МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ

РОССИЙСКОЙ ФЕДЕРАЦИИ

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

**(ННГУ)**

**Институт информационных технологий, математики и механики**

ЛАБОРАТОРНАЯ РАБОТА

на тему:

**«ВЫЧИСЛЕНИЕ АРИФМЕТИЧЕСКИХ ВЫРАЖЕНИЙ НА СТЕКАХ»**

**Выполнил(а):** студент(ка) группы 3822Б1ФИ1

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ / Шпынов Н.А. /

Подпись

**Проверил:** к.т.н., доцент каф. ВВиСП

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ / Кустикова В.Д. /

Подпись

Нижний Новгород  
2023

**Оглавление**

[Введение 1](#_Toc14127)

[1 Постановка задачи 2](#_Toc16327)

[2 Руководство пользователя 3](#_Toc23897)

[2.1 Приложение для демонстрации работы стеков 3](#_Toc4680)

[2.2 Приложение для демонстрации работы постфиксной формы 4](#_Toc28541)

[3 Руководство программиста 6](#_Toc13574)

[3.1 Использованные алгоритмы 6](#_Toc6497)

[3.2 Описание классов 9](#_Toc10266)

[Заключение 17](#_Toc30847)

[Литература 18](#_Toc16852)

[Приложения 19](#_Toc26037)

# Введе**н**ие

Запись арифметического выражения в постфиксной форме - один из самых удобных способов его представления. Ее смысл заключается в размещении операторов перед операторами, что позволяет точно установить порядок выполнения операций, и, как следствие, дает возможность машине удобно и быстро работать с ним.

Для реализации представления арифметического выражения в постфиксной форме часто используется стек.

Стек - [абстрактный тип данных](https://ru.wikipedia.org/wiki/%D0%90%D0%B1%D1%81%D1%82%D1%80%D0%B0%D0%BA%D1%82%D0%BD%D1%8B%D0%B9_%D1%82%D0%B8%D0%BF_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85" \o "Абстрактный тип данных), представляющий собой [список элементов](https://ru.wikipedia.org/wiki/%D0%A1%D0%BF%D0%B8%D1%81%D0%BE%D0%BA_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)" \o "Список (информатика)), организованных по принципу [LIFO](https://ru.wikipedia.org/wiki/LIFO" \o ") (последний пришёл - первый вышел).

# Постановка задачи

Цель – реализовать представление механизма, для преобразования стандартной записи арифметического выражения в постфиксную с помощью стека.

Задачи:

1. Разработка структуры данных стека.
2. Разработка представления класса. Для представления арифметического выражения в постфиксной форме, следует создать соответсвующий класс.
3. Определение операций. Разработать алгоритмы и функции для выполнения операций над арифметическим выражением, таких как преобразование в постфиксную форму, вычисление результата выражения и т.д.
4. Тестирование и отладка. Как и при любой разработке программного обеспечения, в конце необходимо провести тестирование и отладку кода. Следует обнаружить и исправить возможные ошибки и проблемы, а также убедиться в правильности функционирования.

# Руководство пользователя

## Приложение для демонстрации работы стеков

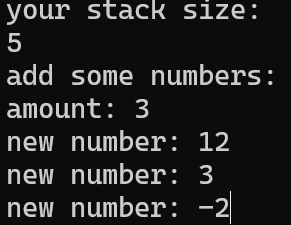
1. Запустить sample\_stack.exe. В результате появится следующее окно (рис. 1):



1. Основное окно приложения

На этом шаге потребуется ввести максимальный размер вашего стека

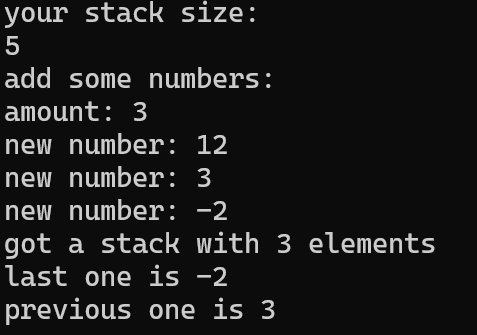
1. После ввода его размера, следует ввести несколько чисел (рис. 2):



1. Ввод длин

Вводить значения надо по порядку через пробел или Enter, предварительно указав, сколько их будет.

