**ОТЧЕТ**

по

лабораторной работе

на тему:

**«Вычисление арифметических выражений»**

Выполнил: студент группы 3821Б1ПМ1

**Иванченко А.М.**

**Оглавление:**

[Введение 3](#_Toc1)

[1. Трансляция и вычисление арифметических выражений 4](#_Toc2)

[1.1. Лексический анализ 4](#_Toc3)

[1.2. Преобразование в обратную польскую запись 5](#_Toc4)

[1.3. Вычисление выражения 6](#_Toc5)

[2. Реализация трансляции и вычисления выражений 7](#_Toc6)

[2.1. Класс ArithmeticExpression 7](#_Toc7)

[2.2. Руководство пользователя 7](#_Toc8)

[Заключение 10](#_Toc9)

[Список литературы 11](#_Toc10)

# Введение

За долгую историю развития математики была разработана математическая нотация, которая удобна для понимания и воспроизведения человеку. С тех пор, как электронные вычислительные машины стали повсеместно использоваться для разного рода вычислений, актуальным вопросом стало представление математических (в частности арифметических) выражений на понятном им языке – языке программирования.

Такая задача может возникнуть, например, при создании программы-калькулятора. Известны способы программного решения многих математических задач: это вычисление простейших арифметических выражений, численное интегрирование и дифференцирование и многое другое. Для использования всех этих алгоритмов необходимо реализовать интерфейс программы. Причем желательно, чтобы пользователь мог вводить задачу в привычной ему математической нотации.

Мы не будем рассматривать выражения, возникающие при решения всех подобных математических задач, а остановимся на работе с арифметическими выражениями. В частности, реализуем для языка программирования с++ ввод арифметических выражений, их хранение и вычисление, при этому учитывая собственный опыт в работе с подобными программами и оптимизируя нашу программу под наиболее часто встречающиеся задачи.

## Трансляция и вычисление арифметических выражений

В этой главе описывается метод трансляции вводимых пользователем арифметических выражений в наиболее удобную для вычисления запись.

### Лексический анализ

Стандартной реализацией лексического анализа является использование конечного автомата, определяемого регулярными выражениями. В силу простоты задачи не будем строго определять формальный язык арифметических выражений и ограничимся ссылкой на общепринятую нотацию, на основе которой и построим конечный автомат.

На вход алгоритма подается арифметическое выражение, состоящее из **чисел**, **переменных** (значения которых будут сообщаются при вычислении выражения), **операндов** и **скобок**. Пробелы не несут смысла и не обрабатываются.

**Числа** имеют вид: <знак><целая часть>< . ><дробная часть>

Допускается только знак ‘-’. Знак может быть опущен. Целая часть и дробная часть состоят из цифр. Целая часть и дробная часть могут быть опущены. Точка может быть опущена, если указана целая часть, а дробная часть равна нулю. Случай ввода нескольких точек в одном числе рассматривать не будем, он будет вызывать неопределенное поведение (подробнее в главе 2).

Допустимые **операнды**: +, -, \*, /

**Переменные** состоят из букв латинского алфавита, символа ‘\_’ и цифр. Имя переменной не может начинаться с цифры.

Допускается использование скобок ‘(‘, ‘)’.

Установив такие ограничения, построим конечный автомат для разбиения текста на лексемы. Его задача – для каждого вводимого символа определить тип лексемы, основываясь на информации о типе предыдущего символа. Такая постановка задачи (анализ только одного предшествующего символа) хоть и сужает возможности, но все же обеспечивает определенный выше функционал. Изобразим состояния и переходы между ними в виде графа (рис. 1). Переход между состояний обуславливается очередным поданным символом, соответствующие символы подписаны у ребер графа.

