeticExpression expression(str);  for(int i = 0; i < 3; i++)  cout << expression.calculate() << endl;    return 0;  } |
|  |