1. Программа выводит размер стека, последний и предпоследний полученные элементы (рис. 3):



1. Вывод

## Приложение для демонстрации работы постфиксной формы

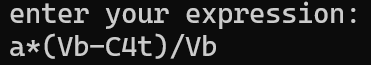
1. Запустить sample\_PN.exe. В результате появится следующее окно   
   (рис. 4):



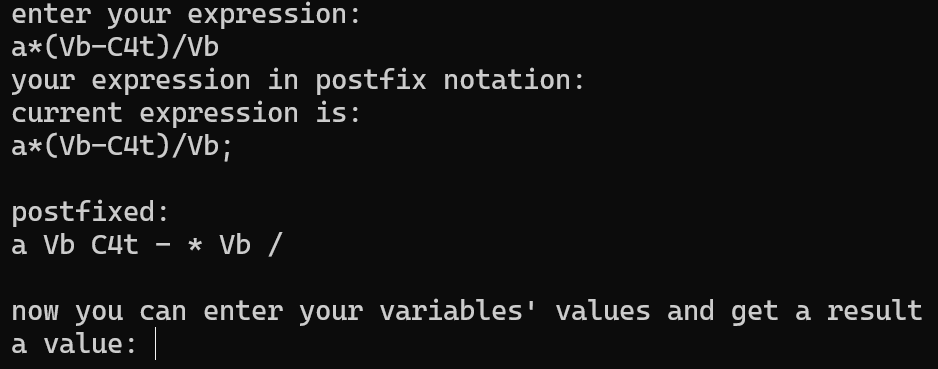
1. Основное окно приложения

Требуется ввести арифметическое выражение. Использовать можно операторы +, -, \*, /, скобки, а также константы и переменные, состоящие из латинских букв (с учетом заглавных) и цифр (не может стоять в начале переменной).

1. После ввода выражения (рис. 5):



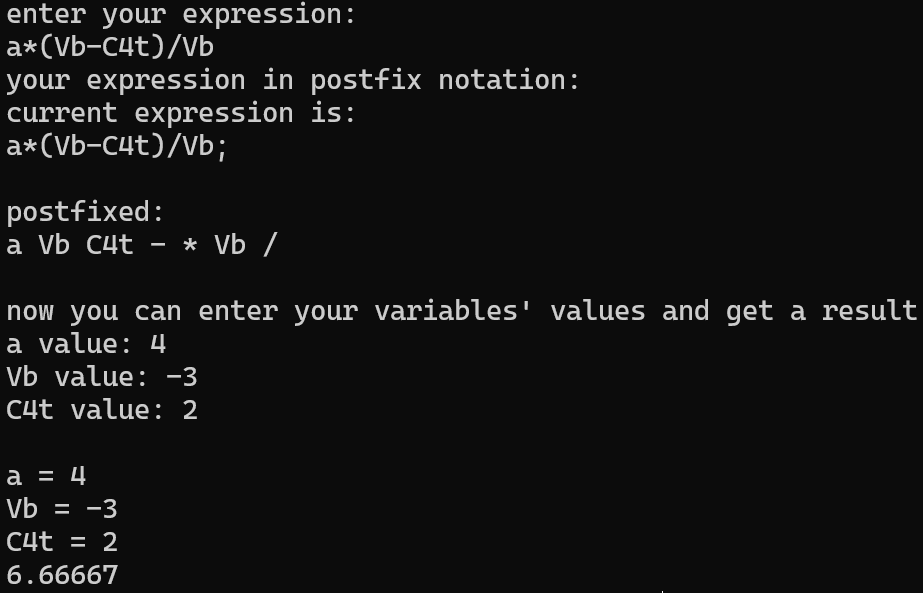
1. Ввод выражения
2. Далее программа для проверки выведет полученные ею данные и попросит значения переменных для того, чтобы посчитать частное значение выражения (рис. 6):



1. Проверка

Переменные вводятся по очереди, в порядке их появления в выражении.

1. Программа считает результат и выводит его (рис. 7).



1. Итог

# Руководство программиста

## Использованные алгоритмы

### Стек

Стек – [абстрактный тип данных](https://ru.wikipedia.org/wiki/%D0%90%D0%B1%D1%81%D1%82%D1%80%D0%B0%D0%BA%D1%82%D0%BD%D1%8B%D0%B9_%D1%82%D0%B8%D0%BF_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85" \o "Абстрактный тип данных), представляющий собой [список элементов](https://ru.wikipedia.org/wiki/%D0%A1%D0%BF%D0%B8%D1%81%D0%BE%D0%BA_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)" \o "Список (информатика)), организованных по принципу [LIFO](https://ru.wikipedia.org/wiki/LIFO" \o ") (последний пришёл - первый вышел).

