МИНИСТЕРСТВО НАУКИ И ВЫСШЕГООБРАЗОВАНИЯ
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:

«ВЕКТОРА И МАТРИЦЫ»

Выполнил(а): студент группы 3822Б1ФИ1
_____ / Петров Н.Д. /
Подпись

Проверил: к.т.н., доцент каф. ВВиСП
_____ / Кустикова В.Д. /
Подпись

Нижний Новгород
2023

Оглавление

Введение 3

1 Постановка задачи 4

2 Руководство пользователя 5

2.1 Приложение для демонстрации работы векторов 5

2.2 Приложение для демонстрации работы матриц 6

3 Руководство программиста 7

3.1 Использованные алгоритмы 7

3.1.1 Вектор 7

3.1.2 Матрица 7

3.2 Описание классов 8

3.2.1 Класс Vector 8

3.2.2 Класс Matrix 12

Заключение 15

Литература 16

Приложения 17

Приложение А. Реализация класса Vector 17

Приложение Б. Реализация класса Matrix 19

Приложение В. Реализация класса sample_vector 20

Введение

В информатике и линейной алгебре вектор — это абстрактный математический объект, который представляет собой упорядоченный набор чисел (элементов), хранящихся в одномерной структуре данных. Векторы часто используются для представления данных, которые имеют линейную структуру, такие как координаты точек в пространстве, компоненты цветов в изображениях и другие сущности.

Одной из интересных и важных абстракций, которые можно представить в виде векторов, являются верхне-треугольные матрицы. Верхне-треугольная матрица — это матрица, у которой все элементы ниже главной диагонали равны нулю. Это часто встречается в различных приложениях, таких как численные методы, где сохранение нулей в таких матрицах может существенно ускорить вычисления.

С использованием векторов, в частности, вектора векторов, верхне-треугольные матрицы можно представить в более компактной форме, не храня нулевые элементы. Такое представление позволяет экономить память и оптимизировать операции над матрицей, такие как сложение и умножение.

1 Постановка задачи

Цель – разработка и реализация класса **`Vector`** для представления векторов и класса **`Matrix`** для представления верхне-треугольных матриц.

Задачи:

1. Исследовать тематическую литературу.
2. Реализовать класс **`Vector`**.
3. Реализовать класс **`Matrix`**.
4. Провести тестирование разработанных классов для проверки их корректной работы.

2 Руководство пользователя

2.1 Приложение для демонстрации работы векторов

1. Запустить sample_vector.exe. В результате появится следующее окно (рис. 1).

