

Министерство образования и науки Российской Федерации

**Федеральное государственное автономное образовательное  
учреждение высшего образования  
«КАЗАНСКИЙ (ПРИВОЛЖСКИЙ) ФЕДЕРАЛЬНЫЙ  
УНИВЕРСИТЕТ»**

Институт вычислительной математики и информационных технологий  
Специальность: 01.03.04 – Прикладная математика

Отчет по архитектуре компьютеров и программированию на C++

**ШАБЛОННЫЕ КЛАССЫ ВЕКТОР И МАТРИЦА**

**Выполнил**

студент группы 09-521

Талипов Р. Ф.

**Проверил**

доцент кафедры

прикладной математики

Тумаков Д. Н.

**Казань 2018**

# Содержание

1	Постановка задачи	3
2	Теоретические сведения о шаблонных классах	3
3	Описание и примеры работы с классом вектор	4
4	Описание и примеры работы с классом матрица	15
5	Возникшие проблемы при перегрузке оператора * в классах	31
6	Листинг TVector.h	33
7	Листинг TMatrix.h	39

# 1 Постановка задачи

Необходимо реализовать шаблонные классы вектор и матрица, для того чтобы использовать матрицы и векторы разных типов, не переписывая аналогичный код. Необходимо перегрузить различные операции для работы с векторами и матрицами.

## 2 Теоретические сведения о шаблонных классах

При построении шаблона определение класса меняется на определение шаблона, а функции – члены на функции – члены шаблона. Необходимо предварить шаблонный класс следующим кодом:

```
template <class Type>
```

Ключевое слово *template* говорит компилятору, что нужно определить шаблон. Часть кода в угловых скобках аналогична списку аргументов в функции. Можно считать, что ключевое слово *class* служит именем типа для переменной, которая получает тип как значение, а *Type* является именем этой переменной. Использование *class* не означает, что *Type* должен быть классом; это означает только, что *Type* служит в качестве общего спецификатора типа, который будет заменен реальным типом при использовании шаблона. Последние реализации C++ позволяют применять вместо *class* более точное ключевое слово *typename*:

```
template <typename Type>
```

Можно указать общее имя типа вместо *Type*; правила наименования здесь те же, что и для любого другого идентификатора. Обычно используют *T* и *Type*; в данном. При вызове шаблона *Type* заменяется реальным типом *int* или *double*.

Каждая функция шаблонного класса должна предваряться аналогичным объявлением шаблона:

```
template <typename Type> ,
```

или

```
template <class Type>
```

### 3 Описание и примеры работы с классом вектор

В заголовочном файле "TVector.h" объявляем класс шаблонным,

```
template <typename Type>
class TVector;
```

Полями данного класса будут размер вектора и указатель на тип.

```
class TVector
{
private:
    Type * ptr;
    int size;

    ...
}
```

Приведем конструкторы и деструктор данного класса:

```
TVector()
{
    size = 0;
    ptr = nullptr;
}

TVector(int size_)
{
    size = size_;
    ptr = new Type[size];
    for (int i = 0; i != size; i++)
    {
        ptr[i] = 1.0;
    }
}

TVector (const TVector<Type>& other)
{
    size = other.size;
    ptr = new Type[size];
    for (size_t i = 0; i != size; ++i)
        ptr[i] = other.ptr[i];
}
```

```

~TVector()
{
    delete[] ptr;
    ptr = nullptr;
}

```

Первый конструктор – конструктор по умолчанию, задает размер вектора нулем, а указателю присваивает нулевой указатель. Второй конструктор в качестве параметра принимает значение типа `int`, которое будет присвоено полю `size`, с этим размером будет выделена память под тип `Type`, далее в цикле массив заполняется единицами.

Например, объект класса `TVector` специализированный под тип `complex<double>`, то есть это массив типа `complex<double>` размера 5, заполненный единицами:

```
TVector<complex<double>> A(5);
```

Третий – конструктор копирования, который служит для инициализации одного объекта другим. Инициализация возникает в трех случаях: когда один объект инициализирует другой, когда копия объекта передается в функцию и когда создается временный объект (обычно если он служит возвращаемым значением). Например, любая из следующих инструкций вызывает инициализацию:

```

myclass x = y; // initialization
func (x); // as parametr
y = func (); // get temp object

```

В деструкторе освобождается память и `ptr` присваивается значение `nullptr`.

Также есть функции вывода вектора на консоль и возврат размера вектора:

```

int get_size()
{
    return size;
}

void show()
{
    for (int i = 0; i != size; i++)
    {
        cout << ptr[i] << "    ";
    }
    cout << '\n';
}

```

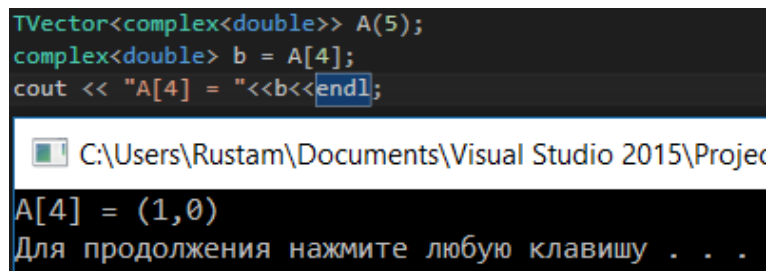
Далее приведем реализации перегруженных операций. Перегрузка - это возможность поддерживать несколько функций с одним названием, но разными сигнатурами вызова.

- Перегрузка оператора взятия элемента вектора по индексу:

```
Type & operator[](int num)
{
    try {
        if ((num >= 0) && (num < size))
            return ptr[num];
        else
            throw "size error";
    }
    catch (char *str)
    {
        cout << str << endl;
    }
}
```

В блоке try происходит попытка взять элемент на позиции num, если  $\text{num} \in [0, \text{size})$ , то попытка будет успешной, иначе будет выведена на консоль ошибка "size error".

Пример работы:



```
TVector<complex<double>> A(5);
complex<double> b = A[4];
cout << "A[4] = " << b << endl;
```

C:\Users\Rustam\Documents\Visual Studio 2015\Projec

A[4] = (1,0)  
Для продолжения нажмите любую клавишу . . .

Рис. 1: Пример использования оператора [ ]

- Перегрузка оператора ==

```
template<typename Type>
bool operator==(TVector<Type>& other)
{
    if (this->size != other.size)
        return false;
    else
```

```

        for (int i = 0; i != this->size; i++)
        {
            if (this->ptr[i] != other.ptr[i])
            {
                return false;
            }
        }
        return true;
    }

```

Так как мы перегружаем оператор шаблонного класса, перед реализацией перегрузки написать `template<typename Type>`. В качестве параметра функция получает вектор такого же типа. То есть, если исходный вектор специализирован типом `double`, то и сравнивать необходимо с вектором типа `double`. В функции сперва сравниваются поля `size` двух объектов, а далее ведется поэлементное сравнение.

- Перегрузка оператора `!=`

```

template<typename Type>
bool operator!=(const TVector<Type>& other)
{
    if (this->size != other.size)
        return true;
    else
        for (int i = 0; i != this->size; i++)
        {
            if (this->ptr[i] != other.ptr[i])
            {
                return true;
            }
        }
        return false;
    }
}

```

Здесь все тоже самое, что и в перегрузке оператора `=`, разве что функция вернет `true`, если не равны размеры векторов или хотя бы один элемент.

Для сравнения объекта класса с `NULL` есть следующая перегрузка `!=`

```

bool operator!=(int)
{
    if (this == NULL)

```

```

{
    return true;
}
else
    return false;
}

```

- Перегрузка оператора присваивания =

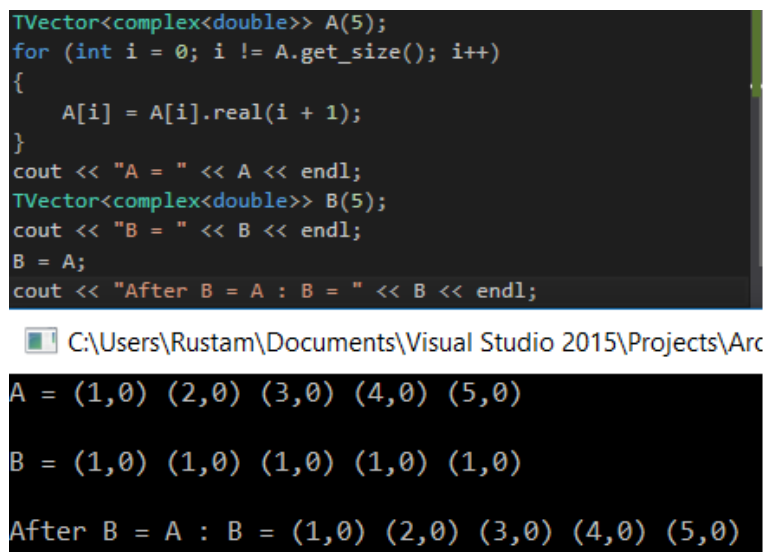
```

template <typename Type>
TVector<Type>& operator=(TVector<Type>& other)
{
    if (this != &other)
    {
        delete[] ptr;
        this->size = other.size;
        this->ptr = new Type[this->size];
        for (int i = 0; i != this->size; i++)
            this->ptr[i] = other.ptr[i];
    }
    return (*this);
}

```

Данный оператор работает в связке с конструктором копирования, то есть прежде чем присвоить одному объекту другой, необходимо создать копию присваиваемого объекта. Конструктор копирования был описан выше.

