**МИНОБРНАУКИ РФ**

**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**

**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

**ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА) “ЛЭТИ”**

**ФАКУЛЬТЕТ КОМПЬЮТЕРНЫХ ТЕХНОЛОГИЙ И ИНФОРМАТИКИ**

**КАФЕДРА САПР**

**Отчёт по курсовой работе   
по дисциплине «Алгоритмы и структуры данных»**

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

**Вариант 2.**

**Преподаватель: \_\_\_\_\_\_\_\_\_\_\_\_\_\_ Тутуева А.В.**

**Студент гр.9302: \_\_\_\_\_\_\_\_\_\_\_\_\_\_ Новокрещенов Д.К.**

**Санкт-Петербург**

**2020**

**Содержание**

[1. Постановка задачи 3](#_Toc59307273)

[2. Обоснование выбора используемых структур данных 3](#_Toc59307274)

[3. Описание алгоритма решения 4](#_Toc59307275)

[4. Пример работы программы 6](#_Toc59307276)

[5. Листинг программы 6](#_Toc59307277)

[ЗАКЛЮЧЕНИЕ 21](#_Toc59307278)

# Постановка задачи

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 случае выводится польская нотация введенного выражения, а также отображается результат вычисления.

# Обоснование выбора используемых структур данных

Для решения поставленной задачи я реализовал классы BiList и prefix.

BiList – реализация двунаправленного списка, ранее использовавшегося в лабораторных работах. Данная реализация является достаточно надежной, что подтверждается многочисленными тестами.

struct Node

|  |  |
| --- | --- |
| **Компонент** | **Описание** |
| Node \*prev; | Указатель на предыдущий элемент списка |
| string value; | Тип данных, который хранится в списке. |
| Node \*next; | Указатель на следующий элемент списка |

**class BiList**

|  |  |  |
| --- | --- | --- |
| **Компонент** | **Описание** | |
| Node \*head; | Указатель на начало списка | |
| Node \*last; | Указатель на конец списка | |
| size\_t size; | Переменная, хранящая размер списка | |
| **Метод** | **Описание** | **Оценка временной сложности** |
| void push\_back(int) | Вставка нового элемента в конец списка | O(1) |
| void push\_front(int) | Вставка нового элемента в начало списка | O(1) |
| void pop\_back() | Удаление элемента из конца списка | O(1) |
| void pop\_front() | Удаление элемента из конца списка | O(1) |
| void insert(string, size\_t) | Вставка нового элемента по индексу | O(n) |
| string at(int) | Получение элемента по индексу | O(n) |
| void remove(size\_t) | Удаление элемента по индексу | O(n) |
| size\_t get\_size() | Получение размера списка | O(n) |
| void print\_to\_console() | Вывод элементов в консоль через разделитель | O(n) |
| void clear() | Удаление всех элементов списка | O(n) |
| void set(size\_t, int) | Замена элемента по индексу | O(n) |
| bool isEmpty() | Проверка на пустоту списка | O(1) |
| void reverse() | Обращение списка | O(n) |

prefix – новый класс, в котором я реализовал алгоритм преобразования алгебраического выражения из инфиксной нотации в префиксную, а также алгоритм расчёта выражения, записанного в префиксной нотации.

Каждый метод умеет обнаруживать ошибки и сообщать о них.

**class prefix**

|  |  |
| --- | --- |
| **Компонент** | **Описание** |
| BiList calculate; | Список компонентов выражения |
| BiList temp; | Временный список для всех компонентов выражения |
| **Метод** | **Описание** |
| friend string make\_func(string); | Дружественная функция для расчета функций одного аргумента |
| string get\_result(); | Расчёт выражения в префиксной нотации |
| string print\_prefix(); | Вывод префиксной нотации |
| void break\_into\_pieces(string); | Разбиение первоначального выражения на составляющие |
| void clear(); | Очистка всех компонентов |
| void clone(); | Клонирование информации из temp в calculate |
| void get\_normal\_string(); | Нормализация списка компонентов |
| void get\_prefix(); | Получение префиксной нотации из инфиксной. |
| void make\_prefix(string input); | Набор инструкций для получения префиксного выражения |

# Также мною был реализован ряд алгоритмов, не являющихся методами классов.

|  |  |
| --- | --- |
| **Название** | **Описание** |
| bool isNumber(string) | Проверка, является ли строка числом |
| bool isOperator(string, size\_t, size\_t) | Проверка, является ли строка оператором |
| char get\_priority(string) | Получение приоритета оператора |
| char isConst(string, size\_t) | Проверка, является ли строка константой |
| char isFunc(string, size\_t) | Проверка, является ли строка функцией |
| string make\_func(string) | Расчёт функций одной переменной |
| string make\_operator(string, string, string) | Расчёт функций двух переменных |