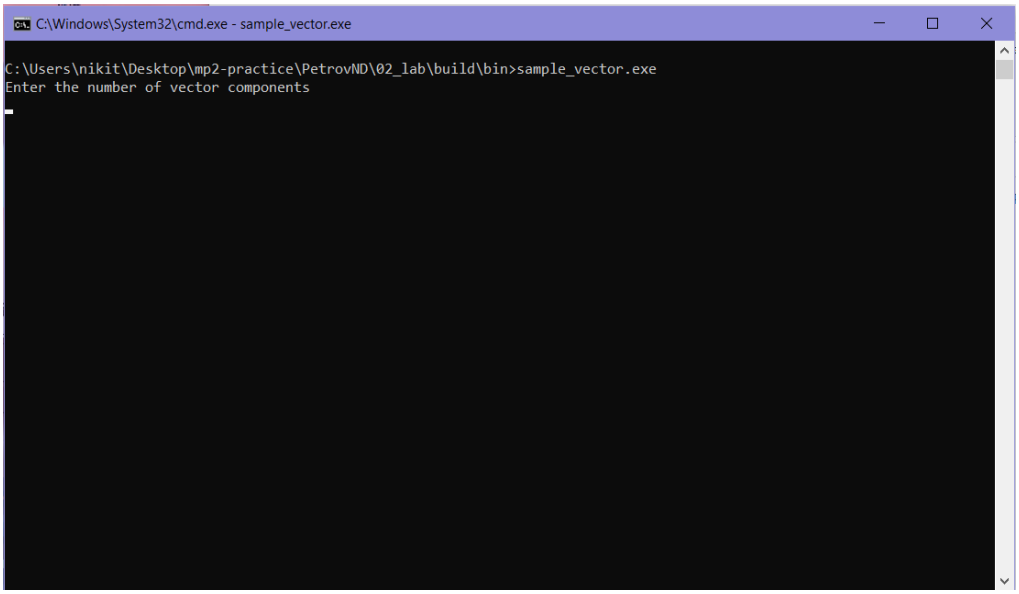

Рис. 1. Основное окно приложения sample_ vector

2. Необходимо ввести размер одного вектора (остальные вектора, необходимые для демонстрации, будут сгенерированы автоматически), затем вам будет предложено ввести этот вектор и скаляр, на который он умножится. Далее будет продемонстрирована работа класса вектор (рис. 2).

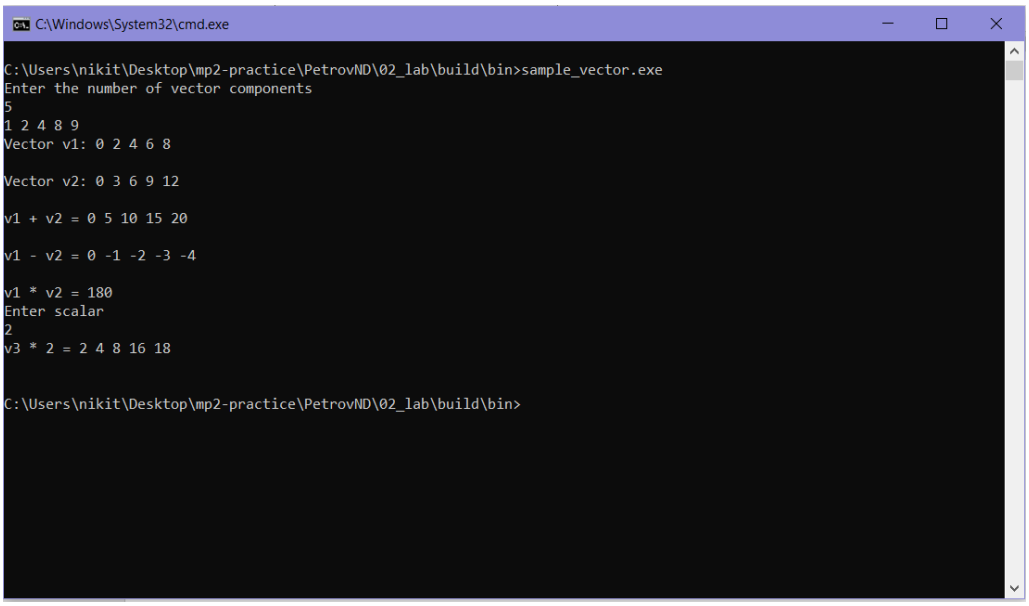

Рис. 2. Основное окно приложения sample_vector

2.2 Приложение для демонстрации работы матриц

1. Запустить sample_matrix.exe. В результате появится следующее окно (рис. 3).