Продemonстрируем работу перегруженного оператора:



```

TVector<complex<double>> A(5);
for (int i = 0; i != A.get_size(); i++)
{
    A[i] = A[i].real(i + 1);
}
cout << "A = " << A << endl;
TVector<complex<double>> B(5);
cout << "B = " << B << endl;
B = A;
cout << "After B = A : B = " << B << endl;

```

C:\Users\Rustam\Documents\Visual Studio 2015\Projects\Arc

```

A = (1,0) (2,0) (3,0) (4,0) (5,0)
B = (1,0) (1,0) (1,0) (1,0) (1,0)
After B = A : B = (1,0) (2,0) (3,0) (4,0) (5,0)

```

Рис. 2: Пример использования оператора =

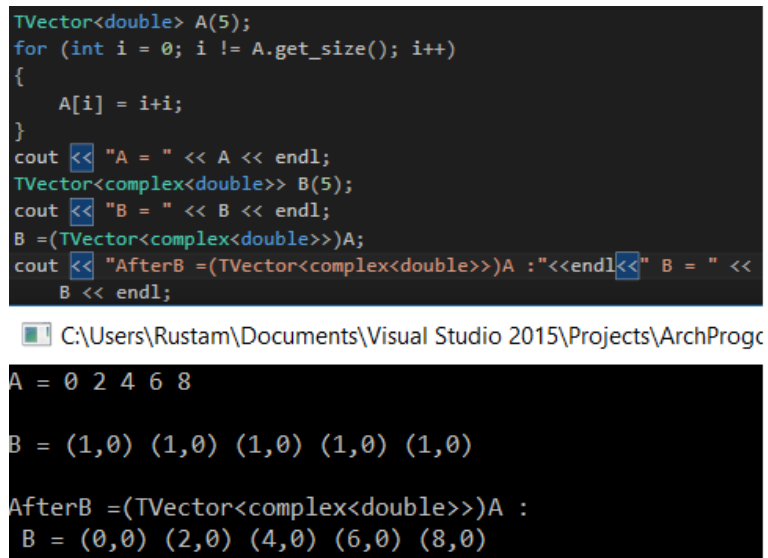


- Оператор приведения типа ()

```
template<typename U> operator TVector<U>()
{
    TVector<U> temp(size);
    for (int i = 0; i != size; i++)
    {
        temp[i]=static_cast<U>(ptr[i]);
    }
    return (temp);
}
```

В функции создается временный объект класса типа U, его элементам присваиваются элементы из this, с помощью преобразования static\_cast к типу U.

Пример:



```
TVector<double> A(5);
for (int i = 0; i != A.get_size(); i++)
{
    A[i] = i+i;
}
cout << "A = " << A << endl;
TVector<complex<double>> B(5);
cout << "B = " << B << endl;
B =(TVector<complex<double>>)A;
cout << "AfterB =(TVector<complex<double>>)A : "<<endl<<" B = " <<
B << endl;
```

C:\Users\Rustam\Documents\Visual Studio 2015\Projects\ArchProgc

```
A = 0 2 4 6 8
B = (1,0) (1,0) (1,0) (1,0) (1,0)
AfterB =(TVector<complex<double>>)A :
B = (0,0) (2,0) (4,0) (6,0) (8,0)
```

Рис. 3: Пример приведения типа TVector<double> к TVector<complex<double>>

- Перегрузка оператора умножения \*

```
template<typename Type>
TVector<Type> operator*(Type b)
{
    for (int i = 0; i != size; i++)
    {
        ptr[i] *= b;
    }
}
```

```

        return (*this);
    }

```

В качестве входного параметра функция принимает скаляр *b* такого же типа, что и элементы самого вектора, далее ведется поэлементное умножение.

Приведем пример умножения вектора на скаляр, скаляр и элементы вектора одного типа:

```

TVector<complex<double>> A(5);
for (int i = 0; i != A.get_size(); i++)
{
    A[i] = complex<double>(i+i,i+i);
}
cout << "A = " << A << endl;
A = A * complex<double>(5.0,0);
cout << "After A * complex<double>(5,0):" << endl << " A = " << A << endl;

```

C:\Users\Rustam\Documents\Visual Studio 2015\Proje

A = (0,0) (2,2) (4,4) (6,6) (8,8)

After A \* complex<double>(5,0):  
A = (0,0) (10,10) (20,20) (30,30) (40,40)

```

TVector<complex<double>> A(5);
int b = 5;
for (int i = 0; i != A.get_size(); i++)
{
    A[i] = complex<double>(i+i,i+i);
}
cout << "A = " << A << endl;
A = A * complex<double>(b);
cout << "After A * b" << endl << " A = " << A << endl;

```

C:\Users\Rustam\Documents\Visual Studio 2015\Proje

A = (0,0) (2,2) (4,4) (6,6) (8,8)

After A \* b  
A = (0,0) (10,10) (20,20) (30,30) (40,40)

Рис. 4: Пример умножения вектора, элементы которого комплексные, на комплексный скаляр и на целочисленный скаляр справа

Для умножения вектора на скаляр, тип которого не совпадает с типом элементов вектора, сперва необходимо привести к необходимому типу, рис.4 (справа)

О проблеме перегрузки умножения вектора на скаляр слева будет описано в пункте 5

Для того, чтобы складывать, вычитать, умножать векторы, а так же вводить с консоли и выводить их консоль используются дружественные функции–операторы. Так как дружественные функции не являются членами класса, они не могут иметь неявный аргумент *this*. Поэтому при использовании дружественной функции-оператора оба операнда передаются функции при перегрузке бинарных операторов, а при перегрузке унарных операторов передается один операнд. Объявляя функцию другом класса, мы позволяем ей иметь те же привилегии доступа к полям класса, что имеют функции-члены класса.

Внутри класса TVector необходимо лишь объявить дружественные функции – операторы, а реализовать вне класса.

- Дружественная функция–оператор сложения векторов +  
Объявление в классе:

```
template <typename Type, typename Type_other>
friend TVector<std::common_type_t<Type, Type_other>>
    operator+ (TVector<Type> & a, TVector<Type_other> & b);
```

`std::common_type_t` определяет общий тип между всеми его аргументами шаблона. То есть, если в его аргументах будут `int` и `complex<double>`, то общий тип будет `complex<double>`.

Реализация вне класса:

```
template<typename Type, typename Type_other>
TVector<std::common_type_t<Type, Type_other>> operator +
(TVector<Type> &a, TVector<Type_other> &b)
{
    TVector<std::common_type_t<Type, Type_other>> res(a.size);
    if (a.size != b.size)
    {
        exit(1);
    }
    else
    {
        for (int i = 0; i != a.size; i++)
        {
            res[i] = a[i] + b[i];
        }
        return res;
    }
}
```

Здесь сперва создается временный вектор `res`, тип которого определяется с помощью *`common_type_t`*, далее ведется поэлементное сложение векторов, в случае, если размеры векторов не равны, то программа завершит свою работу, иначе, функция вернет сумму векторов.

- Дружественная функция–оператор вычитания векторов -

Объявление в классе:

```
template <typename Type, typename Type_other>
friend TVector<std::common_type_t<Type, Type_other>>
    operator- (TVector<Type> & a, TVector<Type_other> & b);
```

Реализация вне класса:

```
template<typename Type, typename Type_other>
TVector<std::common_type_t<Type, Type_other>> operator -
(TVector<Type> &a, TVector<Type_other> &b)
{
    TVector<std::common_type_t<Type, Type_other>> res(a.size);
    if (a.size != b.size)
    {
        exit(1);
    }
    else
    {
        for (int i = 0; i != a.size; i++)
        {
            res[i] = a[i] - b[i];
        }
        return res;
    }
}
```

Здесь сперва создается временный вектор `res`, тип которого определяется с помощью *common\_type\_t*, далее ведется поэлементное вычитание векторов, в случае, если размеры векторов не равны, то программа завершит свою работу, иначе, функция вернет разность векторов.