Поддерживает операции возвращения значения верхнего элемента, его удаление, добавление элемента наверх, проверка стека на полноту и пустоту.

1. Верхний элемент

|  |
| --- |
| 3 |
| 4 |
| -2 |
| 2 |

Фунция .top() вернет 3

1. Удаление верхнего элемента

.del():

|  |  |  |
| --- | --- | --- |
| 3 |  |  |
| 4 | → | 4 |
| -2 |  | -2 |
| 2 |  | 2 |

1. Добавление элемента наверх

.add(3):

|  |  |  |
| --- | --- | --- |
|  |  | 3 |
| 4 | → | 4 |
| -2 |  | -2 |
| 2 |  | 2 |

4)

Проверка на пустоту/полноту

|  |
| --- |
|  |
|  |
|  |
|  |

.is\_empty() вернет true, .is\_full вернет false

|  |
| --- |
| 11 |
| 1 |
| 1 |
| 1 |

.is\_empty() вернет false, .is\_full вернет true

|  |
| --- |
|  |
| 1 |
| 1 |
| 1 |

.is\_empty() вернет false, .is\_full вернет false

### Постфиксная форма

Постфиксная форма - представление арифметического выражения так, что операторы размещаются перед операндами например выражение

(a\*b-c)/(d-a)-b

будет представлено как

ab\*c-da-/b-

Постфиксная форма позволяет точно установить порядок выполнения операций, и, как следствие, даёт возможность машине удобно и быстро работать с ним.

Алгоритм преобразования выражения в постфиксную запись:

1. Для операций вводится приоритет: «\*», «/» – 3, «+», «-» – 2, «(» – 1. Для «)» приоритета нет.
2. Исходное выражение просматривается слева направо:
   1. Операнды по мере их появления помещаются в один стек
   2. Символы операций и левой скобки помещаются в другой стек следующим образом: если новая операция имеет меньший приоритет операция на верхушке второго стека, все операции в нем с большим или равным приоритетом перекладывается в первый стек, после чего текущая операция кладётся во второй. В случае «)» из второго стека изымаются элементы в первый до тех пор, пока не придёт левая скобка или стек не опустошится, левая скобка в стек не переносится, она удаляется.

Алгоритм вычисления значения

1. Читаются знаки слева направо до конца последовательности.
   1. Если пришёл операнд, его значение помещается в соответствующий стек.
   2. Если пришел оператор, из стека операндов берутся 2 верхних операнда, выполняется указанная операция, результат возвращается стек.
2. Когда будет достигнут конец строки, в стеке будет лежать только ответ

## Описание классов

### Класс TStack

template <typename T> class TStack {

private:

int max\_size;

int current\_amount;

T\* elements;

public:

TStack();

TStack(int size = 20);

~TStack();

bool is\_full() const;

bool is\_empty() const;

T& top();

void add(const T& elem);

void del() ;

};

*Поля:*

max\_size – максимально допустимый размер стека.

current\_amount – текущее количество элементов в стеке

elements – массив элементов слека

*Конструкторы:*

**TStack()** – конструктор по умолчанию.

**TStack(int** size = 20) – конструктор с параметром.

Назначение: создание стека с заданной максимальной длиной, 2 по умолчанию

Входные данные: size – длина вектора.

*Деструктор*:

~TStack()

*Методы:*

bool is\_full() const – проверка на полноту.

Назначение: проверяет стек, полон ли он.

Входные данные: нет.

Выходные данные: true если полон, false если нет.

bool is\_empty() const – проверка на пустоту.

Назначение: проверяет стек, пуст ли он.

Входные данные: нет.

Выходные данные: false если полон, true если нет.

T& top() – получение верхнего элемента.

Назначение: возвращает верхний элемент в стеке.

Входные данные: нет.

Выходные данные: элемент заданного типа данных.

void add(const T& elem) – добавление элемента в стек.

Назначение: добавление элемента наверх стека.

Входные данные: elem – элемент.