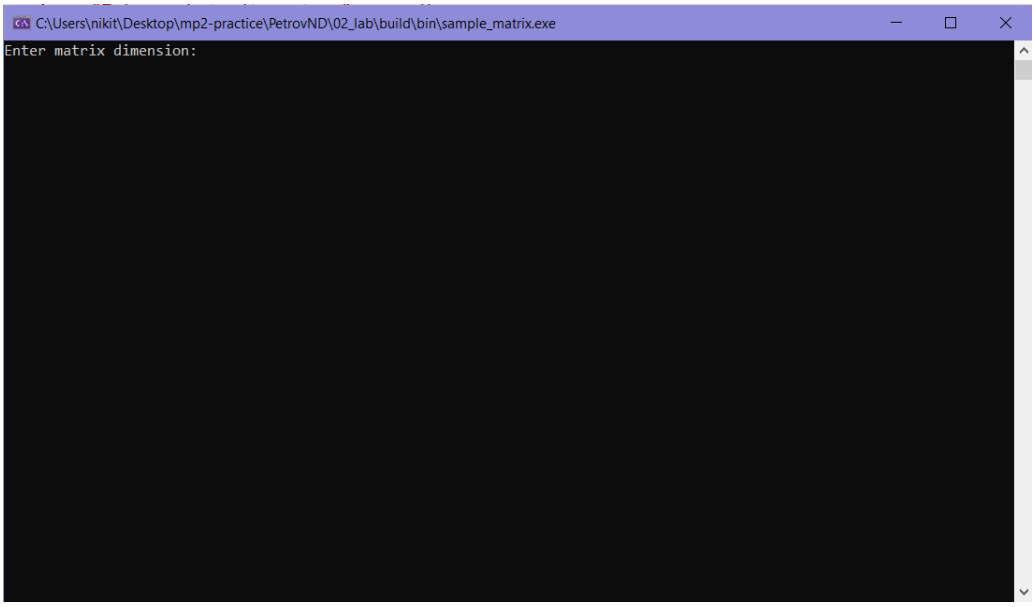

Рис. 3. Основное окно приложения sample_matrix

2. Необходимо ввести размерность матрицы, затем будет предложены ввести эту матрицу. Далее будет продемонстрирована работа класса матрица (рис. 4).

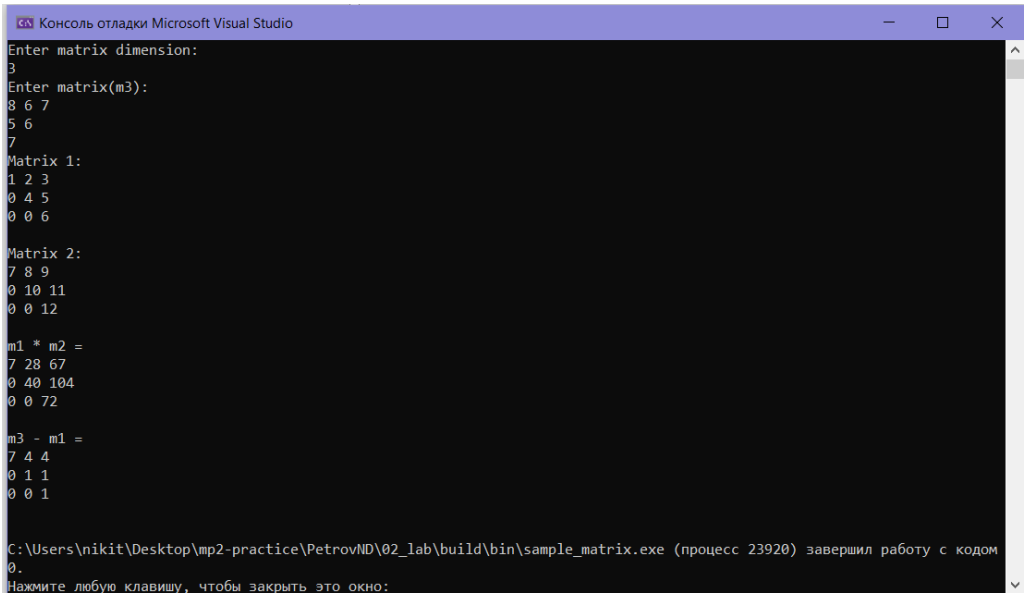

Рис. 4. Основное окно приложения sample_matrix

3 Руководство программиста

3.1 Использованные алгоритмы

3.1.1 Вектор

Вектор – абстрактная структура данных, которая представляет собой упорядоченный набор элементов одного типа. Основная его характеристика – размер(длина). Это число определяет количество элементов, хранящихся в векторе. Доступ к элементам осуществляется по индексу элемента.

Пример вектора длины 5.

3	11	-4	12	9

Векторные операции:

1. Сложение и вычитание векторов. Осуществляется путём поэлементного сложения(вычитания).

A	2	11	15	-18	4
B	6	15	-12	-21	0
A-B	-4	-4	27	3	4

2. Сложение, вычитание, умножение вектора на скаляр. Сложение/вычитание/умножение всех элементов вектора на скаляр.
3. Операция скалярного произведения векторов. Скалярное произведение – это сумма произведений соответствующих по номеру элементов двух векторов.

A	2	11	15	-18	4
B	6	15	-12	-21	0
A*B	2*6 + 11*15 + 15*(-12) + (-18)*(-21) +4*0 = 12 + 165 - 180 + 378 + 0 = 375				

3.1.2 Матрица

Верхне-треугольная матрица представляется в виде вектора векторов. Верхне-треугольная матрица — это матрица, у которой все элементы ниже главной диагонали равны нулю.