- Дружественная функция–оператор умножения векторов \*

Объявление в классе:

```
template <typename Type, typename Type_other>
friend TVector<std::common_type_t<Type, Type_other>>
    operator* (TVector<Type> & a, TVector<Type_other> & b);
```

Реализация вне класса:

```
template<typename Type, typename Type_other>
TVector<std::common_type_t<Type, Type_other>> operator *
(TVector<Type> &a, TVector<Type_other> &b)
{
    TVector<std::common_type_t<Type, Type_other>> res(a.size);
    if (a.size != b.size)
```

```

        {
            exit(1);
        }
        else
        {
            for (int i = 0; i != a.size; i++)
            {
                res[i] = a[i] * b[i];
            }
            return res;
        }
    }
}

```

Здесь сперва создается временный вектор `res`, тип которого определяется с помощью `common_type_t`, далее ведется поэлементное умножение векторов, в случае, если размеры векторов не равны, то программа завершит свою работу, иначе, функция вернет результат поэлементного умножения.

Приведем результаты сложения, вычитания и умножения векторов, элементы которых `complex<double>` и `double`.

The image contains two side-by-side screenshots of a Visual Studio 2015 IDE. Each screenshot shows a C++ source file and its corresponding output window.

**Left Screenshot:** The source code defines a vector `A` of type `complex<double>` with 5 elements, each being  $i + i^2$  (where  $i$  is the imaginary unit). It then defines a vector `B` of type `double` with 5 elements, all equal to 1. The code performs the operation `A = A + B`. The output window shows:
   
A = (0,0) (2,2) (4,4) (6,6) (8,8)
   
B = 1 1 1 1 1
   
After A + B
   
A = (1,0) (3,2) (5,4) (7,6) (9,8)

**Right Screenshot:** The source code is identical to the left one, but it performs the operation `A = A - B` instead. The output window shows:
   
A = (0,0) (2,2) (4,4) (6,6) (8,8)
   
B = 1 1 1 1 1
   
After A - B
   
A = (-1,0) (1,2) (3,4) (5,6) (7,8)

Рис. 5: Пример сложения и вычитания умножения векторов

```

TVector<complex<double>> A(5);
for (int i = 0; i != A.get_size(); i++)
{
    A[i] = complex<double>(i+i,i+i);
}
cout << "A = " << A << endl;
TVector<double> B(5);
for (int i = 0; i != B.get_size(); i++)
{
    B[i] += i;
}
cout << "B = " << B << endl;
A = A * B;
cout << "After A * B"<<endl<<" A = " <<
    A << endl;

```

C:\Users\Rustam\Documents\Visual Studio 2015\Proje

```

A = (0,0) (2,2) (4,4) (6,6) (8,8)
B = 1 2 3 4 5
After A * B
A = (0,0) (4,4) (12,12) (24,24) (40,40)

```

Рис. 6: Пример поэлементного умножения векторов

- Дружественная функция–оператор ввода вектора > >

Объявление в классе:

```

friend istream & operator >>
    <Type> (istream & input, TVector<Type> & obj);

```

Реализация вне класса:

```

template<typename Type>
istream & operator >> (istream & input, TVector<Type> & obj)
{
    for (int i = 0; i < obj.size; i++)
        input >> obj.ptr[i];
    return input;
}

```

- Дружественная функция–оператор вывода вектора < <

Объявление в классе:

```

friend ostream & ::operator<<
    <Type> (ostream & output, const TVector<Type>& object);

```

Реализация вне класса:

```
template<typename Type>
ostream & operator<< (ostream & output,
    const TVector<Type>& object)
{
    for (int i = 0; i != object.size; i++)
    {
        output << object.ptr[i] << " ";
    }
    output << endl;
    return output;
}
```

Так же используя оператор typedef определены следующие псевдонимы для вектора из целочисленных значений, с двойной точностью и `complex<double>` :

```
typedef TVector<int> VectorInt;
typedef TVector<double> VectorDouble;
typedef TVector<complex<double>> VectorComplex;
```

Можно было определить и для `complex<int>`, но это бессмысленно, так как в вычислениях почти всегда возникают дробные значения.

То есть таким образом можно упростить написание кода в дальнейшем.

## 4 Описание и примеры работы с классом матрица

В заголовочном файле "TMatrix.h" объявляем класс шаблонным,

```
template<typename Type>
class TMatrix;
```

Полями данного класса будут количество строк и столбцов матрицы и указатель типа `TVector<Type>`:

```
class TMatrix
{
private:
    int row, col;
    TVector<Type> * vptr;
```

```
...  
}
```

Приведем конструкторы и деструктор данного класса:

```
TMatrix() {}  
TMatrix(int row, int col)  
{  
    this->row = row;  
    this->col = col;  
    vptr = new TVector<Type>[row];  
    for (int i = 0; i != row; i++)  
    {  
        vptr[i] = TVector<Type>(col);  
    }  
    for (int i = 0; i != row; i++)  
    {  
        for (int j = 0; j != col; j++)  
            vptr[i][j] = 0.0;  
    }  
}  
TMatrix(int row) // identity  
{  
    this->col = this->row = row;  
    vptr = new TVector<Type>[row];  
    for (int i = 0; i != row; i++)  
    {  
        vptr[i] = TVector<Type>(row);  
    }  
    for (int i = 0; i != row; i++)  
    {  
        for (int j = 0; j != row; j++)  
            if (i == j)  
                vptr[i][j] = 1.0;  
            else vptr[i][j] = 0.0;  
    }  
}  
  
TMatrix(const TMatrix<Type> & other)  
{  
    this->row = other.row;  
    this->col = other.col;  
    this->vptr = new TVector<Type>[row];
```



```

        for (int i = 0; i != row; i++)
            this->vptr[i] = TVector<Type>(col);
        for (int i = 0; i != row; i++)
            this->vptr[i] = other.vptr[i];
    }

    ~TMatrix()
    {
        delete[] vptr;
        vptr = nullptr;
    }

```

Первый конструктор – конструктор по умолчанию, второй – присваиваются полям значения и выделяется память, в массиве vptr размера row хранятся векторы типа Type размера col, таким образом получается матрица размера row x col. Заполняются нулями элементы матрицы. Третий конструктор служит для создания единичной матрицы. Четвертый – конструктор копирования.

В деструкторе освобождается память и указателю vptr присваивается nullptr.

Также есть следующие функции:

```

int get_row()
{
    return row;
}

int get_col()
{
    return col;
}

Type determinant()
{
    if (this->col != this->row)
    {
        exit(1);
    }
    TMatrix<Type> B = *this;
    // getting to upper triangular
    for (int step = 0; step != row; step++)
    {
        for (int r = step + 1; r != row; r++)
        {

```

```

        if (B[step][step] != 0.0)
        {
            Type coef = -B[r][step] / B[step][step];
            for (int c = step; c != row; c++)
                B[r][c] += B[step][c] * coef;
        }
        else
        {
            cerr << "singular matrix" << endl;
            system("pause");
            exit(1);
        }
    }
}
Type det = 1.0;
for (int i = 0; i != row; i++)
{
    det *= B[i][i];
}
return det;
}

void show()
{
    for (int i = 0; i != row; i++)
    {
        for (int j = 0; j != col; j++)
        {
            cout << vptr[i][j] << " ";
        }
        cout << endl;
    }
}

template<typename Type>
TMatrix<Type> & inverse(TMatrix<Type> & other)
{
    TMatrix<Type> * inver =
        new TMatrix<Type>(other.get_row(), other.get_col());
    TMatrix<Type> temp = other;
    int ki, kj;
    for (int indexi = 0; indexi != temp.row; indexi++)
    {

```

```

for (int indexj = 0; indexj != temp.col; indexj++)
{
    ki = indexi; kj = indexj;
    TMatrix<Type> adj(temp.row - 1, temp.col - 1);
    int si = 0, sj = 0;
    for (int i = 0; i != temp.row; i++)
    {
        if (i != ki)
        {
            sj = 0;
            for (int j = 0; j != temp.col; j++)
            {
                if (j != kj)
                {
                    adj[si][sj] = temp[i][j];
                    sj++;
                }
            }
            si++;
        }
    }
    cout << "A" << indexi+1 << indexj+1
         << " = " <<endl<< adj<<endl;
    (*inver)[indexi][indexj] = adj.determinant()
        *pow(-1.0, indexi + indexj);
}
}
for (int indexi = 0; indexi != temp.row-1; indexi++)
{
    for (int indexj = indexi+1; indexj != temp.col; indexj++)
    {
        swap((*inver)[indexj][indexi],
            (*inver)[indexi][indexj]);
    }
}
Type det = temp.determinant();
for (int indexi = 0; indexi != temp.row; indexi++)
{
    for (int indexj = 0; indexj != temp.col; indexj++)
    {
        (*inver)[indexi][indexj] = (*inver)[indexi][indexj]
            * 1.0 / det;
    }
}

```

```

    }
    return (*inver);
}

```

В первых двух функциях возвращаются значения полей класса `row` и `col` соответственно.