Выходные данные: нет.

void del() – удаление элемента из стека

Назначение: удаление из стека послежнего элемента.

Входные данные: нет.

Выходные данные: нет.

*Операторы:*

Нет

### Класс tPostfix

class tPostfix {

private:

string expression;

vector<operand> vars;

vector<string> expression\_refined;

vector<string> expression\_postfixed;

int get\_operator\_priority(const string op);

bool is\_operator(const int i) const;

bool is\_operator(char c) const;

bool is\_operator(const string c) const;

void divide\_string();

void add\_a\_new\_variable(const string n);

double operation(double a, char op, double b);

void create\_postfixed\_expression(TStack<string>& S);

public:

void get\_values\_for\_variables();

void postfix\_notation();

int get\_variables\_amount(){

return vars.size();

}

double calculate\_the\_result();

tPostfix(const string& S);

friend ostream& operator<<(ostream& out, const tPostfix& p) {

out << "current expression is:" << endl

<< p.expression << ";" << endl << endl;

out << "postfixed:" << endl;

for (int i = 0; i < p.expression\_postfixed.size(); i++)

out << p.expression\_postfixed[i] << " ";

out << endl;

return out;

}

bool evaluate\_operator(const int i) const {

return is\_operator(i);

}

bool evaluate\_operator(char i) const {

return is\_operator(i);

}

bool evaluate\_operator(const string i) const {

return is\_operator(i);

}

double test\_operation(double a, char op, double b) {

return operation(a, op, b);

}

bool check\_postfixed(vector<string> a) {

return (a == expression\_postfixed);

}

bool check\_refined(vector<string> a) {

return (a == expression\_refined);

}

};

*Поля*:

expression – полученное выражение.

vars – переменные в выражении.

expression\_refined – выражение, разделённое на операторы и операнды.

expression\_postfixed – выражение в постфиксной форме.

*Конструкторы*:

tPostfix(const string& S)– конструктор копирования

Назначение: создание элемента класса tPostfix, равного данному.

Входные данные: s – элемент.

*Методы*:

int get\_operator\_priority(const string op)– приоритет оператора.

Назначение: возвращает приоритет полученного на вход оператора.

Входные данные: op – оператор.

Выходные данные: приоритет оператора.

bool is\_operator(const int i) const – проверка знака на оператор.

Назначение: проверяет символ из expression, является ли он каким либо оператором.

Входные данные: **i** – номер элемента в строке.

Выходные данные: true или false в зависимости от того, является ли оператором этот символ.

bool is\_operator(char c) const – проверка знака на оператор.

Назначение: проверяет полученный символ, является ли он каким либо оператором.

Входные данные: с – элемент.

Выходные данные: true или false в зависимости от того, является ли оператором этот символ.

bool is\_operator(const string c) const – проверка знака на оператор.

Назначение: проверяет полученный символ, является ли он каким либо оператором, если по какой то причине он был передан в формате строки.

Входные данные: s – строка.

Выходные данные: true или false в зависимости от того, является ли оператором этот символ.

void divide\_string()– деление строки на операторы и операнды.

Назначение: делит полученную строку, разделяя константы, операторы и операнды.

Входные данные: нет.

Выходные данные: нет.

void add\_a\_new\_variable(const string n)– добавление константы.

Назначение: добавляет константу в **vars.**

Входные данные: **n** – строка с именем константы.

Выходные данные: нет.

double operation(double a, char op, double b)– проведение операции.

Назначение: проводит заданную операцию с двумя операндами.

Входные данные: **a, b** – операнды, op – оператор.

Выходные данные: результат подсчета.

void create\_postfixed\_expression(TStack<string>&S) – разворачивает стек постфиксного выражения, полученного задом наперед.

Назначение: получает стек постфиксного выражения, вычисленного задом наперед, разворачивает его и записывает результат.

Входные данные: **s** – стек.

Выходные данные: нет.

void get\_values\_for\_variables() – получает значения переменных.

Назначение: получает от пользователя значения переменных в выражении.

Входные данные: нет.

Выходные данные: нет.

void postfix\_notation() – создает постфиксную форму выражения.

Назначение: получает уже разделенное выражение и создаёт из него постфиксную форму.

Входные данные: нет.

Выходные данные: нет.

int get\_variables\_amount() – получение количества переменных.

Назначение: считает текущее количество переменных в выражении.

Входные данные: нет.

Выходные данные: количество переменных.

double calculate\_the\_result() – считает значение выражения.

Назначение: после получения программой значения всех переменных, считает значение выражения.

Входные данные: нет.

Выходные данные: результат вычислений.

bool evaluate\_operator(const int i) const – узнает, есть ли данный оператор в tPostfix на данный момент.

Назначение: позволяет узнать, есть ли данный оператор в **tPostfix** на данный момент, нужен для отладки.

Входные данные: **i** – номер элемента в строке.

Выходные данные: true или false в зависимости от того, является ли оператором этот символ.

bool evaluate\_operator(char с) const – узнает, есть ли данный оператор в tPostfix на данный момент.

Назначение: позволяет узнать, есть ли данный оператор в **tPostfix** на данный момент, нужен для отладки

Входные данные: с – элемент.

Выходные данные: true или false в зависимости от того, является ли оператором этот символ.

bool evaluate\_operator(const string с) const – узнает, есть ли данный оператор в tPostfix на данный момент.

Назначение: позволяет узнать, есть ли данный оператор в **tPostfix** на данный момент, нужен для отладки

Входные данные: с – строка.

Выходные данные: true или false в зависимости от того, является ли оператором этот символ.

double test\_operation(double a, char op, double b) – проверяет корректность вычисления.

Назначение: проверяет, правильно ли вычислится выражение программой, нужен для отладки

Входные данные: **a, b** – операнды, op – оператор.

Выходные данные: true или false в зависимости от того, правильно ли был произведен подсчет.

bool check\_postfixed(vector<string> a) – проверяет корректность перевода в постфиксную форму.

Назначение: проверяет, правильно ли переведется выражение, нужен для отладки.

Входные данные: **a** – выражение в постфиксной форме, которое должно быть в соответствующем поле.

Выходные данные: true или false в зависимости от того, правильные ли данные лежат в поле.

bool check\_refined(vector<string> a)) – проверяет корректность разделения выражения на составляющие.

Назначение: проверяет, правильно ли разделится выражение, нужен для отладки

Входные данные: **a** – разобранное на составляющие выражение, которое должно быть в соответствующем поле класса

Выходные данные: true или false в зависимости от того, правильные ли данные лежат в поле.

*Операторы:*

friend ostream& operator<<(ostream& out, const tPostfix& p)

Назначение: вывод постфиксной формы выражения

Входные данные: **out** – выходной поток, **p** – элемент класса **tPostfix** на вывод

Выходные данные: постфиксная форма выражения в консоли

# Заключение

Реализация работы с выражениями при помощи постфиксной формы оказалась эффективным подходом, позволяющим удобно хранить и оперировать выражениями и считать их частные значения. Этот метод легко был создан на основе стеков, функционал которых оказался крайне полезным для выполнения задачи.

# Литература

1. Стек в c++

[https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D0%B5%D0%BA].

1. Алгоритм преобразования в постфиксную форму

[https://www.cyberforum.ru/cpp-beginners/thread444330.html]

1. Справочник по С++ [[https://prog-cpp.ru/cpp](https://prog-cpp.ru/cpp/)].
2. Справочник по С++ [[https://en.cppreference.com/w](https://en.cppreference.com/w/)].

# Приложения

### Реализация полей класса TVector:

template<typename T> TVector<T>::TVector(int s, int index) {

    if (s <= 0)

        throw "got negative size";

    size = s;

    start\_index = index;

    elements = new T[size];

}

template<typename T> TVector<T>::TVector(const TVector<T>& v) {

    size = v.size;

    start\_index = v.start\_index;

    elements = new T[size];

    for (int i = 0; i < size; i++)

        elements[i] = v.elements[i];

}

template<typename T> TVector<T>::~TVector()

{

    delete[] elements;

}

template<typename T> TVector<T> TVector<T>::operator+(const TVector<T>& v) {

    if (size != v.size)

        throw "different lengths on addition";

    else if (start\_index != v.start\_index)

        throw "different indexes on addition";

    TVector vectTmp(size, start\_index);

        for (int i = 0; i < size; i++)

            vectTmp[i] = elements[i] + v.elements[i];

        return vectTmp;

}

template<typename T> TVector<T> TVector<T>::operator+(const T& e) {

    TVector vectTmp(size, start\_index);

        for (int i = 0; i < size; i++)

            vectTmp[i] = elements[i] + e;

        return vectTmp;

