**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра САПР**

ОТЧЁТ

**по курсовой работе**

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

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

**Вариант 2.**

|  |  |  |
| --- | --- | --- |
| Студент гр. 9302 |  | Давтян С.Д. |
| Преподаватель |  | Тутуева А.В. |

Санкт-Петербург

2020

## Постановка задачи и описание реализуемого класса и методов.

Задание: 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 польская нотация введенного выражения, а также отображается результат вычисления.

Для этого мне понадобились классы «Expression», «List» (взят из предыдущих лабораторных работ и переделан под нужную реализацию), «Node».

class List

|  |  |  |
| --- | --- | --- |
| Поля | Описание | |
| Node \*head; | Указатель на начало списка | |
| Node \*tail; | Указатель на конец списка | |
| size\_t size; | Переменная, хранящая размер списка | |
| Методы | Описание | Оценка временной сложности |
| void push\_back(string) | Добавление в конец списка | O(1) |
| void push\_front(string) | Добавление в начало списка | O(1) |
| void pop\_back() | Удаление последнего элемента | O(1) |
| void pop\_front() | Удаление первого элемента | O(1) |
| void insert(int, size\_t) | Добавление элемента по индексу | O(n) |
| int at(const int) | Получение элемента по индексу | O(n) |
| void remove(size\_t) | Удаление элемента по индексу | O(n) |
| size\_t GetSize() | Получение размера списка | O(1) |
| void print\_to\_console() | Вывод элементов в консоль через разделитель | O(n) |
| bool isEmpty() | Проверка на пустоту списка | O(1) |
| void clear() | Удаление всех элементов списка | O(n) |
| void reverse | Изменение порядка элементов в списке на обратный | O(n) |
| void set(size\_t, string) | Замена элемента по индексу | O(n) |
| string top() | Получение первого элемента в списке | O(1) |

class Node

|  |  |
| --- | --- |
| Поля | Описание |
| Node \*next; | Указатель на следующий элемент списка |
| Node \*prev; | Указатель на предыдущий элемент списка |
| string data; | Строки, хранящиеся в списке |

class Expression

|  |  |
| --- | --- |
| Поля | Описание |
| List decision | Части выражения в префиксной форме записи |
| List example | Части примера для преобразования в префиксную форму записи |
| Методы | Описание |
| void parts\_of\_expression(string) | Разбиение выражения на части |
| void prefix\_form(string) | Получение префиксной ФЗ |
| string get\_result() | Нахождение результата |
| string print\_expression() | Вывод выражения в префиксной ФЗ |

Также для реализации работы мне понадобились данные функции:

|  |  |
| --- | --- |
| Функции | Описание |
| size\_t IsConst(string, size\_t) | Проверка, является ли строка константой и получение размера константы |
| bool IsNumber(string) | Проверка, является ли строка числом |
| bool IsCorrectNumber(string) | Проверка, является ли введенное число корректным |
| bool IsOperator(string, size\_t ) | Проверка, является ли строка оператором |
| size\_t IsFunction(string, size\_t) | Проверка, является ли строка функцией и получение размера функции |
| size\_t priority\_of\_operations(string) | Получение приоритета оператора |
| string to\_Function(string) | Получение результата при расчете функции |
| string to\_Operator(string, string, string) | Получение результата выражений с одним оператором |

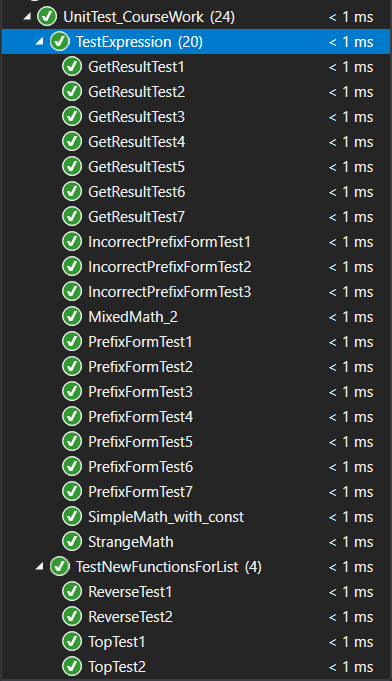
## Описание алгоритма решения.