В функции `determinant()` сперва проверяется квадратная ли матрица, если нет, то выход из приложения. Если квадратная, то матрица приводится к верхне-треугольному виду с помощью преобразований Гаусса, так же есть обработка на случай вырожденности матрицы – сообщение о вырожденности и выход из приложения. После приведения к верхне-треугольному виду перемножаются элементы на главной диагонали и результат записывается в `det`, типа `Type`, то есть такого же типа как и элементы самой матрицы.

The image shows two side-by-side screenshots from Visual Studio 2015. The left screenshot displays C++ code for a complex matrix `A` of type `TMatrix<complex<double>>` (5x5). The code initializes the matrix with values `complex<double>(j+1, 5+i)` and prints the matrix and its determinant. The right screenshot shows the same code but for a real matrix `A` of type `TMatrix<double>`, where elements are assigned using `rand() % 10`. Below the code, the console output is shown for both cases. For the complex matrix, the determinant is a complex number with a very small imaginary part. For the real matrix, the determinant is an integer.

```

// Left Screenshot: Complex Matrix
TMatrix<complex<double>> A(5, 5);
for (int i = 0; i != 5; i++)
{
    for (int j = 0; j != 5; j++)
    {
        A[i][j] = complex<double>(j+1, 5+i);
    }
}
cout << "A = " << endl << A << endl;
cout << "det(A) = " << A.determinant() << endl;

// Right Screenshot: Real Matrix
TMatrix<double> A(5, 5);
for (int i = 0; i != 5; i++)
{
    for (int j = 0; j != 5; j++)
    {
        A[i][j] = rand() % 10;
    }
}
cout << "A = " << endl << A << endl;
cout << "det(A) = " << A.determinant() << endl;

```

**Console Output (Left - Complex Matrix):**

```

A =
(1,5) (2,5) (3,5) (4,5) (5,5)
(1,6) (2,6) (3,6) (4,6) (5,6)
(1,7) (2,7) (3,7) (4,7) (5,7)
(1,8) (2,8) (3,8) (4,8) (5,8)
(1,9) (2,9) (3,9) (4,9) (5,9)

det(A) = (3.81115e-45, -1.48984e-44)

```

**Console Output (Right - Real Matrix):**

```

A =
1 7 4 0 9
4 8 8 2 4
5 5 1 7 1
1 5 2 7 6
1 4 2 3 2

det(A) = -2122

```

Рис. 7: Пример вычисления детерминанта для комплексной матрицы и вещественной

Функция `show()` отвечает за вывод матрицы на консоль.

В функции `inverse` ведется вычисление обратной матрицы. На входе ссылка на матрицу типа `Type`, на выходе ссылка на матрицу типа `Type`. Матрица вычисляется с помощью алгебраических дополнений. Алгоритм нахождения обратной матрицы с помощью алгебраических дополнений следующий:

- Найти детерминант матрицы  $A$ . Если  $\det(A) \neq 0$ , то обратная матрица существует, иначе обратная матрица не существует.
- Найти матрицу миноров  $M$ . Из матрицы  $M$  найти матрицу алгебраических дополнений  $C^*$ .

- Транспонировать матрицу  $C^* = C^{*T}$ ,
- По формуле найти обратную матрицу

$$A^{-1} = \frac{C^{*T}}{\det(A)}$$

Далее приведем реализации перегруженных операций.

- Перегрузка оператора взятия элемента по индексу [ ]

```

TVector<Type> & operator [] (int num)
{
    try {
        if ((num >= 0) && (num < row))
            return vptr[num];
        else
            throw "size error";
    }
    catch (char *str)
    {
        cout << str << endl;
    }
}

```

Когда мы хотим обратиться к элементу в матрице таким образом:  $A[num][num]$  сперва срабатывает перегруженная функция–оператор из класса `TMatrix` и мы получаем доступ к необходимому вектору по номеру `num`, а затем срабатывает аналогичный оператор из класса `TVector` и мы получаем нужный элемент в позиции `num`.

- Перегрузка оператора приведения типа ( )

```

template<typename U>
operator TMatrix<U>()
{
    TMatrix<U> temp(row, col);
    for (int i = 0; i != row; i++)
    {
        for (int j = 0; j != col; j++)
        {
            temp[i][j] = static_cast<U>(this->vptr[i][j]);
        }
    }
}

```

```

    }
    return temp;
}

```

Принцип работы данного оператора аналогичен работе оператора приведения типа в классе TVector.

- Перегрузка оператора умножения матрицы на число \*

```

template<typename Type>
TMatrix<Type> operator*(Type b)
{
    for (int i = 0; i != row; i++)
    {
        for(int j = 0 ; j!=col;j++)
            vptr[i][j] *= b;
    }
    return (*this);
}

```

Здесь ведется поэлементное умножение матрицы на число справа. Тип элементов матрицы и числа должно совпадать. О проблеме перегрузке умножения матрицы одного типа на число другого типа будет описано в пункте 5

- Перегрузка оператора присваивания =

```

TMatrix<Type>& operator= (TMatrix<Type> & other)
{
    if (this != &other)
    {
        delete[] vptr;
        col = other.col;
        row = other.row;
        vptr = new TVector<Type>[row];
        for (int i = 0; i != row; i++)
            vptr[i] = TVector<Type>(col);
        for (int i = 0; i != row; i++)
            for (int j = 0; j != col; j++)
                vptr[i][j] = other.vptr[i][j];
    }
    return (*this);
}

```

Работает в связке с конструктором копирования. Типы двух матриц должны совпадать, иначе необходимо сперва применить операцию приведения типа, рис.8 (справа). Сперва в матрице, которая вызвала данный оператор, освобождается память, присваиваются значения полей и матрицы other, заново выделяется память и заполняется элементами из матрицы other. Функция – оператор возвращает ссылку на матрицу.

```

// Left Screenshot Code
#include <iostream>
using namespace std;

template <typename T>
class TMatrix {
public:
    TMatrix(int n) {
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                A[i][j] = rand() % 10;
            }
        }
    }
    TMatrix<T> operator=(const TMatrix<T> &other) {
        // ... (omitted for brevity)
    }
    friend ostream &operator<<(ostream &os, const TMatrix<T> &m) {
        // ... (omitted for brevity)
    }
private:
    T **A;
};

int main() {
    TMatrix<double> A(5, 5);
    for (int i = 0; i < 5; i++) {
        for (int j = 0; j < 5; j++) {
            A[i][j] = rand() % 10;
        }
    }
    cout << "A = " << endl << A << endl;
    TMatrix<double> B(5);
    cout << "B = " << endl << B << endl;
    B = A;
    cout << "after B = A: B = " << endl << B << endl;
    system("pause");
    return 0;
}

```

```

// Right Screenshot Code
#include <iostream>
using namespace std;

template <typename T>
class TMatrix {
public:
    TMatrix(int n) {
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                A[i][j] = rand() % 10;
            }
        }
    }
    TMatrix<T> operator=(const TMatrix<T> &other) {
        // ... (omitted for brevity)
    }
    friend ostream &operator<<(ostream &os, const TMatrix<T> &m) {
        // ... (omitted for brevity)
    }
private:
    T **A;
};

int main() {
    TMatrix<double> A(5, 5);
    for (int i = 0; i < 5; i++) {
        for (int j = 0; j < 5; j++) {
            A[i][j] = rand() % 10;
        }
    }
    cout << "A = " << endl << A << endl;
    TMatrix<complex<double>> B(5);
    cout << "B = " << endl << B << endl;
    B = (TMatrix<complex<double>>)A;
    cout << "after B = A: B = " << endl << B << endl;
    system("pause");
    return 0;
}

```

Рис. 8: Пример работы оператора присваивания для матриц, элементы которых одинакового типа (слева) и разного типа (справа)

Для того, чтобы складывать, вычитать, умножать матрицы, умножать матрицу на вектор, находить обратную матрицу, решать СЛАУ, а так же вводить с консоли и выводить их консоль используются дружественные функции–операторы.

Аналогично классу TVector, внутри класса TMatrix необходимо лишь объявить дружественные функции – операторы, а реализовать вне класса.

- Дружественная функция–оператор сложения матриц +

Объявление в классе:

```

template <typename Type, typename Type_other>
friend TMatrix <std::common_type_t<Type, Type_other>>
    operator +(TMatrix<Type> a, TMatrix<Type_other> b);

```

Как можно заметить, можно складывать матрицы, элементы которых разного типа.

Реализация вне класса:

```

template<typename Type, typename Type_other>
TMatrix<std::common_type_t<Type, Type_other>>
operator+(TMatrix<Type> a,    TMatrix<Type_other> b)
{
    TMatrix<std::common_type_t<Type, Type_other>>
    res(a.row, a.col);
    if ((a.row != b.row) || (a.col != b.col))
    {
        exit(1);
    }
    else
    {
        for (int i = 0; i != a.row; i++)
        {
            for (int j = 0; j != a.col; j++)
                res[i][j] = a[i][j] + b[i][j];
        }
        return res;
    }
}

```

Здесь сперва создается временная матрица *res*, тип которой определяется с помощью *common\_type\_t*, далее ведется поэлементное сложение матриц, в случае, если размеры матриц не равны, то программа завершит свою работу, иначе, функция вернет сумму матриц.

