МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение

высшего образования

**«Нижегородский государственный университет им. Н.И. Лобачевского»**

**Национальный исследовательский университет**

**Институт информационных технологий, математики и механики**

**Кафедра математического обеспечения и суперкомпьютерных технологий**

**Отчет по Лабораторной работе**

**«Матрицы на шаблонах»**

**Выполнил:** студент группы 3822Б1ПР2

Страхов Андрей Андреевич

\_\_\_\_\_\_\_\_\_\_\_ Подпись

Нижний Новгород

2023.

Содержание

1. Введение……………………………………………….………….……3
2. Постановка задачи………………………………………………..……4
3. Руководство пользователя…………………………………….………5
4. Руководство программиста………………………………….………...6
5. Эксперименты…………………………………………….……..……..9
6. Заключение………………………………………………….………...10
7. Литература……………………………………………….……….…...11
8. Приложение………………………………………………..………….12
9. Введение

Матрица является математическим объектом, представленным в виде прямоугольной таблицы элементов из кольца или поля, таких как целые, действительные или комплексные числа. Она состоит из строк и столбцов, где каждый элемент находится на их пересечении, а количество строк и столбцов определяет размер матрицы. Матрица также может быть интерпретирована как функция двух дискретных аргументов.

Операции, которые можно выполнять с матрицами, включают:

* Сложение матриц одинакового размера;
* Умножение матриц подходящего размера (матрицу с n столбцами можно умножить справа на матрицу с n строками);
* Включая умножение матрицы на вектор-столбец и умножение вектор-строки на матрицу (согласно обычным правилам матричного умножения), где вектор рассматривается как частный случай матрицы;
* