**МИНОБРНАУКИ РОССИИ**

**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**

**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

**«ЛЭТИ» ИМ. В. И. УЛЬЯНОВА (ЛЕНИНА)**

**Кафедра САПР**

**ОТЧЕТ**

**по курсовой работе**

**по дисциплине «Алгоритмы и структуры данных»**

**Тема: «Преобразование алгебраических формул из инфиксной в постфиксную форму записи и вычисление значения выражения»**

**Вариант 1**

Студент гр. 9302 \_\_\_\_\_\_\_\_\_\_\_\_ Тарабурин А.П.

Преподаватель \_\_\_\_\_\_\_\_\_\_\_\_ Тутуева А.В.

Санкт-Петербург

2020

Оглавление

[1. Постановка задачи и описание реализуемых класса и алгоритмов 3](#_Toc59977831)

[2. Описание реализованных unit-тестов 5](#_Toc59977832)

[3. Код программы 5](#_Toc59977833)

[4. Пример работы 13](#_Toc59977834)

[5. Вывод 14](#_Toc59977835)

# Постановка задачи и описание реализуемых класса и алгоритмов

Необходимо реализовать простейшую версию калькулятора. Пользователю должен быть доступен ввод математического выражения, состоящего из чисел и арифметических знаков. Программа должна выполнить проверку корректности введенного выражения. В случае некорректного ввода необходимо вывести сообщение об ошибке с указанием позиции некорректного ввода. В противном выводится обратная польская нотация введенного выражения, а также отображается результат вычисления.

Для реализации задачи потребуется класс строки strl, stack, элементами которого являются строки, а также класс postfix, в котором будет аккумулироваться выходные данные.

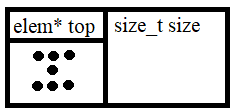
Class strl



Class elem



Class stack



Class postfix



Strl, stack и postfix позаимствованы с предыдущих лабораторных работ

Для этого используем следующие функции:

|  |  |  |
| --- | --- | --- |
| **Функция** | **Описание** | **Оценка временной сложности** |
| **Class strl** | | |
| Strl() | Конструктор по умолчанию | O(1) |
| Strl(cons char [N]) | Конструктор с использованием массива символов | O(l) |
| Strl(double) | Конструктор из числа | O(n) |
| ~strl() | Деструктор | O(1) |
| Char& operator[] | Возвращение символа по индексу | O(1) |
| Strl& operator+= (char added) | Присоединение символа в конец строки | O(1) |
| Strl& operator+= (strl added) | Присоединение строки в конец строки | O(n) |
| Bool operator==(strl&, strl&) | Проверка равенства двух строк | O(n) |
| Bool operator!=(strl&, strl&) | Проверка неравенства двух строк | O(n) |
| Size\_t getSize() | Возвращение размера строки | O(1) |
| Void input(fstream&) | Ввод строки из файла | O(n) |
| Void output(fstream&) | Вывод строки в файл | O(n) |
| Bool isDigit() | Проверка является ли строка числом | O(n) |
| Bool isOperation() | Проверка является ли строка операцией | O(n) |
| Int getOperationWeight() | Возвращение приоритета операции | O(1) |
| Double perform(strl) | Выполнение действия с одним аргументом | O(1) |
| Double perform(strl, strl) | Выполнение действия с двумя аргументами | O(1) |
| Bool isCorrectInfix(fstream&) | Проверка корректности инфиксной формы | O(n) |
| **Class elem** | | |
| elem(strl, elem\*) | Конструктор с использованием строки | O(1) |
| elem(double, elem\*) | Конструктор с использованием числа | O(1) |
| ~elem() | Деструктор | O(1) |
| bool isBracket() | Проверка является ли элемент скобкой | O(1) |
| int getWeight() | Возврат приоритета операции | O(1) |
| elem\* getNext() | Возврат следующего элемента стека | O(1) |
| strl getSymbol() | Возврат строки элемента | O(1) |
| **Class stack** | | |
| stack() | Конструктор по умолчанию | O(1) |
| ~stack() | Деструктор по умолчанию | O(1) |
| elem getTop() | Возвращение верхнего элемента стека | O(1) |
| void pop() | Удаление верхнего элемента стека | O(1) |
| void push(strl) | Добавление строки наверх стека | O(1) |
| void push(double) | Добавление числа наверх стека | O(n) |
| size\_t getSize() | Возвращение количества элементов в стеке | O(1) |
| bool isEmpty() | Проверка стека на пустоту | O(1) |
| void clear() | Очистка стека от элементов | O(n) |
| **Class postfix** | | |
| postfix(strl) | Конструктор постфиксной форму по инфиксной | O(n) |
| strl calculate() | Расчет результата в постфиксной форме | O(n^2) |
| strl getStream() | Возврат постфиксной формы | O(1) |
| void addOperationToStream() | Удаление операции из стека и добавление в выходной поток | O(1) |