# Описание алгоритма решения

1. **Ввод выражения.**
2. **Разбиение на компоненты.**

* Если символ – оператор или скобка, то добавляем его в список temp
* Если набор символов является константой, то добавляем его в список temp
* Если набор символов является числом, то добавляем его в список temp
* Если набор символов является функцией, то необходимо найти аргумент функции. Необходимо учитывать, что внутри скобок, где содержится аргумент, могут быть другие скобки или функции одного аргумента. После функцию с аргументом добавляем цельным элементом в список temp

1. **Нормализация выражения**

* Если знак минуса стоит в начале выражения перед открывающей скобкой, то добавляем перед минусом «0»
* Если знак минуса стоит в начале или после открывающей скобки перед числом или константой, то добавляем этот минус к числу или константе (реализуем унарный минус)
* Если элемент является функцией, то аргумент преобразуем по алгоритму преобразования в префиск.

1. **Преобразование выражения из инфиксной нотации в префиксную.**

* Делаем обращение списка temp (с учётом поворота скобок), а также создаём вспомогательный список stack.
* Реализуем обработку по алгоритму преобразования в постфиксную нотацию:

Если входящий элемент число, константа или функция, то добавляем его в calculate.

Если входящий элемент оператор (+, -, \*, /, ^) то проверяем:

* Если stack пуст или содержит левую скобку в вершине, то добавляем входящий оператор в stack.
* Если входящий оператор имеет более высокий или равный приоритет, чем вершина stack, помещаем его в stack.
* Если входящий оператор имеет более низкий приоритет, чем вершина stack, выгружаем первый элемент stack в calculate, пока не увидим оператор с меньшим приоритетом или левую скобку на вершине stack, затем добавляем входящий оператор в stack.

Если входящий элемент является левой скобкой, помещаем его в stack.

Если входящий элемент является правой скобкой, выгружаем stack и добавляем его элементы в calculate, пока не увидим левую круглую скобку. Удаляем найденную скобку из stack.

В конце выражения выгружаем stack в calculate.

* Делаем обращение полученного списка calculate.

1. **Расчёт выражения в префиксной нотации.**

В данном пункте рассматривается несколько вариантов:

1. Если получившийся список содержит один элемент

Рассматриваем элемент.

* Если элемент – число или константа, то выводим число как результат
* Если элемент – функция, то рассчитываем эту функцию и выводим результат расчёта
* При любом другом варианте выдаём ошибку

1. Если в получившемся списке два элемента

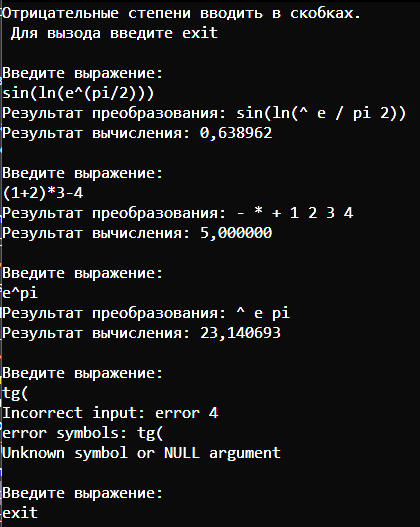
Рассматриваем пару элементов:

* Если элементы не пара «Оператор (+ или - ) и функция» выдаём ошибку
* В противном случае вычисляем функцию и меняем знак в соответствии с оператором.

1. Если в получившемся списке три и более элементов

* Если паросочетание трех элементов образует связку – «оператор, число, число», то рассчитываем по условию выражение и заносим его в список вместо трех элементов.
* Если встречается функция – заменяем её результатом выполнения этой функции.
* Делаем так до тех пор, пока длина calculate больше 1.
* При зацикливании выводим ошибку.

# Пример работы программы



Также методы и функции были покрыты Unit-тестами, для контроля за корректностью алгоритмов.