Алгоритм:  
Пользователем вводится выражение в консоль. Далее программа считывает введенную строку. Затем данная строка разбивается на части. Получается список, элементы которого – части выражения (отдельно числа, операторы, функции), при этом выражение переворачивается. Далее данное выражение преобразуется в постфиксную форму записи и переворачивается обратно. Получается выражение в инфиксной форме записи. Оно выводится в консоль. Затем происходит расчет результата данного выражения. Рассматриваются различные варианты (если в выражении 1 элемент, 2 элемента или же больше). После этого программа выводит полученный результат в консоль.

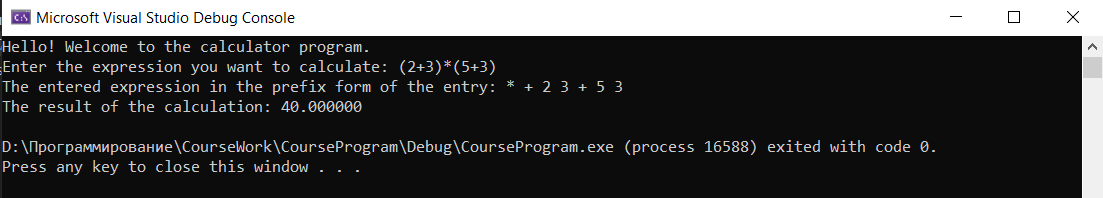
## Описание реализованных unit-тестов.

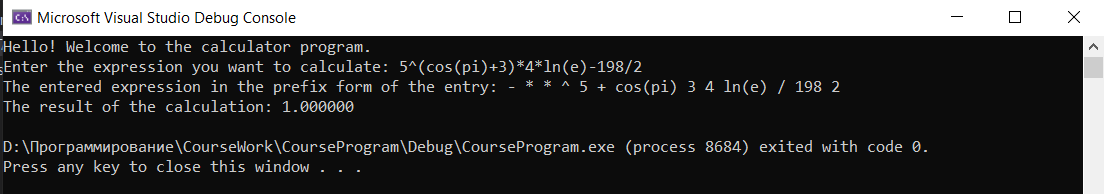
|  |  |
| --- | --- |
| Название теста | Что проверяет |
| Проверка новых функций для класса «List» | |
| ReverseTest1 | Проверка метода reverse на непустом списке. |
| ReverseTest2 | Проверка метода reverse на пустом списке. |
| TopTest1 | Проверка метода top на непустом списке. |
| TopTest2 | Проверка метода top на пустом списке. |
| Проверка методов класса «Expression» | |
| PrefixFormTest1 | Проверка метода prefix\_form на простом выражении. |
| PrefixFormTest2 | Проверка метода prefix\_form на простом выражении со скобками. |
| PrefixFormTest3 | Проверка метода prefix\_form на выражении с функцией. |
| PrefixFormTest4 | Проверка метода prefix\_form на простом выражении с унарным минусом. |
| PrefixFormTest5 | Проверка метода prefix\_form на выражении с функцией со сложным аргументом. |
| PrefixFormTest6 | Проверка метода prefix\_form на выражении с константами. |
| PrefixFormTest7 | Проверка метода prefix\_form на выражении с подряд идущими знаками. |
| IncorrectPrefixFormTest1 | Проверка на некорректный ввод. |
| IncorrectPrefixFormTest2 | Проверка на некорректный ввод. |
| IncorrectPrefixFormTest3 | Проверка на некорректный ввод. |
| GetResultTest1 | Проверка метода get\_result на простом выражении. |
| GetResultTest2 | Проверка метода get\_result на выражении с функцией и константой. |
| GetResultTest3 | Проверка метода get\_result на простом выражении со скобками. |
| GetResultTest4 | Проверка метода get\_result на выражении с функцией со сложным аргументом. |
| GetResultTest5 | Проверка метода get\_result на выражении с функцией со сложным аргументом, константой и числом. |
| GetResultTest6 | Проверка метода get\_result на выражении с функцией и числом. |
| GetResultTest7 | Проверка метода get\_result сложном математическом выражении. |