Алгоритм:

Просматриваем выражение слева направо. Если просматриваемый символ является левой скобкой, помещаем его в стек. Если просматриваемый символ – операнд, добавляем его в выходной поток (постфиксную ФЗ). Если просматриваемый символ является правой скобкой, извлекаем из стека все символы до появления левой скобки и добавляем их в выходной поток. Если просматриваемый символ является оператором, удаляем все операторы из стека, пока не увидим оператор с меньшим приоритетом, и помещаем их в выходной поток. Затем помещаем просматриваемый оператор в стек. Иначе помещаем просматриваемый оператор в стек сразу. Операции с одним аргументом имеют самый высокий приоритет, скобки – самый низкий. Для вычисления Просматриваем выражение в постфиксной ФЗ. Если просматриваемый символ является операндом, то помещаем его в стек. Если просматриваемый символ является оператором, то применяем его к операндам, которые извлекаем из стека

# Описание реализованных unit-тестов

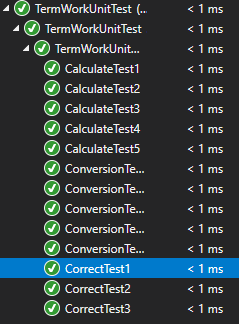
|  |  |
| --- | --- |
| Имя теста | Описание |
| СorrectTest1 | Закрывающая скобка в начале выражение - ошибка |
| СorrectTest2 | Нет пробелов между операндами и операциями |
| СorrectTest3 | Корректная строка |
| ConversionTest1 | Простое выражение |
| ConversionTest2 | Тригонометрия |
| ConversionTest3 | Деление |
| ConversionTest4 | Степень |
| ConversionTest5 | Много скобок |
| CalculateTest1 | Много скобок |
| CalculateTest2 | Тригонометрия |
| CalculateTest3 | Тригонометрия |
| CalculateTest4 | Натуральный логарифм |
| CalculateTest5 | Большие числа |