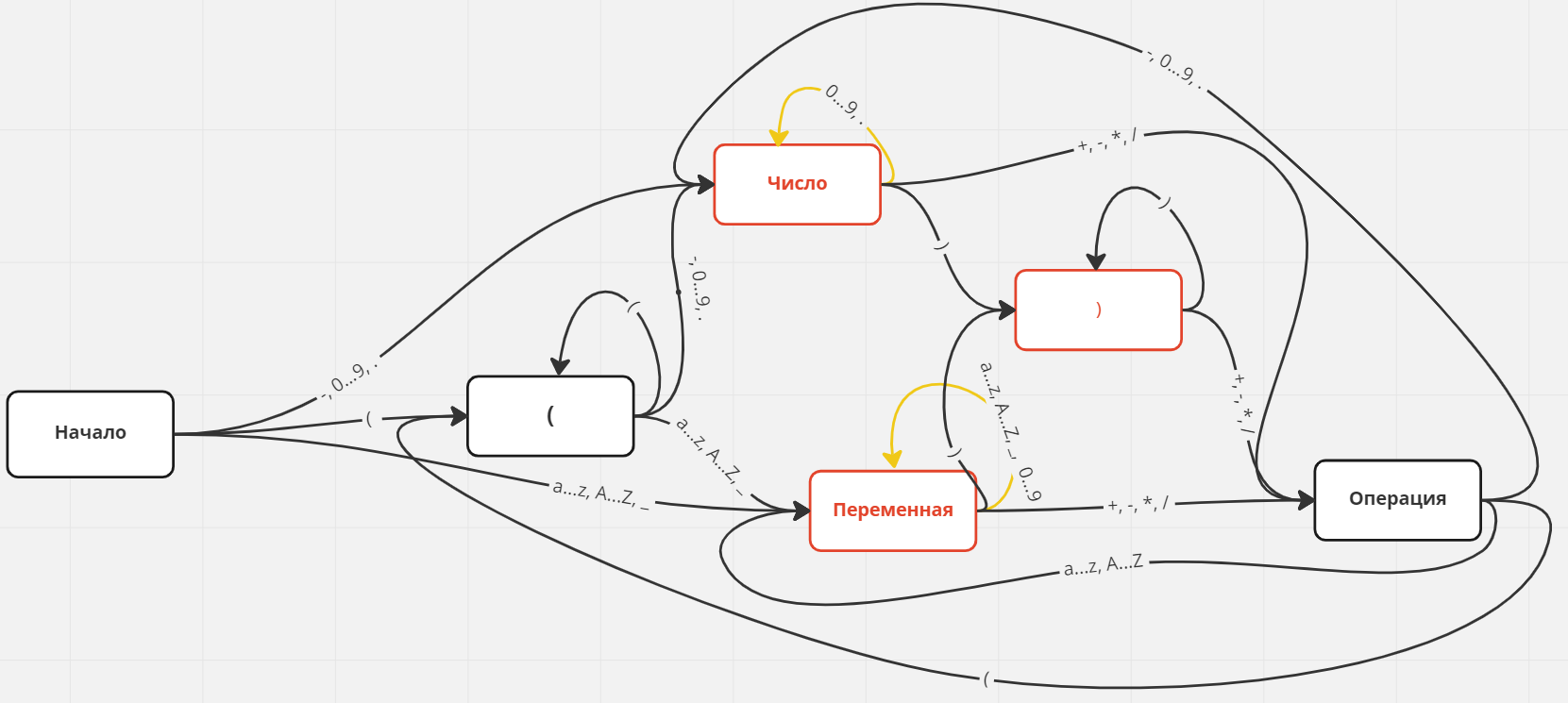


Рисунок 1 – граф состояний и переходов конечного автомата, определяющего тип лексемы очередного символа на основе информации о типе предыдущего символа. Красным помечены конечные состояния. Желтым помечены переходы, при которых происходит накопление символов.

Таким образом, алгоритм действий прост: следуя по графу состояний, копим поступающие символы, если переходим по петле число-число или переменная-переменная (отмечены на графе жёлтым). При нашей постановке задачи только числа и имена переменных могут состоять более чем из одного символа. При переходе по другим ребрам накопленные к моменту перехода добавляем в последовательность лексем и помечаем соответствующим типом.