# Листинг программы

|  |
| --- |
| main.cpp |
| #include <iostream>  #include <string>  #include "prefix.h"  using namespace std;  int main()  {  prefix Test;  string input;  setlocale(LC\_ALL, "Rus");  cout << "Отрицательные степени вводить в скобках.\n Для вызода введите exit\n ";  while (1)  {  cout << "\nВведите выражение:\n";  getline(cin, input);  if (input == "exit")  break;  try {  Test.make\_prefix(input);  cout << "Результат преобразования: " << Test.print\_prefix() << endl;  cout << "Результат вычисления: " << Test.get\_result() << endl;  }  catch(const char\* error)  {  cout << error << endl;  }  }  } |
| List.h |
| #pragma once  #include <cstddef>  #include <iostream>  #include <string>  using namespace std;  struct Node  {  string data;  Node\* prev = NULL;  Node\* next = NULL;  };  class BiList  {  public:  BiList();  ~BiList();  void push\_back(string);  void push\_front(string);  void pop\_back();  void pop\_front();  void remove(size\_t);  void insert(string,size\_t);  string at(size\_t);  string top();  size\_t get\_size();  void set(size\_t index, string value);  void print\_to\_console();  bool isEmpty();  void reverse();  void clear();  private:  Node\* head;  Node\* last;  size\_t size;  }; |
| List.cpp |
| #pragma once  #include "List.h"  BiList::BiList()  {  head = NULL;  last = NULL;  size = 0;  }  BiList::~BiList()  {  while (head)  {  last = head->next;  delete head;  head = last;  }  }  //The function inserts a new node at back of the list.  void BiList::push\_back(string value)  {  Node\* temp = new Node;  temp->next = NULL;  temp->data = value;  if (!this->isEmpty())  {  temp->prev = last;  last->next = temp;  last = temp;  }  else {  temp->prev = NULL;  head = last = temp;  }  size++;  }  //The function inserts a new node at front of the list.  void BiList::push\_front(string value)  {  Node\* temp = new Node;  temp->prev = NULL;  temp->data = value;  if (!this->isEmpty())  {  temp->next = head;  head->prev = temp;  head = temp;  }  else  {  temp->prev = NULL;  temp->next = NULL;  head = last = temp;  }  size++;  }  //The function deletes the last node in the list.  void BiList::pop\_back()  {  if(size==0)  return;  if (size == 1)  {  delete last;  last = head = NULL;  size--;  return;  }  delete last->next;  last = last->prev;  last->next = NULL;  size--;  }  //The function deletes the first node in the list.  void BiList::pop\_front()  {  if (size == 0)  return;  if (size == 1)  {  delete head;  head = last = NULL;  size--;  return;  }  delete head->prev;  head->prev = NULL;  head = head->next;  size--;  }  //The function gets the value from the node with the number "index".  string BiList::top()  {  if (head)  return head->data;  else throw "is Empty";  }  void BiList::insert(string value, size\_t index)  {  if ((size == 0 && index == 0) || index >= size)  throw "Incorrect index.";  if (index == 0)  {  this->push\_front(value);  return;  }  Node\* cursor = head;  for (size\_t i = 0; i < index; i++)  cursor = cursor->next;  Node\* temp = new Node;  temp->data = value;  temp->next = cursor;  temp->prev = cursor->prev;  cursor->prev = temp;  temp->prev->next = temp;  size++;  }  //The function gets the number of items in the list  size\_t BiList::get\_size()  {  return size;  }  void BiList::set(size\_t index, string value)  {  if (index >= size || index < 0)  throw"Incorrect index.";  Node\* cursor = head;  for (size\_t i = 0; i < index; i++)  cursor = cursor->next;  cursor->data = value;  }  //The function gets the value from the node with the number "index".  string BiList::at(size\_t index)  {  if (index >= size || index < 0)  throw "Incorrect index.";  Node\* cursor = head;  for (size\_t i = 0; i < index; i++)  cursor = cursor->next;  return cursor->data;  }  //The function deletes all node in list.  void BiList::clear()  {  while (head != NULL)  this->pop\_front();  head = last = NULL;  }  //The function prints all list in console.  