## Результат выполнения всех unit-тестов



## Примеры работы программы





## Листинг

|  |
| --- |
| List.h |
| #pragma once  #include <iostream>  #include <string>  using namespace std;  class List  {  public:  List();  ~List();  void push\_back(string data);  void push\_front(string data);  void pop\_back();  void pop\_front();  void clear();  size\_t GetSize();  string at(const int index);  bool isEmpty();  void remove(size\_t index);  void set(size\_t index, string data);  void insert(string data, size\_t index);  void reverse();  void print\_to\_console();  string top();  private:  class Node  {  public:  Node\* next;  Node\* prev;  string data;  Node(string data ="\0", Node\* next = nullptr, Node\* prev =nullptr)  {  this->data = data;  this->next = next;  this->prev = prev;  }  };  Node\* head;  Node\* tail;  size\_t size;  }; |
| List.cpp |
| #include "List.h"  // constructor  List::List()  {  size = 0;  head = nullptr;  tail = nullptr;  }  // destructor  List::~List()  {  clear();  }  // inserting an element at the end of the list  void List::push\_back(string data)  {  if (head == nullptr)  {  this->head = this->tail = new Node(data);  }  else  {  Node\* temp = new Node(data, nullptr, tail);  tail->next = temp;  tail = temp;  }  size++;  }  // inserting an element at the beginning of the list  void List::push\_front(string data)  {  if (!(isEmpty()))  {  Node\* lasthead = head;  head = new Node(data, head);  lasthead->prev = head;  }  else  {  head = new Node(data, head);  }  size++;  }  //remove the last element  void List::pop\_back()  {  if (isEmpty())  throw "Error! Linked list is Empty.";  else  {  Node\* todelete = tail;  tail = tail->prev;  delete todelete;  size--;  }  }  // remove the first element  void List::pop\_front()  {  if (head != nullptr) {  Node\* temp = this->head;  head = head->next;  delete temp;  size--;  }  else throw "Error! Linked list is Empty.";  }  // clear list  void List::clear()  {  while (size) {  pop\_front();  }  }  // getting list size  size\_t List::GetSize()  {  return size;  }  // output the list to the console  void List::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 "Error! List is empty.";  }  // getting an element by index  string List::at(const int index)  {  if (index >= size || index < 0)  throw "Error! Incorrect input.";  int counter = 0;  Node\* current = head;  while (current != nullptr)  {  if (counter == index)  return current->data;  current = current->next;  counter++;  }  }  // checking the list for emptiness  bool List::isEmpty()  {  if (head == nullptr)  return true;  else return false;  }  // replacing the element by index with the passed element  void List::set(size\_t index, string data)  {  if (index >= size || index < 0)  throw "Error! Incorrect input.";  Node\* cursor = head;  for (size\_t i = 0; i < index; i++)  cursor = cursor->next;  cursor->data = data;  }  // insert into an arbitrary place in the list by index  void List::insert(string data, size\_t index)  {  if (!isEmpty())  {  if (index >= size || index < 0)  throw "Error! Incorrect input.";  }  else  {  if (index != size)  throw "Error! Incorrect input.";  }  if (index == 0)  {  push\_front(data);  }  else  {  if (index == size)  push\_back(data);  else {  if (index < (size - 1) / 2)  {  Node\* previous = this->head;  for (size\_t i = 0; i < index - 1; i++)  {  previous = previous->next;  }  previous->next = new Node(data, previous->next, previous);  }  else  {  Node\* previous = this->tail;  for (size\_t i = size - 1; i > index - 1; i--)  {  previous = previous->prev;  }  previous->next = new Node(data, previous->next, previous);  }  size++;  }  }  }  // кeverse the order of items in the list  void List::reverse()  {  if (head == nullptr)  throw "Error! Linked list is empty.";  if (!head || !head->next)  return;  tail = head;  Node\* temp = nullptr;  Node\* current = head;  while (current != nullptr)  {  temp = current->prev;  current->prev = current->next;  current->next = temp;  current = current->prev;  }  head = temp->prev;  }  // deleting an element by index  void List::remove(size\_t index)  {  if (index >= size || index < 0)  throw "Error! Incorrect input.";  if (index == 0)  this->pop\_front();  else if (index == size - 1)  this->pop\_back();  else {  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--;  }    }  // getting the first item in the list  string List::top()  {  if (head)  return head->data;  else throw "Error! Stack is empty.";  } |