- Дружественная функция–оператор вычитания матриц -

Объявление в классе:

```

template <typename Type, typename Type_other>
friend TMatrix <std::common_type_t<Type, Type_other>>
    operator -(TMatrix<Type> a, TMatrix<Type_other> b);

```

Реализация вне класса:

```

template<typename Type, typename Type_other>
TMatrix<std::common_type_t<Type, Type_other>> operator-(TMatrix
{
    TMatrix<std::common_type_t<Type, Type_other> > res(a.row, a
    if ((a.row != b.row) || (a.col != b.col))
    {

```



```

        exit(1);
    }
    else
    {
        for (int i = 0; i != a.row; i++)
        {
            for (int j = 0; j != a.col; j++)
                res[i][j] = a[i][j] - b[i][j];
        }
        return res;
    }
}

```

Здесь сперва создается временная матрица `res`, тип которой определяется с помощью *common\_type\_t*, далее ведется поэлементное вычитание матриц, в случае, если размеры матриц не равны, то программа завершит свою работу, иначе, функция вернет разность матриц.

- Дружественная функция–оператор умножения матриц \*

Объявление в классе:

```

template <typename Type, typename Type_other>
friend TMatrix <std::common_type_t<Type, Type_other>>
    operator *(TMatrix<Type> a, TMatrix<Type_other> b);

```

Реализация вне класса:

```

template<typename Type, typename Type_other>
TMatrix<std::common_type_t<Type, Type_other>>
operator*(TMatrix<Type> a, TMatrix<Type_other> b)
{
    TMatrix<std::common_type_t<Type, Type_other>>
        res(a.get_col(), b.get_row());
    if (a.get_col() != b.get_row())
    {
        exit(1);
    }
    else
    {
        for (int i = 0; i != a.row; i++)
        {

```

```

        for (int j = 0; j != b.col; j++)
        {
            res[i][j] = 0.0;
            for (int k = 0; k != a.col; k++)
                res[i][j] += a[i][k] * b[k][j];
        }
    }
    return res;
}
}

```

Здесь сперва создается временная матрица `res`, тип которой определяется с помощью `common_type_t`, далее ведется умножение векторов, в случае, если количество столбцов матрицы `a` не равно количеству строк матрицы `b`, то программа завершит свою работу, иначе, функция вернет результат матричного умножения.

Приведем результаты сложения, вычитания и умножения векторов, элементы которых `complex<double>` и `double`.

```

// ...
TMatrix<double> A(5, 5);
for (int i = 0; i != 5; i++)
{
    for (int j = 0; j != 5; j++)
    {
        A[i][j] = rand() % 10;
    }
}
cout << "A = " << endl << A << endl;
TMatrix<complex<double>> B(5);
cout << "B = " << endl << B << endl;
B = B + A;
cout << "after B = B + A: B = " << endl << B << endl;
system("pause");
return 0;

```

A =  
1 7 4 0 9  
4 8 8 2 4  
5 5 1 7 1  
1 5 2 7 6  
1 4 2 3 2

B =  
(1,0) (0,0) (0,0) (0,0) (0,0)  
(0,0) (1,0) (0,0) (0,0) (0,0)  
(0,0) (0,0) (1,0) (0,0) (0,0)  
(0,0) (0,0) (0,0) (1,0) (0,0)  
(0,0) (0,0) (0,0) (0,0) (1,0)

after B = B + A: B =  
(2,0) (7,0) (4,0) (0,0) (9,0)  
(4,0) (9,0) (8,0) (2,0) (4,0)  
(5,0) (5,0) (2,0) (7,0) (1,0)  
(1,0) (5,0) (2,0) (8,0) (6,0)  
(1,0) (4,0) (2,0) (3,0) (3,0)

```

// ...
TMatrix<double> A(5, 5);
for (int i = 0; i != 5; i++)
{
    for (int j = 0; j != 5; j++)
    {
        A[i][j] = rand() % 10;
    }
}
cout << "A = " << endl << A << endl;
TMatrix<complex<double>> B(5);
cout << "B = " << endl << B << endl;
B = B - A;
cout << "after B = B - A: B = " << endl << B << endl;
system("pause");
return 0;

```

A =  
1 7 4 0 9  
4 8 8 2 4  
5 5 1 7 1  
1 5 2 7 6  
1 4 2 3 2

B =  
(1,0) (0,0) (0,0) (0,0) (0,0)  
(0,0) (1,0) (0,0) (0,0) (0,0)  
(0,0) (0,0) (1,0) (0,0) (0,0)  
(0,0) (0,0) (0,0) (1,0) (0,0)  
(0,0) (0,0) (0,0) (0,0) (1,0)

after B = B - A: B =  
(0,0) (-7,0) (-4,0) (0,0) (-9,0)  
(-4,0) (-7,0) (-8,0) (-2,0) (-4,0)  
(-5,0) (-5,0) (0,0) (-7,0) (-1,0)  
(-1,0) (-5,0) (-2,0) (-6,0) (-6,0)  
(-1,0) (-4,0) (-2,0) (-3,0) (-1,0)

Рис. 9: Пример сложения и вычитания матриц

```

setlocale(LC_ALL, "Russian");
// ...
TMatrix<double> A(5, 5);
for (int i = 0; i != 5; i++)
{
    for (int j = 0; j != 5; j++)
    {
        A[i][j] = rand() % 10 ;
    }
}
cout << "A = " << endl << A << endl;
TMatrix<complex<double>> B(5);
cout << "B = " << endl << B << endl;
B = B+A;
cout << "after B = B + A: B = " << endl << B << endl;
system("pause");
return 0;

```

```

A =
1 7 4 0 9
4 8 8 2 4
5 5 1 7 1
1 5 2 7 6
1 4 2 3 2
B =
(1,0) (0,0) (0,0) (0,0) (0,0)
(0,0) (1,0) (0,0) (0,0) (0,0)
(0,0) (0,0) (1,0) (0,0) (0,0)
(0,0) (0,0) (0,0) (1,0) (0,0)
(0,0) (0,0) (0,0) (0,0) (1,0)
after B = B - A: B =
(2,0) (7,0) (4,0) (0,0) (9,0)
(4,0) (9,0) (8,0) (2,0) (4,0)
(5,0) (5,0) (2,0) (7,0) (1,0)
(1,0) (5,0) (2,0) (8,0) (6,0)
(1,0) (4,0) (2,0) (3,0) (3,0)

```

```

setlocale(LC_ALL, "Russian");
// ...
TMatrix<double> A(5, 5);
for (int i = 0; i != 5; i++)
{
    for (int j = 0; j != 5; j++)
    {
        A[i][j] = rand() % 10 ;
    }
}
cout << "A = " << endl << A << endl;
TVector<complex<double>> B(5);
for (int i = 0; i != 5; i++)
{
    B[i] = complex<double>(i + 1, i + 1);
}
cout << "B = " << endl << B << endl;
B = B*A;
cout << "after B = B * A: B = " << endl << B << endl;
system("pause");
return 0;

```

```

A =
1 7 4 0 9
4 8 8 2 4
5 5 1 7 1
1 5 2 7 6
1 4 2 3 2
B =
(1,1) (2,2) (3,3) (4,4) (5,5)
after B = B * A: B =
(72,72) (72,72) (51,51) (75,75) (37,37)

```

Рис. 10: Пример умножения матриц и матрицы на вектор

- Дружественная функция–оператор умножения матрицы на вектор \*

Объявление в классе:

```

template <typename Type, typename Type_other>
friend TVector<std::common_type_t<Type, Type_other>>
    operator *(TMatrix<Type> a, TVector<Type_other> b);

```

Реализация вне класса:

```

template<typename Type, typename Type_other>
TVector<std::common_type_t<Type, Type_other>>
operator*(TMatrix<Type> a, TVector<Type_other> b)
{
    TVector<std::common_type_t<Type, Type_other>> res(a.row);
    if (a.col != b.get_size())
    {
        cerr << "wrong size";
        exit(1);
    }
    else
    {
        for (int i = 0; i != a.row; i++)
        {
            res[i] = 0;
            for (int j = 0; j != a.col; j++)
            {
                res[i] += a[i][j] * b[j];
            }
        }
    }
}

```

```

    }
    return res;
}

```

Здесь сперва создается временный вектор `res`, тип которой определяется с помощью `common_type_t`, далее ведется умножение матрицы вектор, в случае, если количество столбцов матрицы `a` не равно количеству элементов вектора `b`, то программа завершит свою работу, иначе, функция вернет вектор в качестве результата.