# Код программы

|  |
| --- |
| Strl.h |
| #pragma once  #include <fstream>  #include <cmath>  #ifndef STRL\_H  #define STRL\_H  const int N = 128;  using namespace std;  class strl {  private:  char p[N];  size\_t size;  public:  strl() {  size = 0;  }  strl(const char arr[N]) {  size = 0;  while (size < N && arr[size] != '\0') p[size] = arr[size++];  }  strl(double value) { //parsing double to string using sprintf  char temp[N];  sprintf\_s(temp, "%f", value);  int i = 0;  while (i < N && ((temp[i] >= '0' && temp[i]<='9') || temp[i] == '.' || temp[i] == '-'))  p[i] = temp[i++];  size = i;  }  ~strl() {  //delete[] p;  }  char& operator[] (int i) {  return p[i];  }  strl& operator+= (char added) { //adding single char to string`s end  p[size++] = added;  return \*this;  }  strl& operator+= (strl added) {  for (size\_t i = 0; i < added.getSize(); i++)  p[size + i] = added.p[i];  size += added.getSize() + 1;  p[size - 1] = ' ';  return \*this;  }    friend bool operator== (const strl& first\_compared, const strl& second\_compared) {  if (first\_compared.size != second\_compared.size) return false;  for (int i = 0; i < first\_compared.size; i++)  if (first\_compared.p[i] != second\_compared.p[i]) return false;  return true;  }  friend bool operator!= (const strl& first\_compared, const strl& second\_compared) {  if (first\_compared.size != second\_compared.size) return true;  for (int i = 0; i < first\_compared.size; i++)  if (first\_compared.p[i] != second\_compared.p[i]) return true;  return false;  /\*  if (first\_compared == second\_compared) return false;  else return true;  \*/  }  /\*  friend bool operator== (const strl& first\_compared, const char second\_compared[128]) {    }  \*/  size\_t getSize() {  return size;  }  void input(fstream& f){  size = 0;  char s = ' ';  f >> noskipws;  while (!f.eof()) {  f >> s;  if (s != '\n' && !f.eof()) p[size++] = s;  else break;  }  }  void output(fstream& f) {  int i = 0;  while (i < size) f << p[i++];  }  bool isDigit() {  if ((p[0] >= '0' && p[0] <= '9') || \*this == "pi" || \*this == "e") return true;  else if (p[0] == '-' && p[1] >= '0' && p[1] <= '9') return true;  else return false;  }  bool isOperation() {  if (getOperationWeight() == -1) return false;  else return true;  }  int getOperationWeight() {  if (\*this == "(" || \*this == ")") return 0;  else if (\*this == "+" || \*this == "-") return 1;  else if (\*this == "\*" || \*this == "/" || \*this == "%") return 2;  else if (\*this == "^") return 3;  else if (\*this == "cos" || \*this == "sin" || \*this == "tg" || \*this == "ctg" || \*this == "ln" || \*this == "log" || \*this == "sqrt") return 4;  else return -1;  }  double perform(strl first\_operand) { //carrying out operations with single operand  double operand = atof(first\_operand.p);  if (\*this == "cos") return cos(operand);  if (\*this == "sin") return sin(operand);  if (\*this == "tg") return tan(operand);  if (\*this == "ctg") return 1 / tan(operand);  if (\*this == "ln") return log(operand);  if (\*this == "log") return log10(operand);  if (\*this == "sqrt") return sqrt(operand);  }  double perform(strl first\_operand, strl second\_operand) { //carrying out operations with 2 operands  double second = atof(first\_operand.p); // in case we are taking two operands they already swapped in stack  double first = atof(second\_operand.p); // we need to swap them again  if (\*this == "+") return first + second;  if (\*this == "-") return first - second;  if (\*this == "\*") return first \* second;  if (\*this == "/") return first / second;  if (\*this == "%") return fmod(first, second);  if (\*this == "^") return pow(first, second);  }  bool isCorrectInfix(fstream &toOut) {  int bracketsCount = 0;  strl cur;  for (int i = 0; i <= size; i++) {  if (p[i] != ' ' && i != size) cur += p[i];  else {  if (!cur.isOperation() && !cur.isDigit()) {  toOut << "\nUnknown character detected at " << i << " position is ";  cur.output(toOut);  return false;  }  else {  if (cur == "(") bracketsCount += 1;  else if (cur == ")") {  if (bracketsCount > 0) bracketsCount -= 1;  else {  toOut << "\nIncorrect placement of brackets";  return false;  }  }  }  cur = "";  }  }  if (!bracketsCount) return true;  else {  toOut << "\nIncorrect placement of brackets";  return false;  }  }  };  #endif // !STRL\_H |
| Elem.h |
| #pragma once  #include "strl.h"  #ifndef ELEM\_H  #define ELEM\_H  class elem {  private:  strl symbol;  int weight;  elem\* next;  public:  elem(strl elem\_symbol, elem\* elem\_next) {  symbol = elem\_symbol;  next = elem\_next;  weight = elem\_symbol.getOperationWeight();  }  elem(double value, elem\* elem\_next) {  symbol = value;  next = elem\_next;  weight = -1;  }  ~elem() {  }  bool isBracket() {  if (symbol == "(" || symbol == ")") return true;  else return false;  }  elem\* getNext() {  return next;  }  int getWeight() {  return weight;  }  strl getSymbol() {  return symbol;  }  };  #endif // !ELEM\_H |
| Stack.h |
| #pragma once  #include "elem.h"  #ifndef STACK\_H  #define STACK\_H  class stack {  private:  size\_t size;  elem\* top;  public:  stack() {  size = 0;  top = nullptr;  }  ~stack() {  clear();  }  elem getTop() {  return \*top;  }  void pop()  {  elem\* temp = top;  top = top->getNext();  delete temp;  size--;  }  void push(strl value) {  top = new elem(value, top);  size++;  }  void push(double value) {  top = new elem(value, top);  size++;  }  size\_t getSize() {  return size;  }  bool isEmpty() {  if (size == 0) return true;  else return false;  }  void clear() {  while (!isEmpty()) pop();  }  };  #endif // !STACK\_H |
| Postfix.h |
| #pragma once  #ifndef POSTFIX\_H  #include "stack.h"  #include "strl.h"  #define POSTFIX\_H  class postfix {  private:  stack operations;  strl outstream;  public:  postfix(strl infix) { //convert correct infix form to postfix  strl cur;  int curOperationWeight = 0;  bool flag = false;  for (int i = 0; i <= infix.getSize(); i++) {  if (infix[i] != ' ' && i != infix.getSize()) cur += infix[i];  else {  if (cur == "(")  operations.push(cur);  else if (cur.isDigit())  outstream += cur;  else if (cur == ")") {  while (!operations.isEmpty() && operations.getTop().getSymbol() != "(")  addOperationToStream();  operations.pop();  }  else {  if (cur != " ") {  curOperationWeight = cur.getOperationWeight();  while (!operations.isEmpty() && operations.getTop().getWeight() >= curOperationWeight)  addOperationToStream();  operations.push(cur);  }  }  cur = "";  }  }  while (!operations.isEmpty())  addOperationToStream();  }  strl calculate() { //calculating expression in postfix form  double res = 0;  strl cur;  for (int i = 0; i < outstream.getSize(); i++) {  if (outstream[i] != ' ') cur += outstream[i];  else {  if (cur.isDigit()) {  if (cur == "pi") cur = 3.1415926535;  else if (cur == "e") cur = 2.71828182846;  operations.push(cur);  }  else if (cur.getOperationWeight() == 4) {  res = cur.perform(operations.getTop().getSymbol());  operations.pop();  operations.push(res);  }  else {  res = cur.perform(operations.getTop().getSymbol(), operations.getTop().getNext()->getSymbol());  operations.pop();  operations.pop();  operations.push(res);  }  cur = "";  }  }  return operations.getTop().getSymbol();  }  strl getStream() {  return outstream;  }  void addOperationToStream() { // removing current operation from stack and place to output  outstream += operations.getTop().getSymbol();  operations.pop();  }    };  #endif // !POSTFIX\_H |