| Expression.h |
| #pragma once  #include "List.h"  using namespace std;  class Expression {  public:  void prefix\_form(string infix);  string get\_result();  string print\_expression();  private:  List decision;  List example;  void parts\_of\_expression(string example);  }; |
| Expression.cpp |
| #include "Expression.h"  #include <string>  #include <math.h>  #define pi 3.1415926535897932384626433832795  #define e 2.7182818284  using namespace std;  // checking whether a string is a constant and getting the size of the constant  size\_t IsConst(string potencial\_const, size\_t index = 0)  {  if (potencial\_const[index] == 'p' && potencial\_const[index + 1] == 'i') //pi size=2  return 2;  else if (potencial\_const[index] == 'e') //e size=1  return 1;  return 0;  }  // checking whether the entered number is correct  bool IsCorrectNumber(string number)  {  size\_t count\_of\_signs = 0;  for (int i = 0; i < number.size(); i++)  {  if (number[i] == ',')  count\_of\_signs++;  }  if (count\_of\_signs > 1) return false;  else return true;  }  // checking whether a string is an operator  bool IsOperator(string potencial\_operaor, size\_t index = 0)  {  if (potencial\_operaor[index] == ')' || potencial\_operaor[index] == '(')  return false;  return (!isalpha(potencial\_operaor[index]) && !isdigit(potencial\_operaor[index]));  }  // checking whether a string is a number  bool IsNumber(string potencial\_number)  {  if ((potencial\_number[0] >= '0' && potencial\_number[0] <= '9') || (potencial\_number[0] == '-' && potencial\_number[1] >= '0' && potencial\_number[1] <= '9') || IsConst(potencial\_number))  {  if(IsCorrectNumber(potencial\_number))  return true;  }  return false;  }  // checking whether a string is a function and getting the size of the function  size\_t IsFunction(string potencial\_function, size\_t index = 0)  {  if (potencial\_function[index] == 's' && potencial\_function[index + 1] == 'i' && potencial\_function[index + 2] == 'n') //sin size=3  return 3;  else if (potencial\_function[index] == 'c' && potencial\_function[index + 1] == 'o' && potencial\_function[index + 2] == 's') //cos size=3  return 3;  else if (potencial\_function[index] == 't' && potencial\_function[index + 1] == 'g') //tg size=2  return 2;  else if (potencial\_function[index] == 'c' && potencial\_function[index + 1] == 't' && potencial\_function[index + 2] == 'g') //ctg size=3  return 3;  else if (potencial\_function[index] == 'l' && potencial\_function[index + 1] == 'n') //ln size=2  return 2;  else if (potencial\_function[index] == 'l' && potencial\_function[index + 1] == 'o' && potencial\_function[index + 2] == 'g') //log size=3  return 3;  else if (potencial\_function[index] == 's' && potencial\_function[index + 1] == 'q' && potencial\_function[index + 2] == 'r' && potencial\_function[index + 3] == 't') //sqrt size=4  return 4;  else if (potencial\_function[index] == 's' && potencial\_function[index + 1] == 'q' && potencial\_function[index + 2] == 'r') //sqr size=3  return 3;  return 0;  }  // getting operator priority  size\_t priority\_of\_operators(string oper)  {  if (IsNumber(oper) || IsFunction(oper, 0) || IsConst(oper, 0) || (oper[0] == '+' || oper[0] == '-'))  return 0;  if (oper[0] == '\*' || oper[0] == '/')  return 1;  return 2;  }  // getting the prefix form of a record  void Expression::prefix\_form(string infix)  {  parts\_of\_expression(infix);  