- Дружественная функция–оператор деления /

Объявление в классе:

```

template<typename Type>
friend TMatrix<Type> & operator / (double x,
TMatrix<Type> & other);

```

Реализация вне класса:

```

template <typename Type>
TMatrix<Type> & operator / (double x,
TMatrix<Type> & other)
{
    TMatrix<Type> *inv = new TMatrix<Type>(other.row,
other.col);
    *inv = inv->inverse(other);
    return *inv;
}

```

В качестве параметров поступает число и матрица, на выходе – матрица. Внутри функции вызывается функция обратной матрицы. Данная перегрузка необходима для того, чтобы найти обратную матрицу, написав следующее:  $B = 1/A$ .

- Дружественная функция–оператор деления /

Объявление в классе:

```

template <typename Type, typename U>
friend TVector<std::common_type_t<Type, U>> & operator /
(TVector<U> & b, TMatrix<Type> & other);

```

<pre> TMatrix&lt;double&gt; A(5, 5); for (int i = 0; i != 5; i++) {     for (int j = 0; j != 5; j++)     {         A[i][j] = rand() % 10 ;     } } A = 1.0 / A; cout &lt;&lt; "A = " &lt;&lt; endl &lt;&lt; A &lt;&lt; endl; cout &lt;&lt; "after A =1/A: A = " &lt;&lt; endl &lt;&lt; A &lt;&lt; endl; system("pause"); return 0; </pre>	<pre> A = 0.0772856 0.0777568 0.239397 -0.0188501 -0.566447 0.101791 -0.165881 0.0226202 -0.29312 0.741753 -0.157399 0.237983 -0.146089 0.184731 -0.248822 -0.118756 0.0268615 -0.0263902 0.175306 -0.0320452 0.0933082 0.0146089 0.0207352 0.147974 -0.403393  after A =1/A: A = 0.0772856 0.0777568 0.239397 -0.0188501 -0.566447 0.101791 -0.165881 0.0226202 -0.29312 0.741753 -0.157399 0.237983 -0.146089 0.184731 -0.248822 -0.118756 0.0268615 -0.0263902 0.175306 -0.0320452 0.0933082 0.0146089 0.0207352 0.147974 -0.403393 </pre>
---	---

Рис. 11: Пример вычисления обратной матрицы

Реализация вне класса:

```

template <typename Type, typename U>
TVector<std::common_type_t<Type, U>> & operator /
(TVector<U> & b, TMatrix<Type> & other)
{
    TMatrix<Type> inv(other.row, other.col);
    inv = inv.inverse(other);
    TVector<std::common_type_t<Type, U>> *res= new
    TVector<std::common_type_t<Type, U>>(other.row);
    *res = inv*b;
    return *res;
}

```

В качестве параметров поступает вектор и матрица, которые могут быть разных типов, на выходе – вектор, тип которого определяет *common\_type\_t*. Внутри функции вызывается функция обратной матрицы. Данная перегрузка необходима для того, чтобы найти решение СЛАУ  $Ax = b$ , написав следующее:  $x = b/A$ .

<pre> setlocale(LC_ALL, "Russian");  TMatrix&lt;double&gt; A(5, 5); for (int i = 0; i != 5; i++) {     for (int j = 0; j != 5; j++)     {         A[i][j] = rand() % 10;     } } cout &lt;&lt; "A = " &lt;&lt; endl &lt;&lt; A &lt;&lt; endl; TVector&lt;complex&lt;double&gt;&gt; b(5); for (int j = 0; j != 5; j++) {     b[j] = complex&lt;double&gt;(rand() % 10, rand() % 5); } cout &lt;&lt; "b = " &lt;&lt; b &lt;&lt; endl; TVector&lt;complex&lt;double&gt;&gt; x = b/A; cout &lt;&lt; "Ответ : x = " &lt;&lt; endl &lt;&lt; x &lt;&lt; endl; cout &lt;&lt; "Проверка A * x = " &lt;&lt; endl &lt;&lt; A*x &lt;&lt; endl; </pre>	<pre> A = 1 7 4 0 9 4 8 8 2 4 5 5 1 7 1 1 5 2 7 6 1 4 2 3 2  b =(1,2) (8,1) (7,0) (1,1) (9,3)  Ответ : x = (-2.74175,-1.48586) (5.31574,1.96984) (-1.33082,-0.638549) (-0.201697,-0.13148) (-3.12724,-0.86098)  Проверка A * x = (1,2) (8,1) (7,2.10942e-15) (1,1) (9,3) </pre>
---	---

Рис. 12: Пример решения СЛАУ

- Дружественная функция–оператор ввода вектора > >

Объявление в классе:

```
friend istream & operator >>
    <Type> (istream & input, TVector<Type> & obj);
```

Реализация вне класса:

```
template<typename Type>
istream & operator >> (istream & input, TVector<Type> & obj)
{
    for (int i = 0; i < obj.size; i++)
        input >> obj.ptr[i];
    return input;
}
```

- Дружественная функция–оператор вывода вектора < <

Объявление в классе:

```
friend ostream & ::operator<<
    <Type> (ostream & output, const TVector<Type>& object);
```

Реализация вне класса:

```
template<typename Type>
ostream & operator<< (ostream & output,
    const TVector<Type>& object)
{
    for (int i = 0; i != object.size; i++)
    {
        output << object.ptr[i] << " ";
    }
    output << endl;
    return output;
}
```

Как и в классе TVector, используя оператор typedef, определены следующие псевдонимы для матрицы из целочисленных значений, с двойной точностью и complex<double> :

```
typedef TMatrix<int> MatrixInt;
typedef TMatrix<double> MatrixDouble;
typedef TMatrix<complex<double>>MatrixComplex;
```

## 5 Возникшие проблемы при перегрузке оператора \* в классах

Приведем код для перегрузки оператора \* в классе TMatrix и TVector для умножения матрицы/вектора одного типа на скаляр другого типа: **Умножение матрицы справа на скаляр**

Объявление в классе:

```
template <typename Type, typename Type_other>
friend TMatrix<std::common_type_t<Type, Type_other>>
    operator *(TMatrix<Type> a, Type_other b);
```

Реализация вне класса:

```
template<typename Type, typename Type_other>
TMatrix<std::common_type_t<Type, Type_other>> operator*(TMatrix<Type> a,
{
    TMatrix<std::common_type_t<Type, Type_other>>res(a.row, a.col);
    for (int i = 0; i != a.row; i++)
        for (int j = 0; j != a.col; j++)
            res[i][j] = a[i][j]*b;
    return res;
}
```

**Умножение матрицы слева на скаляр**

Объявление в классе:

```
template <typename Type, typename Type_other>
friend TMatrix<std::common_type_t<Type, Type_other>>
    operator *( Type_other b, TMatrix<Type> a);
```

Реализация вне класса:

```
template<typename Type, typename Type_other>
TMatrix<std::common_type_t<Type, Type_other>> operator*(Type_other b,
{
    TMatrix<std::common_type_t<Type, Type_other>>res(a.row, a.col);
    for (int i = 0; i != a.row; i++)
        for (int j = 0; j != a.col; j++)
            res[i][j] = a[i][j]*b;
    return res;
}
```

## Умножение вектора справа на скаляр

Объявление в классе:

```
template <typename Type, typename Type_other>
friend TVector<std::common_type_t<Type, Type_other>>
    operator *(TVector<Type> a, Type_other b);
```

Реализация вне класса:

```
template<typename Type, typename Type_other>
TVector<std::common_type_t<Type, Type_other>> operator*(TVector<Type> a,
Type_other b)
{
    TVector<std::common_type_t<Type, Type_other>> res(a.size());
    for (int i = 0; i != a.row; i++)
        res[i] = a[i]*b;
    return res;
}
```

## Умножение вектора слева на скаляр

Объявление в классе:

```
template <typename Type, typename Type_other>
friend TVector<std::common_type_t<Type, Type_other>>
    operator *(Type_other b, TVector<Type> a);
```

Реализация вне класса:

```
template<typename Type, typename Type_other>
TVector<std::common_type_t<Type, Type_other>> operator*(Type_other b,
TVector<Type> a)
{
    TVector<std::common_type_t<Type, Type_other>> res(a.size());
    for (int i = 0; i != a.col; i++)
        res[i] = a[i]*b;
    return res;
}
```

После таких перегрузок умножение матрицы и вектора на скаляр слева и справа работало исправно, а умножение матрицы на матрицу и матрицы на вектор не работало и указывало на следующие ошибки, которые устранить не удалось: *C2446 — нет преобразования TVector<std::complex< double> > в complex<double>*

*C2955 — std::decay : для использования класс шаблон требуется список аргументов шаблон*

Обе ошибки указывали на следующий код в файле *type\_traits*:



```

template<class _Ty0,
class _Ty1>
struct common_type<_Ty0, _Ty1>
{ // type is common type of _Ty0 and _Ty1 for two arguments
typedef typename decay<
    decltype(_Always_false<_Ty0>::value
        ? _STD declval<_Ty0>()
        : _STD declval<_Ty1>())
>::type type;
};

```

После удаления вышеперечисленных перегрузок и написания перегрузок для умножения матрицы и вектора на скаляр, которые были приведены в описаниях классов, удалось умножить матрицу на матрицу и матрицу на вектор.