Если алгоритм закончился в состоянии, не являющимся конечным – введенное выражение некорректно. Также проверим, что открывающих и закрывающих скобок одинаковое количество. Эти две проверки вместе с запретом на недопустимые переходы между состояниями обнаруживают большинство некорректных выражений. Остальные ошибочные выражения будут вызывать исключения на этапе вычисления – это не лучшая практика, но вполне допустимая в нашем случае, когда выражение вводится именно для вычислений и после инициализации обязательно последует вычисление.

Стоит заметить, что время выполнения алгоритма зависит от количества введенных символов линейно. Очевидно, что лучшей асимптотики добиться невозможно, и для лучшей производительности на следующих этапах необходимо также уложиться в линейное время.

### Преобразование в обратную польскую запись

После действий, описанных в предыдущем параграфе, входные данные представляют собой набор лексем, расположенных друг за другом в порядке ввода. Стандартной для математических выражений является инфиксная запись – её мы и ожидаем от пользователя. Но для вычисления выражения более удобна постфиксная запись.

Стандартным решением для вычисления арифметических выражений является обратная польская нотация, которая была разработана австралийским философом и специалистом в области теории вычислительных машин Чарльзом Хэмблином в 1957[1] на основе польской нотации, которая была предложена в 1920 году польским математиком Яном Лукасевичем. Особенностью записи является то, что знак операции всегда расположен после операндов. Этот факт в дальнейшем позволит нам вычислять выражение за линейное время, поэтому требуется осуществить преобразование из инфиксной в обратную польскую запись.

У этой проблемы также есть простое стандартное решение, алгоритм которого можно записать следующим образом (рис. 2)

|  |
| --- |
| Для каждой лексемы ***lex***:  если ***lex*** число:  ***lex*** -> ***результат***  если lex '(':  lex -> стек  если lex ')':  По элементам стека elem пока не '(':  elem -> результат  если lex операция:  По элементам стека elem пока не ')':  если приоритет(lex) <= приоритет(elem):  elem -> результат  стек -> результат |

Рисунок 2 – алгоритм построения обратной польской записи из инфиксой записи

Стоит заметить, что предложенный алгоритм работает за линейное время, то есть достигнутая в прошлом параграфе асимптотика не испорчена.

### Вычисление выражения

Теперь, имея в качестве входных данных набор лексем в обратной польской записи, мы можем с легкостью вычислить выражение за один проход по набору. Алгоритм прост и заключается в следующем: запоминаем операнды, как только встречаем операцию, выполняем её для двух предыдущих операндов. Эти два операнда заменяем на результат операции. Если хранить операнды в стеке, реализация алгоритма тривиальна.

Заметим, что время вычисления зависит линейно от длины выражения, а значит обе операции – создание объекта “арифметическое выражение” и его вычисление выполняются за линейное время. Ещё раз отметим, что лучшей асимптотики в такой задаче добиться невозможно.

## Реализация трансляции и вычисления выражений

Для трансляции и вычисления арифметических выражений была написана программа на языке программирования c++. Для представления арифметических выражений был разработан класс ArithmeticExpression. Он отвечает как за трансляцию, так и за вычисление выражений. Для реализации этих функций необходима структура данных стек, поэтому был создан класс Stack, реализующий базовые функции стека.

### Класс Stack

Класс представляет собой стек, реализованный на динамическом массиве с возможностью перевыделения памяти. Класс имеет базовые методы, отвечающие за добавление элемента в верх стека, взятия верхнего элемента и удаление верхнего элемента. Полная реализация доступна в приложении 1.

### Класс ArithmeticExpression

Не будем останавливаться на реализации методов класса, так как алгоритмы были описаны в главе 1. Вместо этого кратко опишем интерфейс класса и возможности его использования. Все публичные методы класса представлены на рисунке 4.

|  |
| --- |
| explicit ArithmeticExpression(const string& text);  string getInfix() const;  string getPostfix() const;  double calculate(istream& input = cin, ostream& output = cout); |

Рисунок 4 – публичные методы класса ArithmeticExpression

В конструктор класса передается строка, в которой находится инфиксная запись арифметического выражения. Конструктор может выкинуть исключение, если строка представляет некорректное выражение.

Методы getInfix() и getPostfix() возвращают строку, которая представляет собой инфиксную и обратную польскую запись данного выражения соответственно.