List string\_stack;  for (size\_t i = 0; i < example.GetSize(); i++)  {  if (IsNumber(example.at(i)) || IsFunction(example.at(i)))  decision.push\_back(example.at(i));  else if (IsOperator(example.at(i)))  {  if (string\_stack.isEmpty())  string\_stack.push\_front(example.at(i));  else if (string\_stack.top() == "(")  string\_stack.push\_front(example.at(i));  else if (priority\_of\_operators(example.at(i)) >= priority\_of\_operators(string\_stack.top()))  string\_stack.push\_front(example.at(i));  else  {  while (!string\_stack.isEmpty() && string\_stack.top() != "(" && priority\_of\_operators(example.at(i)) < priority\_of\_operators  (string\_stack.top()))  {  decision.push\_back(string\_stack.top());  string\_stack.pop\_front();  }  string\_stack.push\_front(example.at(i));  }  }  else if (example.at(i) == "(")  string\_stack.push\_front(example.at(i));  else if (example.at(i) == ")")  {  while (!string\_stack.isEmpty() && string\_stack.top() != "(")  {  decision.push\_back(string\_stack.top());  string\_stack.pop\_front();  }  string\_stack.pop\_front();  }  }  while (!string\_stack.isEmpty())  {  decision.push\_back(string\_stack.top());  string\_stack.pop\_front();  }  decision.reverse();  example.clear();  }  //Output of an expression in the prefix form of a record  string Expression::print\_expression()  {  if (decision.GetSize() == 0)  {  decision.clear();  throw "Error! Expression is empty.";  }  string result;  for (size\_t i = 0; i < decision.GetSize(); i++)  result += decision.at(i) + " ";  if (result.size() != 0)  result.pop\_back();  return result;  }  // splitting an expression into parts  void Expression::parts\_of\_expression(string example)  {  string tmp;  int check;  size\_t check\_frac;  for (int i = 0; i < example.size(); i++)  {  tmp.clear();  check = 0;  if (example[i] == ' ')  continue;  while (((example[i] >= '0' && example[i] <= '9') || (example[i] == ',')) && i < example.size())  {  tmp += example[i];  i++;  }  if (tmp.size() != 0)  {  this->example.push\_back(tmp);  i--;  continue;  }  if (example[i] == '(')  tmp += ")";  if (example[i] == ')')  tmp += "(";  if (IsOperator(example,i))  tmp += example[i];  if (tmp.size() != 0)  {  this->example.push\_back(tmp);  continue;  }  size\_t size\_fun = IsFunction(example, i);  if (size\_fun > 0)  {  for (size\_t index\_fun = i; index\_fun < i + size\_fun; index\_fun++)  tmp.push\_back(example[index\_fun]);  size\_fun--;  do  {  size\_fun++;  if (example[i + size\_fun] == '(')  check++;  if (example[i + size\_fun] == ')')  check--;  tmp += example[i + size\_fun];  } while (check != 0 && example.size() > i + size\_fun);  if (check != 0)  throw "Error! Incorrect input.";  this->example.push\_back(tmp);  i += size\_fun;  continue;  }  size\_t size\_const = IsConst(example , i);  if (size\_const>0)  {  for (size\_t index\_const = i; index\_const < i + size\_const; index\_const++)  tmp.push\_back(example[index\_const]);  this->example.push\_back(tmp);  i += size\_const - 1;  continue;  }  throw "Error! Unknown symbols.";  }  for (size\_t i = 0; i < this->example.GetSize(); i++)  {  string temp;  if (i == 0 && this->example.at(i) == "-" && this->example.at(i + 1) != ")" && !IsFunction(this->example.at(i + 1)) && !IsOperator(this->example.at(i + 1)))  {  temp += this->example.at(i) + this->example.at(i + 1);  this->example.remove(i + 1);  this->example.set(i, temp);  }  else if (i == 0 && this->example.at(i) == "-" && (this->example.at(i + 1) == ")" || IsFunction(this->example.at(i + 1))))  {  this->example.push\_front("0");  i++;  }  else if (this->example.at(i) == ")" && i + 1 < this->example.GetSize() && this->example.at(i + 1) == "-" && !IsFunction(this->example.at(i + 2)))  {  temp += this->example.at(i + 1) + this->example.at(i + 2);  this->example.remove(i + 1);  this->example.set(i + 1, temp);  }  else if (this->example.at(i) == ")" && i + 1 < this->example.GetSize() && this->example.at(i + 