Матричные операции:

1. Сложение и вычитание матриц. Каждый элемент одной матрицы складывается/вычитается с элементом другой матрицы соответственно.

Матрица А:

1	2	3
0	4	5
0	0	6

Матрица Б:

2	1	5
0	8	7
0	0	3

Матрица А+Б:

3	3	8
0	12	12
0	0	9

2. Умножение матриц. Первый элемент матрицы 1 соотносится с первым элементом первого столбца матрицы 2. Они перемножаются. Это и будет первый результирующий элемент результирующей матрицы в первой строке. Первые 2 элемента первой строки соотносятся с первыми двумя элементами второго столбца. Выполняется попарное умножение соответствующих элементов и результаты суммируются. Это и будет второй элемент первой строки результирующей матрицы. Обобщая по формуле:

$$\sum_{k=1}^{n} a_{ik} * b_{kj}$$

где n – размер квадратной матрицы; a_{ik} – элемент первой исходной матрицы, расположенный на пересечении i-ого столбца и k-ой строки; b_{kj} – элемент второй исходной матрицы, расположенный на пересечении k-ого столбца и j-ой строки.

3.2 Описание классов

3.2.1 Класс Vector

Объявление класса:

```
template <typename ValueType> class Vector {
```

```
protected:
    int size;
    int startIndex;
    ValueType* pVector;
public:
    Vector(int size = 0, int startIndex = 0);
    Vector(const Vector& v);
    ~Vector();

    int getSize() const;
    int getStartIndex() const;

    ValueType& operator[](const int index);
    int operator==(const Vector& v) const;
    int operator!=(const Vector& v) const;
    const Vector& operator=(const Vector& v);
    Vector operator+(const ValueType vt);
    Vector operator-(const ValueType vt);
    Vector operator*(const ValueType vt);
    Vector operator+(const Vector& v);
    Vector operator-(const Vector& v);
    ValueType operator*(const Vector& v);

    friend std::istream& operator>>(std::istream& in, Vector& v);

    friend std::ostream& operator<<(std::ostream& out, const Vector& v);
};
```

Поля:

size хранит размер (длину) вектора. Оно указывает, сколько элементов содержит данный вектор.

startIndex хранит начальный индекс вектора.

pVector является указателем на массив элементов вектора. Он используется для хранения самих элементов вектора. Тип элементов вектора определяется шаблонным параметром **ValueType**.

Методы:

Vector(int size = 0, int startIndex = 0)

Назначение: создает объект класса **Vector** с заданным размером и начальным индексом.

Входные данные: **size** - размер вектора, **startIndex** - начальный индекс вектора.

Выходные данные: отсутствуют.

Vector(const Vector& v)

Назначение: создает копию объекта класса **Vector**.

Входные данные: ссылка на объект класса **Vector**.

Выходные данные: отсутствуют.

~Vector()

Назначение: освобождает выделенную память, используемую под вектор.

Входные данные: отсутствуют.

Выходные данные: отсутствуют.

int getSize() const

Назначение: возвращает размер (длину) вектора.

Входные данные: отсутствуют.

Выходные данные: целое число, представляющее размер (длину) вектора.

int getStartIndex() const

Назначение: возвращает начальный индекс вектора.

Входные данные: отсутствуют.

Выходные данные: целое число, представляющее начальный индекс вектора.

ValueType& operator[](const int index)

Назначение: возвращает ссылку на элемент вектора по заданному индексу.

Входные данные: целочисленный индекс.

Выходные данные: ссылка на элемент вектора.

int operator==(const Vector& v) const

Назначение: сравнивает два вектора на равенство.

Входные данные: ссылка на объект класса **Vector**.

Выходные данные: 1, если векторы равны, 0 – в противном случае.

int operator!=(const Vector& v) const

Назначение: сравнивает два вектора на неравенство.

Входные данные: ссылка на объект класса **Vector**.

Выходные данные: 1, если векторы неравны, 0 – в противном случае.

const Vector& operator=(const Vector& v)

Назначение: присваивает значения вектора из другого вектора.

Входные данные: ссылка на объект класса **Vector**.

Выходные данные: ссылка на текущий объект класса **Vector** после присваивания.