## 6 Листинг TVector.h

Листинг 1: C++ code using listings

---

```

1 #include <iostream>
2 #include <type_traits>
3 using namespace std;
4
5 template <typename Type>
6 class TVector;
7
8 typedef TVector<int> VectorInt;
9 typedef TVector<double> VectorDouble;
10 typedef TVector<complex<double>> VectorComplex;
11
12 template<typename Type>
13 ostream & operator<< (ostream & output,
14 const TVector<Type>& object)
15 {
16     for (int i = 0; i != object.size; i++)
17     {
18         output << object.ptr[i] << " ";
19     }
20     output << endl;
21     return output;
22 }
23

```

```

24 template<typename Type>
25 istream & operator >> (istream & input, TVector<Type> & obj)
26 {
27     for (int i = 0; i < obj.size; i++)
28         input >> obj.ptr[i];
29     return input;
30 }
31
32 template<typename Type, typename Type_other>
33 TVector<std::common_type_t<Type, Type_other>> operator +
34 (TVector<Type> &a, TVector<Type_other> &b)
35 {
36     TVector<std::common_type_t<Type, Type_other>> res(a.size);
37     if (a.size != b.size)
38     {
39         exit(1);
40     }
41     else
42     {
43         for (int i = 0; i != a.size; i++)
44         {
45             res[i] = a[i] + b[i];
46         }
47         return res;
48     }
49 }
50
51 template<typename Type, typename Type_other>
52 TVector<std::common_type_t<Type, Type_other>> operator -
53 (TVector<Type> &a, TVector<Type_other> &b)
54 {
55     TVector<std::common_type_t<Type, Type_other>> res(a.size);
56     if (a.size != b.size)
57     {
58         exit(1);
59     }
60     else
61     {
62         for (int i = 0; i != a.size; i++)
63         {
64             res[i] = a[i] - b[i];
65         }
66         return res;

```

```

67     }
68 }
69
70 template<typename Type, typename Type_other>
71 TVector<std::common_type_t<Type, Type_other>> operator *
72 (TVector<Type> &a, TVector<Type_other> &b)
73 {
74     TVector<std::common_type_t<Type, Type_other>> res(a.size);
75     if (a.size != b.size)
76     {
77         exit(1);
78     }
79     else
80     {
81         for (int i = 0; i != a.size; i++)
82         {
83             res[i] = a[i] * b[i];
84         }
85         return res;
86     }
87 }
88
89 template <typename Type>
90 class TVector
91 {
92     Type * ptr;
93     int size;
94 public:
95
96     TVector()
97     {
98         size = 0;
99         ptr = nullptr;
100     }
101
102     TVector(int size_)
103     {
104         size = size_;
105         ptr = new Type[size];
106         for (int i = 0; i != size; i++)
107         {
108             ptr[i] = 1.0;
109         }

```

```

110     }
111
112     TVector (const TVector<Type>& other)
113     {
114         size = other.size;
115         ptr = new Type[size];
116         for (size_t i = 0; i != size; ++i)
117             ptr[i] = other.ptr[i];
118     }
119
120     int get_size()
121     {
122         return size;
123     }
124
125     void show()
126     {
127         for (int i = 0; i != size; i++)
128         {
129             cout << ptr[i] << "    ";
130         }
131         cout << '\n';
132     }
133
134     Type & operator [] (int num)
135     {
136         try {
137             if ((num >= 0) && (num < size))
138                 return ptr[num];
139             else
140                 throw "size error";
141         }
142         catch (char *str)
143         {
144             cout << str << endl;
145         }
146     }
147
148     template<typename Type>
149     bool operator==(TVector<Type>& other)
150     {
151         if (this->size != other.size)
152             return false;

```

```

153         else
154             for (int i = 0; i != this->size; i++)
155             {
156                 if (this->ptr[i] != other.ptr[i])
157                 {
158                     return false;
159                 }
160             }
161         return true;
162     }
163
164     bool operator!=(int)
165     {
166         if (this == NULL)
167         {
168             return true;
169         }
170         else
171             return false;
172     }
173
174     template<typename Type>
175     bool operator!=(const TVector<Type>& other)
176     {
177         if (this->size != other.size)
178             return true;
179         else
180             for (int i = 0; i != this->size; i++)
181             {
182                 if (this->ptr[i] != other.ptr[i])
183                 {
184                     return true;
185                 }
186             }
187         return false;
188     }
189
190     template <typename Type, typename Type_other>
191     friend TVector<std::common_type_t<Type, Type_other>>
192     operator+ (TVector<Type> & a, TVector<Type_other> & b);
193
194     template <typename Type, typename Type_other>
195     friend TVector<std::common_type_t<Type, Type_other>>

```

```

196     operator- (TVector<Type> & a, TVector<Type_other> & b);
197
198     template<typename Type>
199     TVector<Type> operator*(Type b)
200     {
201         for (int i = 0; i != size; i++)
202         {
203             ptr[i] *= b;
204         }
205         return (*this);
206     }
207
208     template <typename Type, typename Type_other>
209     friend TVector<std::common_type_t<Type, Type_other>>
210     operator* (TVector<Type> & a, TVector<Type_other> & b);
211
212     template<typename U> operator TVector<U>()
213     {
214         TVector<U> temp(size);
215
216         for (int i = 0; i != size; i++)
217         {
218             temp[i] = static_cast<U>(ptr[i]);
219         }
220         return (temp);
221     }
222
223     TVector<Type>& operator=(TVector<Type>& other)
224     {
225         if (this != &other)
226         {
227             delete [] ptr;
228             this->size = other.size;
229             this->ptr = new Type[this->size];
230
231             for (int i = 0; i != this->size; i++)
232                 this->ptr[i] = other.ptr[i];
233         }
234         return (*this);
235     }
236
237     friend ostream & ::operator<< <Type> (ostream & output,
238     const TVector<Type>& object);

```

```

239
240     friend istream & operator >> <Type> (istream & input ,
241     TVector<Type> & obj);
242
243
244     ~TVector()
245     {
246         delete [] ptr;
247         ptr = nullptr;
248     }
249 };

```

---

## 7 Листинг TMatrix.h

Листинг 2: C++ code using listings

---

```

1  #include <iostream>
2  #include <algorithm>
3  #include <type_traits>
4  #include <complex>
5  #include "TVector.h"
6  using namespace std;
7
8  template<typename Type>
9  class TMatrix;
10
11  typedef TMatrix<int> MatrixInt;
12  typedef TMatrix<double> MatrixDouble;
13  typedef TMatrix<complex<double>> MatrixComplex;
14
15  template<typename Type>
16  ostream &operator<< (ostream &output , const TMatrix<Type>
17  &object)
18  {
19      for (int i = 0; i != object.row; i++)
20          output << object.vptr[i];
21      output << endl;
22      return output;
23  }
24
25  template<typename Type>
26  istream &operator >> (istream &input , TMatrix<Type> &obj)

```

```

27 {
28     for (int i = 0; i < obj.row; i++)
29         input >> obj.vptr[i];
30     return input;
31 }
32
33 template<typename Type, typename Type_other>
34 TMatrix<std::common_type_t<Type, Type_other>>
35 operator+(TMatrix<Type> a, TMatrix<Type_other> b)
36 {
37     TMatrix<std::common_type_t<Type, Type_other>> res(a.row,
38 a.col);
39     if ((a.row != b.row) || (a.col != b.col))
40     {
41         exit(1);
42     }
43     else
44     {
45         for (int i = 0; i != a.row; i++)
46         {
47             for (int j = 0; j != a.col; j++)
48                 res[i][j] = a[i][j] + b[i][j];
49         }
50         return res;
51     }
52 }
53
54 template<typename Type, typename Type_other>
55 TMatrix<std::common_type_t<Type, Type_other>>
56 operator-(TMatrix<Type> a, TMatrix<Type_other> b)
57 {
58     TMatrix<std::common_type_t<Type, Type_other> >
59     res(a.row, a.col);
60     if ((a.row != b.row) || (a.col != b.col))
61     {
62         exit(1);
63     }
64     else
65     {
66         for (int i = 0; i != a.row; i++)
67         {
68             for (int j = 0; j != a.col; j++)
69                 res[i][j] = a[i][j] - b[i][j];