1) == "-" && IsFunction(this->example.at(i + 2)))  {  this->example.insert("0", i + 1);  }  }  this->example.reverse();  }  // getting the result when calculating a function  string to\_Function(string function)  {  string fun, arg;  double result = 0, number;  char check = 0;  for (size\_t i = 0; i < function.size() - 1; i++)  {  if (function[i] == '(' && check == 0)  {  check++;  continue;  }  if (check)  arg += function[i];  else fun += function[i];  }    if (arg == "pi")  number = pi;  else if (arg == "e")  number = e;  else if ((arg != "pi") && (arg != "e"))  {  Expression Argument;  Argument.prefix\_form(arg);  arg = Argument.get\_result();  number = stod(arg);  }  else number = stod(arg);  if (fun == "cos") result = cos(number);  else if (fun == "sin") result = sin(number);  else if (fun == "tg") result = tan(number);  else if (fun == "ctg") result = 1 / tan(number);  else if (fun == "ln") result = log(number);  else if (fun == "log") result = log10(number);  else if (fun == "sqrt") result = sqrt(number);  else if (fun == "sqr") result = pow(number, 2);  else throw "Error! Incorrect function or argument.";  return to\_string(result);  }  // getting the result of expressions with a single operator  string to\_Operator(string oper, string number1, string number2)  {  double result = 0, num1, num2;  if (number1 == "pi")  num1 = pi;  else if (number1 == "e")  num1 = e;  else num1 = stod(number1);  if (number2 == "pi")  num2 = pi;  else if (number2 == "e")  num2 = e;  else num2 = stod(number2);  if (oper == "+") result = num1 + num2;  if (oper == "-") result = num1 - num2;  if (oper == "\*") result = num1 \* num2;  if (oper == "/") result = num1 / num2;  if (oper == "^") result = pow(num1, num2);  return to\_string(result);  }  // finding the result  string Expression::get\_result()  {  string result;  size\_t check = 0;  if (decision.GetSize() == 0)  result = "0";  else if (decision.GetSize() == 1)  {  if (IsFunction(decision.at(0)))  decision.set(0, to\_Function(decision.at(0)));  else if (IsConst(decision.at(0)))  if (decision.at(0) == "pi")  result = to\_string(pi);  else result = to\_string(e);  else if (decision.at(0) == "-pi")  result = to\_string(-1 \* pi);  else if (decision.at(0) == "-e")  result = to\_string(-1 \* e);  else if (IsNumber(decision.at(0)))  result = decision.at(0);  else  {  decision.print\_to\_console();  decision.clear();  throw "Error! Incorrect function.";  }  }  else if (decision.GetSize() == 2)  {  if ((decision.at(0) != "-" && decision.at(0) != "+") || (IsNumber(decision.at(1))))  {  decision.print\_to\_console();  decision.clear();  throw "Error! Extra operator.";  }  if (IsFunction(decision.at(1)) && decision.at(0) == "-")  {  decision.set(0, to\_string(-1 \* stod(to\_Function(decision.at(1)))));  }  else if (IsFunction(decision.at(1)) && decision.at(0) == "+")  {  decision.set(0, to\_Function(decision.at(1)));  }  decision.pop\_back();  }  else  while (decision.GetSize() > 1)  {  check = 0;  for (size\_t i = 0; i < decision.GetSize(); i++)  {  if (decision.GetSize() - i >= 3 && IsOperator(decision.at(i)) && IsNumber(decision.at(i + 1)) && IsNumber(decision.at(i + 2)))  {  decision.set(i, to\_Operator(decision.at(i), decision.at(i + 1), decision.at(i + 2)));  decision.remove(i + 1);  decision.remove(i + 1);  check++;  }  else if (IsFunction(decision.at(i)))  {  decision.set(i, to\_Function(decision.at(i)));  check++;  }  else if (decision.GetSize() == 2 && IsFunction(decision.at(1)) && decision.at(0) == "-")  {  check++;  decision.set(0, to\_string(-1 \* stod(to\_Function(decision.at(1)))));  decision.pop\_back();  }  }  if (check == 0)  {  decision.clear();  throw "Error! Extra operator or number.";  }  }  result = decision.at(0);  decision.clear();  return result;  } |
| UnitTest\_CourseWork.cpp |