Vector operator+(const ValueType vt)

Назначение: выполняет операцию сложения вектора с скаляром.

Входные данные: значение скаляра.

Выходные данные: новый объект класса **Vector** – результат сложения.

Vector operator-(const ValueType vt)

Назначение: выполняет операцию вычитания из вектора скаляра.

Входные данные: значение скаляра.

Выходные данные: новый объект класса **Vector** – результат вычитания.

Vector operator*(const ValueType vt)

Назначение: выполняет операцию умножения вектора на скаляр.

Входные данные: значение скаляра.

Выходные данные: новый объект класса **Vector** – результат умножения.

Vector operator+(const Vector& v)

Назначение: выполняет операцию сложения двух векторов.

Входные данные: ссылка на объект класса **Vector**.

Выходные данные: новый объект класса **Vector** – результат сложения.

Vector operator-(const Vector& v)

Назначение: выполняет операцию вычитания из текущего вектора другого вектора.

Входные данные: ссылка на объект класса **Vector**.

Выходные данные: новый объект класса **Vector** - результат вычитания.

ValueType operator*(const Vector& v)

Назначение: выполняет операцию скалярного произведения двух векторов.

Входные данные: ссылка на объект класса **Vector**.

Выходные данные: значение типа **ValueType** – результат скалярного произведения.

friend std::istream& operator>>(std::istream& in, Vector& v)

Назначение: перегруженный оператор ввода для вектора.

Входные данные: поток ввода, ссылка на объект класса **Vector**.

Выходные данные: ссылка на поток ввода.

friend std::ostream& operator<<(std::ostream& out, const Vector& v)

Назначение: перегруженный оператор вывода для вектора.

Входные данные: поток вывода, константная ссылка на объект класса **Vector**.

Выходные данные: ссылка на поток вывода.

3.2.2 Класс Matrix

Объявление класса:

```
template <typename ValueType>
class Matrix : public Vector<Vector<ValueType>> {
public:
      Matrix(int size);
      Matrix(const Matrix& m);
      Matrix(const Vector<Vector<ValueType>>& m);
      int operator==(const Matrix& m) const;
      int operator!=(const Matrix& m) const;
      const Matrix& operator=(const Matrix& m);
      Matrix operator+(const Matrix& m);
      Matrix operator-(const Matrix& m);
      Matrix operator*(const Matrix& m);

      friend istream& operator>>(istream& in, Matrix<ValueType>& m);

      friend ostream& operator<<(ostream& out, Matrix<ValueType>& m);
};
```

Matrix(int size)

Назначение: конструктор класса **Matrix**, создающий матрицу заданного размера.

Входные данные: целое число **size** – размер матрицы.

Matrix(const Matrix& m)

Назначение: конструктор копирования для создания матрицы на основе существующей.

Входные данные: константная ссылка на объект типа **Matrix**.

Matrix(const Vector<Vector<ValueType>>& m)

Назначение: конструктор для создания матрицы на основе двумерного вектора.

Входные данные: двумерный вектор.

int operator==(const Matrix& m) const

Назначение: оператор сравнения матриц на равенство.

Входные данные: константная ссылка на объект типа **Matrix**.

Выходные данные: целое число, равное 1, если матрицы равны, и 0 в противном случае.

int operator!=(const Matrix& m) const

Назначение: оператор сравнения матриц на неравенство.

Входные данные: константная ссылка на объект типа **Matrix.**

Выходные данные: целое число, равное 1, если матрицы не равны, и 0 в противном случае.

const Matrix& operator=(const Matrix& m)

Назначение: оператор присваивания для матриц.

Входные данные: константная ссылка на объект типа **Matrix.**

Выходные данные: константная ссылка на объект типа **Matrix.**

Matrix operator+(const Matrix& m)

Назначение: оператор сложения матриц.

Входные данные: константная ссылка на объект типа **Matrix.**

Выходные данные: новая матрица, являющаяся результатом сложения текущей матрицы и переданной матрицы.

Matrix operator-(const Matrix& m)

Назначение: оператор вычитания матриц.

Входные данные: константная ссылка на объект типа **Matrix.**

Выходные данные: новая матрица, являющаяся результатом вычитания из текущей матрицы переданной матрицы.

Matrix operator*(const Matrix& m)

Назначение: оператор умножения матриц.