void BiList::print\_to\_console()  {  Node\* cursor = head;  if (head)  {  while (cursor->next)  {  std::cout << cursor->data << " ";  cursor = cursor->next;  }  std::cout << cursor->data;  std::cout << std::endl;  }  else throw "List is empty.";  }  //The function checks the list for emptiness.  bool BiList::isEmpty()  {  if (head==NULL) return true;  return false;  }  //The function reverses a list.  void BiList::reverse()  {  if (!head || !head->next)  return;  last = head;  Node\* temp = NULL;  Node\* current = head;  while (current != NULL)  {  temp = current->prev;  current->prev = current->next;  current->next = temp;  current = current->prev;  }  head = temp->prev;  }  //The function deletes the node with the "index" number.  void BiList::remove(size\_t index)  {  if (index >= size || index < 0)  throw "Incorrect index.";  if (index == 0)  {  this->pop\_front();  return;  }  if (index == size - 1)  {  this->pop\_back();  return;  }  Node\* cursor = head;  for (size\_t i = 0; i < index; i++)  cursor = cursor->next;  Node\* temp = cursor;  cursor->prev->next = cursor->next;  cursor->next->prev = cursor->prev;  delete temp;  size--;  } |
| prefix.h |
| #pragma once  #include "List.h"  class prefix {  private:  BiList calculate;  BiList temp;  void get\_normal\_string();  void break\_into\_pieces(string);  void get\_prefix();  friend string make\_func(string input);  void clone();  void clear();  public:  void make\_prefix(string input);  string print\_prefix();  string get\_result();  }; |
| prefix.cpp |
| #include "prefix.h"  #include "prefix\_math.h"  bool isOperator(string oper, size\_t index = 0, size\_t ctrl = 0) // + - \* / ^  {  if(ctrl==0 && oper[1] >= '0' && oper[1] <= '9')  return false;  if (oper[index] == '+' || oper[index] == '-' || oper[index] == '\*' || oper[index] == '/' || oper[index] == '^')  return true;  return false;  }  char isFunc(string oper, size\_t index=0) //cos, sin, tg, ctg, ln, log, sqrt, abs  {  if (oper[index] == 't' && oper[index + 1] == 'g') //tg  return 2;  else if (oper[index] == 'l' && oper[index + 1] == 'n') //ln  return 2;  else if (oper[index] == 'c' && oper[index + 1] == 'o' && oper[index + 2] == 's') //cos  return 3;  else if (oper[index] == 'c' && oper[index + 1] == 't' && oper[index + 2] == 'g') //ctg  return 3;  else if (oper[index] == 'l' && oper[index + 1] == 'o' && oper[index + 2] == 'g') //log  return 3;  else if (oper[index] == 's' && oper[index + 1] == 'i' && oper[index + 2] == 'n') //sin  return 3;  else if (oper[index] == 'a' && oper[index + 1] == 'b' && oper[index + 2] == 's') //abs  return 3;  else if (oper[index] == 's' && oper[index + 1] == 'q' && oper[index + 2] == 'r' && oper[index+3]=='t') //sqrt  return 4;  return 0;  }  char isConst(string oper, size\_t index=0) //pi, e  {  if (oper[index] == 'p' && oper[index + 1] == 'i') //pi  return 2;  else if (oper[index] == 'e') //e  return 1;  return 0;  }  bool isNumber(string str)  {  if (str[0] >= '0' && str[0] <= '9')  return true;  if (str[0] == '-' && str[1] >= '0' && str[1] <= '9')  return true;  if (isConst(str))  return true;  return false;  }  char get\_priority(string oper)  {  if (isNumber(oper) || isFunc(oper, 0) || isConst(oper, 0))  return 0;  if (oper[0] == '+' || oper[0] == '-')  return 0;  if (oper[0] == '\*' || oper[0] == '/')  return 1;  return 2;  }  void prefix::get\_normal\_string() //приведение выражения в удобный для расчёта вид  {  for (size\_t i = 0; i < temp.get\_size(); i++)  {  string new\_str;  if (isFunc(temp.at(i)))  {  prefix Temp;  string arg = temp.at(i);  for (size\_t i = isFunc(arg) + 1; i < arg.size() - 1; i++)  new\_str.push\_back(arg[i]);  Temp.make\_prefix(new\_str);  new\_str.clear();  for (size\_t i = 0; i <isFunc(arg)+1; i++)  new\_str.push\_back(arg[i]);  new\_str+= Temp.print\_prefix() + ')';  temp.set(i, new\_str);  }  else if (i == 0 && temp.at(i) == "-" && temp.at(i + 1) != ")" && !isFunc(temp.at(i+1)) && !isOperator(temp.at(i+1))) //выделение унарного минуса для первого числа  {  new\_str += temp.at(i) + temp.at(i + 1);  temp.remove(i + 1);  temp.set(i, new\_str);  }  else if (i == 0 && temp.at(i) == "-" && (temp.at(i + 1) == ")" || isFunc(temp.at(i+1)))) //особая обработка унарного минуса, если вначале скобка  {  temp.push\_front("0");  i++;  }  else if (temp.at(i) == ")" && i+1<temp.get\_size() && temp.at(i + 1) == "-" && !isFunc(temp.at(i + 2))) //выделение унарного минуса для чисел внутри скобок  {  new\_str += temp.at(i+1) + temp.at(i + 2);  temp.remove(i + 1);  temp.set(i+1, new\_str);  }  else if (temp.at(i) == ")" && i + 1 < temp.get\_size() && temp.at(i + 1) == "-" && isFunc(temp.at(i + 2)))  {  temp.insert("0", i + 1);  }  }  }  void prefix::break\_into\_pieces(string input) //разбиение на составляющие  {  string temp;  for (size\_t i = 0; i < input.size(); i++)  {  temp.clear();  size\_t control = 0;  while (((input[i] >= 48 && input[i] <= 57)||(input[i]==',')) && i < input.size()) //выделение чисел  temp.push\_back(input[i++]);  if (temp.size() != 0)  {  this->temp.push\_back(temp);  temp.clear();  i--;  continue;  }  if (isOperator(input, i,1)) //выделение отдельных операторов или скобок  temp.push\_back(input[i]);  if (input[i] == '(')  temp.push\_back(')');  else if (input[i] == ')')  temp.push\_back('(');  if (temp.size() != 0)  {  this->temp.push\_back(temp);  temp.clear();  continue;  }  if (input[i] == ' ')  continue;  try  {  size\_t temp\_2 = isFunc(input, i); //выделение функций одного аргумента  if (temp\_2)  {  for (size\_t j = i; j < i + temp\_2; j++) //вставка самой функции  temp.push\_back(input[j]);  temp\_2--;  size\_t control = 0;  do  {  temp\_2++;  if (input[i + temp\_2] == '(')  control++;  if (input[i + temp\_2] == ')')  control--;  temp.push\_back(input[i + temp\_2]);  } while (control != 0 && input.size() > i + temp\_2); //вставка аргумента функции  if (control != 0)  throw "Incorrect input: error 4";  this->temp.push\_back(temp);  temp.clear();  i += temp\_2;  continue;  }  }  catch (const char\* error)  {  cout << error << endl;  cout << "error symbols: ";  for (size\_t j = i; j < input.size(); j++)  cout << input[j];  cout << endl;  return;  }  try  {  size\_t temp\_2 = isConst(input, i); //выделение констант - пи и экспоненты  if (temp\_2)  {  for (size\_t j = i; j < i + temp\_2; j++)  temp.push\_back(input[j]);  this->temp.push\_back(temp);  temp.clear();  i += temp\_2-1;  continue;  }  }  catch (const char\* error)  {  cout << error << endl;  cout << "error symbols: ";  for (size\_t j = i; j < input.size(); j++)  cout << input[j];  cout << endl;  return;  }  throw "Unknown symbols";  }  get\_normal\_string();  this->temp.reverse();  }  void prefix::get\_prefix() //получение префиксного выражения из инфиксного  {  BiList stack; //вспомогательный стек  for (size\_t i = 0; i < temp.get\_size(); i++)  {  if (isNumber(temp.at(i)) || isFunc(temp.at(i))) //в данной обработке числа и функции одного аргумента равны  calculate.push\_back(temp.at(i));  else if (isOperator(temp.at(i))) //обработка операторов  {  if (stack.isEmpty())  stack.push\_front(temp.at(i));  else if (stack.top() == "(")  stack.push\_front(temp.at(i));  else if (get\_priority(temp.at(i)) >= get\_priority(stack.top()))  stack.push\_front(temp.at(i));  else  {  while (!stack.isEmpty()&&stack.top() != "(" && get\_priority(temp.at(i)) < get\_priority(stack.top()))  {  calculate.push\_back(stack.top());  stack.pop\_front();  }  stack.push\_front(temp.at(i));  }  }  else if (temp.at(i) == "(")  stack.push\_front(temp.at(i));  else if (temp.at(i) == ")")  {  while (!stack.isEmpty()&&stack.top() != "(")  {  calculate.push\_back(stack.top());  stack.pop\_front();  }  stack.pop\_front();  }  }  while (!stack.isEmpty()) //выгрузка стека в результат  {  calculate.push\_back(stack.top());  stack.pop\_front();  }  calculate.reverse();  }  void prefix::clone()  {  for (size\_t i = 0; i < temp.get\_size(); i++)  calculate.push\_back(temp.at(i));  temp.clear();  }  void prefix::clear()  {  temp.clear();  calculate.clear();  }  void prefix::make\_prefix(string input)  {  try {  break\_into\_pieces(input);  get\_prefix();  temp.clear();  }  catch (const char\* error)  {  temp.clear();  }  }  string prefix::print\_prefix()  {  if (calculate.get\_size() == 0)  {  calculate.clear();  throw "Unknown symbol or NULL argument";  }  string result;  for (size\_t i = 0; i < calculate.get\_size(); i++)  result += calculate.at(i) + " ";  if(result.size()!=0)  result.pop\_back();  return result;  }  string prefix::get\_result()  {  string result;  size\_t control = 0;  if (calculate.get\_size() == 0)  result = "0";  else if (calculate.get\_size() == 1)  {  try {  if (isFunc(calculate.at(0)))  calculate.set(0, make\_func(calculate.at(0)));  else if (isConst(calculate.at(0)))  if (calculate.at(0) == "pi")  result = to\_string(pi);  else result = to\_string(e);  else if (calculate.at(0) == "-pi")  result = to\_string(-1 \* pi);  else if (calculate.at(0) == "-e")  result = to\_string(-1 \* e);  else if (isNumber(calculate.at(0)))  result = calculate.at(0);  else  {  calculate.print\_to\_console();  calculate.clear();  throw "Incorrect input: error 1: Incorrect Function";  }  }  catch (const char\* error)  {  calculate.print\_to\_console();  calculate.clear();  cout << error;  throw "Incorrect input: error 5: Unknown Error";  }  }  else if (calculate.get\_size() == 2)  {  if ((calculate.at(0) != "-" && calculate.at(0) != "+")||(isNumber(calculate.at(1))))  {  calculate.print\_to\_console();  calculate.clear();  throw "Incorrect input: error 2: extra operator";  }  if (isFunc(calculate.at(1))&& calculate.at(0) == "-")  {  calculate.set(0, to\_string(-1 \* stod(make\_func(calculate.at(1)))));  }  else if (isFunc(calculate.at(1)) && calculate.at(0) == "+")  {  calculate.set(0, make\_func(calculate.at(1)));  }  calculate.pop\_back();  } else  while (calculate.get\_size()>1)  {  control = 0;  for (size\_t i = 0; i<calculate.get\_size(); i++)  {  if (calculate.get\_size()-i >= 3 && isOperator(calculate.at(i)) && isNumber(calculate.at(i + 1)) && isNumber(calculate.at(i + 2)))  {  calculate.set(i, make\_operator(calculate.at(i), calculate.at(i + 1), calculate.at(i + 2)));  calculate.remove(i + 1);  calculate.remove(i + 1);  control++;  }  else if (isFunc(calculate.at(i)))  {  calculate.set(i, make\_func(calculate.at(i)));  control++;  } else if (calculate.get\_size() == 2 && isFunc(calculate.at(1)) && calculate.at(0) == "-")  {  control++;  calculate.set(0, to\_string(-1 \* stod(make\_func(calculate.at(1)))));  calculate.pop\_back();  }  }  if (control == 0)  {  calculate.clear();  throw "Incorrect input: error 3: extra operator or number";  }  }  result = calculate.at(0);  calculate.clear();  return result;  } |
| prefix\_math.h |
| #pragma once  #include <string>  #include <math.h>  #define pi 3.1415926535897932384626433832795  #define e 2.7182818284  using namespace std;  string make\_func(string);  string make\_operator(string, string, string); |
| prefix\_math.cpp |