}

template<typename T> TVector<T> TVector<T>::operator-(const TVector<T>& v) {

    if (size != v.size)

        throw "different lengths on subtraction";

    else if (start\_index != v.start\_index)

        throw "different indexes on subtraction";

    TVector vectTmp(size, start\_index);

        for (int i = 0; i < size; i++)

            vectTmp[i] = elements[i] - v.elements[i];

    return vectTmp;

}

template<typename T> TVector<T> TVector<T>::operator-(const T& e) {

    TVector vectTmp(size, start\_index);

        for (int i = 0; i < size; i++)

            vectTmp[i] = elements[i] - e;

        return vectTmp;

}

template<typename T> T TVector<T>::operator\*(const TVector<T>& v) {

    if (size != v.size)

        throw "different lengths on multiplication";

    else if (start\_index != v.start\_index)

        throw "different indexes on multiplication";

    int tmp = 0;

    for (int i = 0; i < size; i++)

        tmp += elements[i] \* v.elements[i];

    return tmp;

}

template<typename T> TVector<T> TVector<T>::operator\*(const T& e) {

    TVector vectTmp(size, start\_index);

        for (int i = 0; i < size; i++)

            vectTmp[i] = elements[i] \* e;

        return vectTmp;

}

template<typename T> bool TVector<T>::operator==(const TVector<T>& v) const {

    if ((start\_index != v.start\_index) || (size != v.size))

        return false;

    for (int i = 0; i < size; i++) {

        if (elements[i] != v.elements[i])

            return false;

    }

    return true;

}

template<typename T> bool TVector<T>::operator!=(const TVector<T>& v) const {

    return (!(\*this == v));

}

template<typename T> const TVector<T>& TVector<T>::operator=(const TVector<T>& v) {

    if (this == &v)

        return \*this;

    if (size != v.size) {

        delete[] elements;

        size = v.size;

        elements = new T[size];

    }

    start\_index = v.start\_index;

    for (int i = 0; i < size; i++)

        elements[i] = v.elements[i];

    return \*this;

}

template<typename T>  T& TVector<T>::operator[](int ind) {

    if ((ind > size) || (ind < 0))

        throw "out of range";

    return elements[ind];

}

### Реализация полей класса TMatrix:

template  <typename T> TMatrix<T>::TMatrix(int n) :TVector <TVector<T>>(n) {

    for (int i = 0; i < n; i++) {

        elements[i] = TVector<T>(n - i, i);

    }

}

template  <typename T> TMatrix<T>::TMatrix(const TMatrix& m) : TVector<TVector<T>>(m) {}

template  <typename T> TMatrix<T>::TMatrix(const TVector<TVector<T>>& v) : TVector<TVector<T>>(v) {}

template  <typename T> TMatrix<T> TMatrix<T>::operator+(const TMatrix<T>& m) {

    if (size != m.size)

        throw "got different sizes on addition, can't add, subtract or multiplicate";

    return TVector<TVector<T>>::operator+(m);

}

template  <typename T> TMatrix<T> TMatrix<T>::operator-(const TMatrix<T>& m) {

    if (size != m.size)

        throw "got different sizes on subtractoin, can't add, subtract or multiplicate";

    return TVector<TVector<T>>::operator-(m);

}

template  <typename T> TMatrix<T> TMatrix<T>::operator\*(const TMatrix<T>& m) {

    if (size != m.size)

        throw "got different sizes on multiplication, can't add, subtract or multiplicate";

    TMatrix<T> res(size);

    for (int i = 0; i < size; i++)

        for (int j = i; j < size; j++)

        {

            T sum = 0;

            for (int g = i; g <= j; g++)

                sum += elements[i][g - i] \* m.elements[g][j - g];

            res.elements[i][j - i] = sum;

        }

    return res;

}

template  <typename T> bool TMatrix<T>::operator==(const TMatrix& m) const {

    if (m.size != size)

        return false;

    return TVector<TVector<T>>::operator==(m);

}

template  <typename T> bool TMatrix<T>::operator!=(const TMatrix& m) const {

    return (!(\*this == m));

}

template  <typename T> const TMatrix<T>& TMatrix<T>::operator=(const TMatrix<T>& m)

{

    return  TVector<TVector<T>>::operator=(m);

}