Входные данные: константная ссылка на объект типа **Matrix.**

Выходные данные: новая матрица, являющаяся результатом умножения текущей матрицы на переданную матрицу.

friend istream& operator>>(istream& in, Matrix<ValueType>& m)

Назначение: перегрузка оператора ввода для матрицы.

Входные данные: стандартный поток ввода и ссылка на объект типа **Matrix.**

Выходные данные: ссылка на поток ввода.

friend ostream& operator<<(ostream& out, Matrix<ValueType>& m)

Назначение: перегрузка оператора вывода для матрицы.

Входные данные: стандартный поток вывода и ссылка на объект типа **`Matrix`**.

Выходные данные: ссылка на поток вывода.

Заключение

В рамках данной работы был разработан и реализован класс **`Vector`**, предназначенный для представления векторов, и класс **`Matrix`**, наследующийся от класса **`Vector`**, для представления верхне-треугольных матриц. Реализованные классы поддерживают основные операции над векторами и матрицами, а также операции ввода и вывода.

Литература

1. Metanit.com [https://metanit.com/cpp/tutorial/7.2.php]

2. Microsoft Learn [https://learn.microsoft.com/ru-ru/cpp/standard-library/vector-class?view=msvc-170]

3. StackOverflow [https/stackoverflow.com/questions/43418270/multiplying-two-compressed-upper-triangular-matrices]

Приложения

Приложение А. Реализация класса Vector

```
template <typename ValueType>
Vector<ValueType>::Vector(int size, int startIndex) : size(size),
startIndex(startIndex) {
    pVector = new ValueType[size];
}

template <typename ValueType>
Vector<ValueType>::Vector(const Vector& v) : size(v.size),
startIndex(v.startIndex) {
    pVector = new ValueType[size];
    for (int i = 0; i < size; ++i) {
        pVector[i] = v.pVector[i];
    }
}

template <typename ValueType>
Vector<ValueType>::~Vector() {
    delete[] pVector;
}

template <typename ValueType>
int Vector<ValueType>::getSize() const { return size; }

template <typename ValueType>
int Vector<ValueType>::getStartIndex() const { return startIndex; }

template <typename ValueType>
ValueType& Vector<ValueType>::operator[](const int index) {
    if (index < 0 || index >= size) throw std::out_of_range("Index is out of
range");
    return pVector[index];
}

template <typename ValueType>
int Vector<ValueType>::operator==(const Vector& v) const {
    if (size != v.size) return 0;
    for (int i = 0; i < size; ++i) {
        if (pVector[i] != v.pVector[i]) {
            return 0;
        }
    }
    return true;
}

template <typename ValueType>
int Vector<ValueType>::operator!=(const Vector& v) const {
    return !(*this == v);
}

template <typename ValueType>
const Vector<ValueType>& Vector<ValueType>::operator=(const Vector& v) {
    if (this == &v) return *this;
    delete[] pVector;
    size = v.size;
    startIndex = v.startIndex;
    pVector = new ValueType[size];
    for (int i = 0; i < size; ++i) {
```

```
            pVector[i] = v.pVector[i];
        }
        return *this;
    }

    template <typename ValueType>
    Vector<ValueType> Vector<ValueType>::operator+(const ValueType vt) {
        Vector result(*this);
        for (int i = 0; i < size; ++i) {
            result.pVector[i] += vt;
        }
        return result;
    }

    template <typename ValueType>
    Vector<ValueType> Vector<ValueType>::operator-(const ValueType vt) {
        Vector result(*this);
        for (int i = 0; i < size; ++i) {
            result.pVector[i] -= vt;
        }
        return result;
    }

    template <typename ValueType>
    Vector<ValueType> Vector<ValueType>::operator*(const ValueType vt) {
        Vector result(*this);
        for (int i = 0; i < size; ++i) {
            result.pVector[i] *= vt;
        }
        return result;
    }

    template <typename ValueType>
    Vector<ValueType> Vector<ValueType>::operator+(const Vector& v) {
        if (size != v.size) throw std::invalid_argument("Vectors must have the
    same size for addition.");

        Vector result(*this);
        for (int i = 0; i < size; ++i) {
            result.pVector[i] += v.pVector[i];
        }
        return result;
    }

    template <typename ValueType>
    Vector<ValueType> Vector<ValueType>::operator-(const Vector& v) {
        if (size != v.size) throw std::invalid_argument("Vectors must have the
    same size for subtraction.");

        Vector result(*this);
        for (int i = 0; i < size; ++i) {
            result.pVector[i] -= v.pVector[i];
        }
        return result;
    }

    template <typename ValueType>
    ValueType Vector<ValueType>::operator*(const Vector& v) {
        if (size != v.size) throw std::invalid_argument("Vectors must have the
    same size for dot product.");

        ValueType result = 0;
        for (int i = 0; i < size; ++i) {
```

```
            result += pVector[i] * v.pVector[i];
        }
        return result;
}
```