| #include "prefix\_math.h"  #include "prefix.h"  #include <iostream>  string make\_func(string input) //расчёт функций одного аргумента  {  string func, num;  char control = 0;  for (size\_t i = 0; i < input.size()-1; i++)  {  if (input[i] == '(' && control == 0)  {  control++;  continue;  }  if (control)  num += input[i];  else func += input[i];  }  double result = 0, number;  if (num == "pi")  number = pi;  else if (num == "e")  number = e;  else  {  prefix temp; //обработка неконстантного или составного аргумента  temp.break\_into\_pieces(num);  temp.get\_normal\_string();  temp.clone();  temp.calculate.reverse();  number = stod(temp.get\_result());  }  if (func == "cos") result = cos(number);  else if (func == "sin") result = sin(number);  else if (func == "tg") result = tan(number);  else if (func == "ctg") result = 1 / tan(number);  else if (func == "ln") result = log(number);  else if (func == "log") result = log10(number);  else if (func == "sqrt") result = sqrt(number);  else if (func == "abs") result = abs(number);  else throw "Unknown error";  return to\_string(result);  }  string make\_operator(string input, string number\_1, string number\_2) //вычисления для операторов  {  double result = 0, num\_1, num\_2;  if (number\_1 == "pi")  num\_1 = pi;  else if (number\_1 == "e")  num\_1 = e;  else num\_1 = stod(number\_1);  if (number\_2 == "pi")  num\_2 = pi;  else if (number\_2 == "e")  num\_2 = e;  else num\_2 = stod(number\_2);  if (input == "+") result = num\_1 + num\_2;  if (input == "-") result = num\_1 - num\_2;  if (input == "\*") result = num\_1 \* num\_2;  if (input == "/") result = num\_1 / num\_2;  if (input == "^") result = pow(num\_1, num\_2);  return to\_string(result);  }; |
| MixTest.cpp |
| #include "pch.h"  #include "CppUnitTest.h"  #include "../Курсовая/List.cpp"  #include "../Курсовая/prefix.cpp"  #include "../Курсовая/prefix\_math.cpp"  using namespace Microsoft::VisualStudio::CppUnitTestFramework;  namespace MixTest  {  TEST\_CLASS(ListTest)  {  public:  BiList Empty;  BiList NotEmpty;  TEST\_METHOD\_INITIALIZE(init)  {  NotEmpty.push\_back("Some data");  }  TEST\_METHOD(IsEmpty\_for\_NotEmpty)  {  Assert::AreEqual(NotEmpty.isEmpty(), false);  }  TEST\_METHOD(IsEmpty\_for\_Empty)  {  Assert::AreEqual(Empty.isEmpty(), true);  }  TEST\_METHOD(Get\_Size\_for\_Empty)  {  size\_t real\_size = 0;  Assert::AreEqual(Empty.get\_size(), real\_size);  }  TEST\_METHOD(Get\_Size\_for\_NotEmpty)  {  size\_t real\_size = 1;  Assert::AreEqual(NotEmpty.get\_size(), real\_size);  }  TEST\_METHOD(at\_correct\_index)  {  string str = "Some data";  Assert::AreEqual(NotEmpty.at(0), str);  }  TEST\_METHOD(at\_incorrect\_index)  {  try  {  Empty.at(4);  }  catch (const char\* error)  {  Assert::AreEqual(error, "Incorrect index.");  }  }  TEST\_METHOD(Push\_Back\_NotEmpty)  {  string str = "5";  NotEmpty.push\_back(str);  Assert::AreEqual(NotEmpty.at(1), str );  }  TEST\_METHOD(Push\_Front\_NotEmpty)  {  string str = "1";  NotEmpty.push\_front(str);  Assert::AreEqual(NotEmpty.at(0), str);  }  TEST\_METHOD(remove\_correct\_index)  {  NotEmpty.push\_back("0");  string temp = NotEmpty.at(NotEmpty.get\_size()-2);  NotEmpty.remove(NotEmpty.get\_size()-1);  Assert::AreEqual(NotEmpty.at(NotEmpty.get\_size()-1), temp);  }  TEST\_METHOD(Pop\_Back\_for\_NotEmpty)  {  size\_t real\_size = NotEmpty.get\_size() - 1;  NotEmpty.pop\_back();  Assert::AreEqual(NotEmpty.get\_size(), real\_size);  }  TEST\_METHOD(Pop\_Back\_for\_Empty)  {  Empty.pop\_back();  Assert::AreEqual(Empty.isEmpty(), true);  }  TEST\_METHOD(Pop\_Front\_for\_NotEmpty)  {  size\_t real\_size = NotEmpty.get\_size() - 1;  NotEmpty.pop\_front();  Assert::AreEqual(NotEmpty.get\_size(), real\_size);  }  TEST\_METHOD(insert\_correct\_index)  {  string str = "4";  NotEmpty.insert("4", 0);  Assert::AreEqual(NotEmpty.at(0),str);  }  TEST\_METHOD(insert\_incorrect\_index)  {  try  {  Empty.insert("2", 0);  }  catch (const char\* error)  {  Assert::AreEqual("Incorrect index.", error);  }  }  TEST\_METHOD(remove\_incorrect\_index)  {  try  {  Empty.remove(2);  }  catch (const char\* error)  {  Assert::AreEqual("Incorrect index.", error);  }  }  TEST\_METHOD(reverse\_for\_NotEmpty)  {  string last\_inf = NotEmpty.at(NotEmpty.get\_size() - 1);  NotEmpty.reverse();  Assert::AreEqual(NotEmpty.at(0), last\_inf);  }  TEST\_METHOD(Pop\_Front\_for\_Empty)  {  Empty.pop\_front();  Assert::AreEqual(Empty.isEmpty(), true);  }  TEST\_METHOD(set\_correct\_index)  {  NotEmpty.set(0, "3");  string str = "3";  Assert::AreEqual(NotEmpty.at(0), str);  }  TEST\_METHOD(set\_incorrect\_index)  {  try  {  NotEmpty.set(10000000, "2");  }  catch (const char\* error)  {  Assert::AreEqual(error, "Incorrect index.");  }  }  TEST\_METHOD(Test\_Clear\_NotEmpty)  {  NotEmpty.clear();  size\_t real\_size = 0;  Assert::AreEqual(NotEmpty.get\_size(), real\_size);  }  TEST\_METHOD(Test\_Clear\_Empty)  {  Empty.clear();  size\_t real\_size = 0;  Assert::AreEqual(Empty.get\_size(), real\_size);  }  TEST\_METHOD(Push\_Back\_Empty)  {  Empty.push\_back("5");  string str = "5";  Assert::AreEqual(Empty.at(0), str);  Empty.pop\_back();  }  TEST\_METHOD(Push\_Front\_Empty)  {  Empty.push\_front("0");  string str = "0";  Assert::AreEqual(Empty.at(0), str);  }  };  TEST\_CLASS(PrefixTest)  {  prefix Test;  TEST\_METHOD(SimpleMath)  {  string res = "+ 1 2";  Test.make\_prefix("1+2");  Assert::AreEqual(Test.print\_prefix(), res);  }  TEST\_METHOD(TrigMath)  {  string res = "+ tg(pi) cos(e)";  Test.make\_prefix("tg(pi) + cos(e)");  Assert::AreEqual(Test.print\_prefix(), res);  }  TEST\_METHOD(TrigMath\_Mix)  {  string res = "+ tg(+ / pi 2 3) cos(e)";  Test.make\_prefix("tg(pi/2+3) + cos(e)");  Assert::AreEqual(Test.print\_prefix(), res);  }  TEST\_METHOD(MathWithoutBracket\_1)  {  string res = "+ + 1 \* 2 3 4";  Test.make\_prefix("1+2\*3+4");  Assert::AreEqual(Test.print\_prefix(), res);  }  TEST\_METHOD(MathWithoutBracket\_2)  {  string res = "+ \* 1 2 \* 3 4";  Test.make\_prefix("1\*2+3\*4");  Assert::AreEqual(Test.print\_prefix(), res);  }  TEST\_METHOD(MathWithoutBracket\_3)  {  string res = "+ + + 1 2 3 4";  Test.make\_prefix("1+2+3+4");  Assert::AreEqual(Test.print\_prefix(), res);  }  TEST\_METHOD(MathWithBracket\_1)  {  string res = "\* + 1 2 + 3 4";  Test.make\_prefix("(1+2)\*(3+4)");  Assert::AreEqual(Test.print\_prefix(), res);  }  TEST\_METHOD(MathWithBracket\_2)  {  string res = "- 0 + -2 3";  Test.make\_prefix("-(-2+3)");  Assert::AreEqual(Test.print\_prefix(), res);  }  };  TEST\_CLASS(Prefix\_Math\_Test)  {  prefix test;  TEST\_METHOD(SimpleMath)  {  test.make\_prefix("1+2");  string result = "3.000000";  Assert::AreEqual(test.get\_result(), result);  }  TEST\_METHOD(SimpleTrig)  {  test.make\_prefix("sin(pi/2)");  string result = "1.000000";  Assert::AreEqual(test.get\_result(), result);  }  TEST\_METHOD(SimpleMath\_With\_Brackets)  {  test.make\_prefix("(1+2)-(3+4)");  string result = "-4.000000";  Assert::AreEqual(test.get\_result(), result);  }  TEST\_METHOD(MixedMath)  {  test.make\_prefix("cos(pi)-1");  string result = "-2.000000";  Assert::AreEqual(test.get\_result(), result);  }  TEST\_METHOD(Error\_input)  {  try {  test.make\_prefix("tg(");  }  catch (const char\*)  {  Assert::AreEqual(1, 1);  }  }  TEST\_METHOD(MixedMath\_2)  {  test.make\_prefix("sqrt(2+2)-2");  string result = "0.000000";  Assert::AreEqual(test.get\_result(), result);  }  TEST\_METHOD(StrangeMath)  {  test.make\_prefix("sqrt(sin(pi^2/pi/2))+ln(e)");  string result = "2.000000";  Assert::AreEqual(test.get\_result(), result);  }  TEST\_METHOD(SimpleMath\_with\_const)  {  test.make\_prefix("e\*pi");  string result = "8.539734";  Assert::AreEqual(test.get\_result(), result);  }  };  } |

# ЗАКЛЮЧЕНИЕ

Для написания данной курсовой работы потребовались знания, полученные за третий семестр обучения. Программа успешно выполняет преобразование из инфиксной формы записи выражения в префиксную форму, а также корректно рассчитывает результат по полученному выражению. При ошибочном вводе программа сообщает об этом пользователю.