| #include "pch.h"  #include "CppUnitTest.h"  #include "../CourseProgram/List.cpp"  #include "../CourseProgram/Expression.cpp"  using namespace Microsoft::VisualStudio::CppUnitTestFramework;  namespace UnitTest\_CourseWork  {  TEST\_CLASS(TestNewFunctionsForList)  {  public:  TEST\_METHOD(ReverseTest1)  {  List test;  test.push\_back("a");  test.push\_back("b");  test.push\_back("c");  test.reverse();  Assert::IsTrue(test.at(0) == "c");  }  TEST\_METHOD(ReverseTest2)  {  List test;  try  {  test.reverse();  }  catch (const char\* error)  {  Assert::AreEqual(error, "Error! Linked list is empty.");  }  }  TEST\_METHOD(TopTest1)  {  List test;  test.push\_back("a");  test.push\_back("b");  test.push\_back("c");  Assert::IsTrue(test.top() == "a");  }  TEST\_METHOD(TopTest2)  {  List test;  try  {  test.top();  }  catch (const char\* error)  {  Assert::AreEqual(error, "Error! Stack is empty.");  }  }  };  TEST\_CLASS(TestExpression)  {  Expression Prefix;  TEST\_METHOD(PrefixFormTest1)  {  Prefix.prefix\_form("2+5");  string test = "+ 2 5";  Assert::AreEqual(Prefix.print\_expression(), test);  }  TEST\_METHOD(PrefixFormTest2)  {  Prefix.prefix\_form("(5+3)\*(2+3)");  string test = "\* + 5 3 + 2 3";  Assert::AreEqual(Prefix.print\_expression(), test);  }  TEST\_METHOD(PrefixFormTest3)  {  Prefix.prefix\_form("tg(5)+3");  string test = "+ tg(5) 3";  Assert::AreEqual(Prefix.print\_expression(), test);  }  TEST\_METHOD(PrefixFormTest4)  {  Prefix.prefix\_form("-6+5");  string test = "+ -6 5";  Assert::AreEqual(Prefix.print\_expression(), test);  }  TEST\_METHOD(PrefixFormTest5)  {  Prefix.prefix\_form("tg(pi/2)+3");  string test = "+ tg(pi/2) 3";  Assert::AreEqual(Prefix.print\_expression(), test);  }  TEST\_METHOD(PrefixFormTest6)  {  Prefix.prefix\_form("pi\*e");  string test = "\* pi e";  Assert::AreEqual(Prefix.print\_expression(), test);  }  TEST\_METHOD(PrefixFormTest7)  {  Prefix.prefix\_form("5\*25\*125\*625");  string test = "\* \* \* 5 25 125 625";  Assert::AreEqual(Prefix.print\_expression(), test);  }  TEST\_METHOD(IncorrectPrefixFormTest1)  {  try  {  Prefix.prefix\_form("tg(23");  }  catch (const char\* error)  {  Assert::IsTrue(1);  }  }  TEST\_METHOD(IncorrectPrefixFormTest2)  {  try  {  Prefix.prefix\_form("tg(23)+++3");  }  catch (const char\* error)  {  Assert::IsTrue(1);  }  }  TEST\_METHOD(IncorrectPrefixFormTest3)  {  try  {  Prefix.prefix\_form("avc");  }  catch (const char\* error)  {  Assert::IsTrue(1);  }  }  TEST\_METHOD(GetResultTest1)  {  string test = "4.000000";  Prefix.prefix\_form("2+2");  Assert::AreEqual(Prefix.get\_result(), test);  }  TEST\_METHOD(GetResultTest2)  {  string test = "-1.000000";  Prefix.prefix\_form("cos(pi)");  Assert::AreEqual(Prefix.get\_result(), test);  }  TEST\_METHOD(GetResultTest3)  {  string test = "40.000000";  Prefix.prefix\_form("(5+3)\*(2+3)");  Assert::AreEqual(Prefix.get\_result(), test);  }  TEST\_METHOD(GetResultTest4)  {  string test = "1.000000";  Prefix.prefix\_form("sin(pi/2)");  Assert::AreEqual(Prefix.get\_result(), test);  }  TEST\_METHOD(GetResultTest5)  {  string test = "2.000000";  Prefix.prefix\_form("sin(pi/2) + cos(0)");  Assert::AreEqual(Prefix.get\_result(), test);  }  TEST\_METHOD(GetResultTest6)  {  string test = "1.000000";  Prefix.prefix\_form("sqr(2\*5)-99");  Assert::AreEqual(Prefix.get\_result(), test);  }  TEST\_METHOD(GetResultTest7)  {  string test = "1.000000";  Prefix.prefix\_form("5^(cos(pi)+3)\*4\*ln(e)-198/2");  Assert::AreEqual(Prefix.get\_result(), test);  }  };    } |
| main.cpp |
| #include <iostream>  #include <fstream>  #include "Expression.h"  using namespace std;  int main()  {  Expression expression;  string s;  cout << "Hello! Welcome to the calculator program.\nEnter the expression you want to calculate: ";  getline(cin, s);  try  {  expression.prefix\_form(s);  cout << "The entered expression in the prefix form of the entry: " << expression.print\_expression() << endl;  cout << "The result of the calculation: " << expression.get\_result() << endl;  }  catch (const char\* error)  {  cout << error << endl;  }  } |