Приложение Б. Реализация класса Matrix

```
template <typename ValueType>
Matrix<ValueType>::Matrix(int size) : Vector<Vector<ValueType>>(size) {
      for (int i = 0; i < size; ++i) {
            Vector<ValueType> row(size - i, i);
            this->pVector[i] = row;
      }
}

template <typename ValueType>
Matrix<ValueType>::Matrix(const Matrix& m) : Vector<Vector<ValueType>>(m) { }

template <typename ValueType>
Matrix<ValueType>::Matrix(const      Vector<Vector<ValueType>>&      m)      :
Vector<Vector<ValueType>>(m) { }

template <typename ValueType>
int Matrix<ValueType>::operator==(const Matrix& m) const {
      return Vector<Vector<ValueType>>::operator==(m);
}

template <typename ValueType>
int Matrix<ValueType>::operator!=(const Matrix& m) const {
      return Vector<Vector<ValueType>>::operator!=(m);
}

template <typename ValueType>
const Matrix<ValueType>& Matrix<ValueType>::operator=(const Matrix& m) {
      Vector<Vector<ValueType>>::operator=(m);
      return *this;
}

template <typename ValueType>
Matrix <ValueType> Matrix<ValueType>::operator+(const Matrix& m) {
      if (this->size != m.size) throw invalid_argument("Matrices must have the
same size for addition.");

      Matrix result(*this);
      for (int i = 0; i < this->size; ++i) {
            result.pVector[i] = result.pVector[i] + m.pVector[i];
      }
      return result;
}

template <typename ValueType>
Matrix <ValueType> Matrix<ValueType>::operator-(const Matrix& m) {
      if (this->size != m.size) throw invalid_argument("Matrices must have the
same size for subtraction.");

      Matrix result(*this);
      for (int i = 0; i < this->size; ++i) {
            result.pVector[i] = result.pVector[i] - m.pVector[i];
      }
      return result;
}

template <typename ValueType>
```

```
Matrix<ValueType> Matrix<ValueType>::operator*(const Matrix& m) {
      if (this->size != m.size) throw invalid_argument("Matrices must have the
same size for matrix multiplication.");

      int size = this->getSize();
      Matrix<ValueType> result(size);

      for (int k = 0; k < size; k++) {
            for (int j = k; j < size; j++) {
                  ValueType sum = 0;
                  for (int r = k; r <= j; r++) {
                        sum += this->pVector[k][r - k] * m.pVector[r][j - r];
                  }
                  result.pVector[k][j - k] = sum;
            }
      }
      return result;
}
```

Приложение В. Реализация класса sample_vector

```
int main() {
    try {
            int userInput;
            cout << "Enter the number of vector components" << endl;
            cin >> userInput;

        Vector<int> v1(5);
            Vector<int> v2(5);
            Vector<int> v3(userInput);

            cin >> v3;

            for (int i = 0; i < 5; ++i) {
                  v1[i] = i * 2;
                  v2[i] = i * 3;
            }

            cout << "Vector v1: " << v1 << endl;
            cout << "Vector v2: " << v2 << endl;

            cout << "v1 + v2 = " << v1 + v2 << endl;
            cout << "v1 - v2 = " << v1 - v2 << endl;
            cout << "v1 * v2 = " << v1 * v2 << endl;

            int scalar;
            cout << "Enter scalar" << endl;
            cin >> scalar;
            cout << "v3 * " << scalar << " = " << v3 * scalar << endl;
    }
    catch (const exception& e) {
        cerr << "Error: " << e.what() << endl;
    }
    return 0;
}
```