Метод calculate() используется для вычисления значения выражения. При его вызове значение переменных, если таковые присутствуют в выражении, будет введено из потока input (по умолчанию std::cin), сообщения пользователю будут выводиться в поток output (по умолчанию std::cout). Один раз созданное выражение может быть вычислено несколько раз с разными значениями переменных с помощью вызова данного метода.

Полная реализация класса доступна в приложении 2.

## Руководство пользователя

Для использования транслятора арифметических выражений необходимо импортировать <postfix.h>.

Создайте объект класса ArithmeticExpression, передав в качестве аргумента строку, содержащую арифметическое выражение в стандартной математической нотации(рис. 5). Строка может содержать:

* Сколько угодно знаков пробела.
* Положительные или отрицательные целые числа или десятичные дроби. Унарный плюс в записи положительных чисел не допускается (можно: “1”, “-1”, нельзя: “+1”). Десятичные дроби записываются с использованием точки; можно опускать целую или дробную часть (но не обе сразу), сохраняя точку (можно: “0.0”, “0.”, “.0”, нельзя: “.” ).
* Имена переменных, состоящие из букв латинского алфавита, символа нижнего подчеркивания “\_” и цифр. Имя переменной не может начинаться с цифры.
* Знаки бинарных операций “ + ”, “ - “, “ \* ”, “ / ”.
* Открывающие и закрывающие скобки “ ( “, “ ) ”.

Создав объект, вы можете получить строку с представлением выражения в инфиксой и постфиксной записях, вызвав методы getInfix() и getPostfix() соответственно. Обратите внимание, что инфиксная запись может отличаться от строки, переданной вами в конструктор количеством знаков пробела.

Чтобы вычислить значение арифметического выражения, вызовите метод calculate(). Следуя появившейся в консоли инструкции, введите значения переменных, которые присутствуют в вашем выражении.

Если на этапе создания объекта или вычисления была выброшена ошибка, ваше выражение некорректно и не соответствует вышеописанным правилам. Убедитесь, что:

* Всем открытым скобкам соответствуют закрытые скобки
* Вы используете допустимые операции явным образом: “2\*x” – можно, “2x” – нельзя.
* Вместо унарного минуса “-a” используется умножение “(-1)\*a”
* Переменные названы допустимым образом (см. выше)
* Для записи десятичной дроби используется точка, а не запятая

Ниже показан пример использования описанного выше функционала: код на рисунке 5 и вывод консоли на рисунке 6.

|  |
| --- |
| string str = "2\*x + 3\*y";    ArithmeticExpression expression(str);  cout << "Инфиксная запись: " << expression.getInfix() << endl;  cout << "Обратная польская запись: "<< expression.getPostfix() << endl;  double res = expression.calculate();  cout << "Результат: "<< res << endl; |

Рисунок 5 – пример использования класса ArithmeticExpression

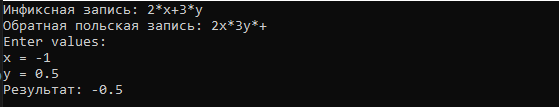


Рисунок 6 – вывод консоли в примере использования класса ArithmeticExpression

Так как на практике часто встречаются задачи, в которых необходимо несколько раз вычислять аналогичные выражения с разными константами, рекомендуется вводить их в виде выражения с переменными и вызывать метод calculate() для одного объекта несколько раз. Это позволит уменьшить расходы на трансляцию выражения и может сильно оптимизировать вашу программу.

Другие примеры работы с программой доступны в приложении 3.

# Заключение

Для представления и вычисления арифметических выражений на языке программирования c++ был разработан класс, представляющий арифметическое выражение. Он позволяет считывать арифметическое выражение из строки, а затем вычислять его значения. При этом выражение может содержать переменные, значение которых будут подставлены непосредственно при вычислении, что позволяет оптимизировать работу в большом числе случаев.

# 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 коды программ, иллюстрирующие возможности ArithmeticExpression, вместе с изображениями диалогового окна.

|  |
| --- |
| #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;  } |
|  |

|  |
| --- |
| #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;  } |
|  |