```



```

70         }
71         return res;
72     }
73 }
74
75 template<typename Type, typename Type_other>
76 TMatrix<std::common_type_t<Type, Type_other>>
77 operator*(TMatrix<Type> a, TMatrix<Type_other> b)
78 {
79     TMatrix<std::common_type_t<Type, Type_other>> res(a.get_co
80     if (a.get_col() != b.get_row())
81     {
82         exit(1);
83     }
84     else
85     {
86         for (int i = 0; i != a.row; i++)
87         {
88             for (int j = 0; j != b.col; j++)
89             {
90                 res[i][j] = 0.0;
91                 for (int k = 0; k != a.col; k++)
92                     res[i][j] += a[i][k] * b[k][j];
93             }
94         }
95         return res;
96     }
97 }
98
99 template<typename Type, typename Type_other>
100 TVector<std::common_type_t<Type, Type_other>>
101 operator*(TMatrix<Type> a, TVector<Type_other> b)
102 {
103     TVector<std::common_type_t<Type, Type_other>> res(a.row);
104     if (a.col != b.get_size())
105     {
106         cerr << "wrong size";
107         exit(1);
108     }
109     else
110     {
111         for (int i = 0; i != a.row; i++)
112         {

```

```

113         res[i] = 0;
114         for (int j = 0; j != a.col; j++)
115         {
116             res[i] += a[i][j] * b[j];
117         }
118     }
119 }
120 return res;
121 }
122
123 template <typename Type>
124 TMatrix<Type> & operator / (double x, TMatrix<Type>
125     & other)
126 {
127     TMatrix<Type> *inv = new TMatrix<Type>(other.row,
128         other.col);
129     *inv = inv->inverse(other);
130     return *inv;
131 }
132
133 template <typename Type, typename U>
134 TVector<std::common_type_t<Type, U>> & operator /
135 (TVector<U> & b, TMatrix<Type> & other)
136 {
137     TMatrix<Type> inv(other.row, other.col);
138     inv = inv.inverse(other);
139     TVector<std::common_type_t<Type, U>> *res= new
140     TVector<std::common_type_t<Type, U>>(other.row);
141     *res = inv*b;
142     return *res;
143 }
144
145
146
147 template<typename Type>
148 class TMatrix
149 {
150     int row, col;
151     TVector<Type> * vptr;
152 public:
153     TMatrix() {}
154     TMatrix(int row, int col)
155     {

```

```

156         this->row = row;
157         this->col = col;
158         vptr = new TVector<Type>[row];
159         for (int i = 0; i != row; i++)
160         {
161             vptr[i] = TVector<Type>(col);
162         }
163         for (int i = 0; i != row; i++)
164         {
165             for (int j = 0; j != col; j++)
166                 vptr[i][j] = 0.0;
167         }
168     }
169     TMatrix(int row)
170     {
171         this->col = this->row = row;
172
173         vptr = new TVector<Type>[row];
174         for (int i = 0; i != row; i++)
175         {
176             vptr[i] = TVector<Type>(row);
177         }
178         for (int i = 0; i != row; i++)
179         {
180             for (int j = 0; j != row; j++)
181                 if (i == j)
182                     vptr[i][j] = 1.0;
183                 else vptr[i][j] = 0.0;
184         }
185     }
186
187     ~TMatrix()
188     {
189         delete[] vptr;
190         vptr = nullptr;
191     }
192
193     TVector<Type> & operator [] (int num)
194     {
195         try {
196             if ((num >= 0) && (num < row))
197                 return vptr[num];
198             else

```

```

199         throw "size error";
200     }
201     catch (char *str)
202     {
203         cout << str << endl;
204     }
205 }
206
207 TMatrix(const TMatrix<Type> & other)
208 {
209     this->row = other.row;
210     this->col = other.col;
211
212     this->vptr = new TVector<Type>[row];
213     for (int i = 0; i != row; i++)
214         this->vptr[i] = TVector<Type>(col);
215     for (int i = 0; i != row; i++)
216         this->vptr[i] = other.vptr[i];
217 }
218
219 Type determinant()
220 {
221     if (this->col != this->row)
222     {
223         exit(1);
224     }
225     TMatrix<Type> B = *this;
226     for (int step = 0; step != row; step++)
227     {
228         for (int r = step + 1; r != row; r++)
229         {
230             if (B[step][step] != 0.0)
231             {
232                 Type coef = -B[r][step] / B[step][step];
233                 for (int c = step; c != row; c++)
234                     B[r][c] += B[step][c] * coef;
235             }
236             else
237             {
238                 cerr << "singular matrix" << endl;
239                 system("pause");
240                 exit(1);
241             }

```

```

242
243     }
244 }
245 Type det = 1.0;
246 for (int i = 0; i != row; i++)
247 {
248     det *= B[i][i];
249 }
250 return det;
251
252 }
253 int get_row()
254 {
255     return row;
256 }
257 int get_col()
258 {
259     return col;
260 }
261
262 template <typename Type, typename Type_other>
263 friend TMatrix <std::common_type_t<Type, Type_other>>
264 operator +(TMatrix<Type> a, TMatrix<Type_other> b);
265
266 template <typename Type, typename Type_other>
267 friend TMatrix <std::common_type_t<Type, Type_other>>
268 operator -(TMatrix<Type> a, TMatrix<Type_other> b);
269
270 template <typename Type, typename Type_other>
271 friend TMatrix <std::common_type_t<Type, Type_other>>
272 operator *(TMatrix<Type> a, TMatrix<Type_other> b);
273
274
275 template<typename Type>
276 TMatrix<Type> & inverse(TMatrix<Type> & other)
277 {
278     TMatrix<Type> * inver = new TMatrix<Type>(other.get_row()
279         other.get_col());
280     TMatrix<Type> temp = other;
281     int ki, kj;
282     for (int indexi = 0; indexi != temp.row; indexi++)
283     {
284         for (int indexj = 0; indexj != temp.col; indexj++)

```

```

285     {
286         ki = indexi; kj = indexj;
287         TMatrix<Type> adj(temp.row - 1, temp.col - 1);
288         int si = 0, sj = 0;
289         for (int i = 0; i != temp.row; i++)
290         {
291             if (i != ki)
292             {
293                 sj = 0;
294                 for (int j = 0; j != temp.col; j++)
295                 {
296                     if (j != kj)
297                     {
298                         adj[si][sj] = temp[i][j];
299                         sj++;
300                     }
301                 }
302                 si++;
303             }
304         }
305         (*inver)[indexi][indexj] = adj.determinant()
306             *pow(-1.0, indexi + indexj);
307     }
308 }
309
310 for (int indexi = 0; indexi != temp.row-1; indexi++)
311 {
312     for (int indexj = indexi+1; indexj != temp.col;
313         indexj++)
314     {
315         swap((*inver)[indexj][indexi],
316             (*inver)[indexi][indexj]);
317     }
318 }
319 Type det = temp.determinant();
320 for (int indexi = 0; indexi != temp.row; indexi++)
321 {
322     for (int indexj = 0; indexj != temp.col; indexj++)
323     {
324         (*inver)[indexi][indexj] =
325             (*inver)[indexi][indexj] * 1.0 / det;
326     }
327 }

```

```

328         return (*inver);
329     }
330
331     template <typename Type, typename Type_other>
332     friend TVector<std::common_type_t<Type, Type_other>>
333     operator *(TMatrix<Type> a, TVector<Type_other> b);
334
335     template<typename U>
336     operator TMatrix<U>()
337     {
338         TMatrix<U> temp(row, col);
339         for (int i = 0; i != row; i++)
340         {
341             for (int j = 0; j != col; j++)
342             {
343                 temp[i][j] =
344                     static_cast<U>(this->vptr[i][j]);
345             }
346         }
347         return temp;
348     }
349
350     template<typename Type>
351     friend TMatrix<Type> & operator / (double x,
352     TMatrix<Type> & other);
353
354     template<typename Type>
355     TMatrix<Type> operator*(Type b)
356     {
357         for (int i = 0; i != row; i++)
358         {
359             for(int j = 0 ; j!=col;j++)
360                 vptr[i][j] *= b;
361         }
362         return (*this);
363     }
364
365     TMatrix<Type>& operator= (TMatrix<Type> & other)
366     {
367         if (this != &other)
368         {
369
370             delete [] vptr;

```

```

371         col = other.col;
372         row = other.row;
373         vptr = new TVector<Type>[row];
374         for (int i = 0; i != row; i++)
375             vptr[i] = TVector<Type>(col);
376         for (int i = 0; i != row; i++)
377             for (int j = 0; j != col; j++)
378                 vptr[i][j] = other.vptr[i][j];
379     }
380     return (*this);
381 }
382 friend ostream &::operator<< <Type> (ostream &output,
383 const TMatrix<Type> &object);
384 friend istream &operator>> <Type> (istream &input,
385 TMatrix<Type> &obj);
386 void show()
387 {
388     for (int i = 0; i != row; i++)
389     {
390         for (int j = 0; j != col; j++)
391         {
392             cout << vptr[i][j] << " ";
393         }
394         cout << endl;
395     }
396 }
397 };

```

---