| TermWorkUnitTest.cpp |
| #include "pch.h"  #include "CppUnitTest.h"  #include "../termWork/postfix.h"  #include "../termWork/strl.h"  using namespace Microsoft::VisualStudio::CppUnitTestFramework;  namespace TermWorkUnitTest  {  TEST\_CLASS(TermWorkUnitTest)  {  public:  strl infix;  TEST\_METHOD(СorrectTest1)  {  infix = ") 2 + 2";  fstream f;  Assert::IsFalse(infix.isCorrectInfix(f));  }  TEST\_METHOD(СorrectTest2)  {  infix = "2+2";  fstream f;  Assert::IsFalse(infix.isCorrectInfix(f));  }  TEST\_METHOD(СorrectTest3)  {  infix = "( ( 2 + 2 ) \* 2 ) / 8";  fstream f;  Assert::IsTrue(infix.isCorrectInfix(f));  }  TEST\_METHOD(ConversionTest1)  {  infix = "( ( 2 + 2 ) \* 2 ) / 8";  postfix pf(infix);  Assert::IsTrue(pf.getStream() == "2 2 + 2 \* 8 / ");  }  TEST\_METHOD(ConversionTest2)  {  infix = "cos ( pi \* 2 - ( 2 \* pi ) )";  postfix pf(infix);  Assert::IsTrue(pf.getStream() == "pi 2 \* 2 pi \* - cos ");  }  TEST\_METHOD(ConversionTest3)  {  infix = "10 + 3 \* 5 / ( 16 - 4 )";  postfix pf(infix);  Assert::IsTrue(pf.getStream() == "10 3 5 \* 16 4 - / + ");  }  TEST\_METHOD(ConversionTest4)  {  infix = "5 \* 3 ^ ( 4 - 2 )";  postfix pf(infix);  Assert::IsTrue(pf.getStream() == "5 3 4 2 - ^ \* ");  }  TEST\_METHOD(ConversionTest5)  {  infix = "( 2 + ( 3 + ( 4 + ( 5 + 6 \* ( 7 + 8 ) ) ) ) )";  postfix pf(infix);  Assert::IsTrue(pf.getStream() == "2 3 4 5 6 7 8 + \* + + + + ");  }  TEST\_METHOD(CalculateTest1)  {  infix = "( 2 + ( 3 + ( 4 + ( 5 + 6 \* ( 7 + 8 ) ) ) ) )";  postfix pf(infix);  Assert::IsTrue(pf.calculate() == 104);  }  TEST\_METHOD(CalculateTest2)  {  infix = "cos ( pi \* 2 - ( 2 \* pi ) )";  postfix pf(infix);  Assert::IsTrue(pf.calculate() == 1);  }  TEST\_METHOD(CalculateTest3)  {  infix = "( 2 + 3 \* ( 4 ^ cos ( pi / 4 ) ) )";  postfix pf(infix);  Assert::IsTrue(pf.calculate() == 9.995435);  }  TEST\_METHOD(CalculateTest4)  {  infix = "ln e";  postfix pf(infix);  Assert::IsTrue(pf.calculate() == 1);  }  TEST\_METHOD(CalculateTest5)  {  infix = "2382323.471247 \* 35612731287312.415467126412";  postfix pf(infix);  Assert::IsTrue(pf.calculate() == 84841045620976762880.000000);  }  };  } |
| Main.cpp |
| #include <fstream>  #include <iostream>  #include "strl.h"  #include "postfix.h"  using namespace std;  int main(){  fstream input, output;  input.open("input.txt",ios::in);  output.open("output.txt", ios::out);  strl infix;  infix.input(input);  infix.output(output);  if (infix.isCorrectInfix(output)) {  postfix pf(infix);  output << "\nPostfix form: ";  pf.getStream().output(output);  output << "= ";  pf.calculate().output(output);  }  output.close();  input.close();  return 0;  } |
| Elem.h |
| #pragma once  #include "node.h"  class Elem {  public:  Elem\* next;  Node\* value;  Elem(Node\* elem\_value, Elem\* elem\_next) {  value = elem\_value;  next = elem\_next;  }  ~Elem() {  delete value;  }  }; |

# Пример работы





# Вывод

При выполнении данной курсовой работы я познакомился с методами реализации вычисления выражений с различными операциями, а также представлением выражений в постфиксной форме.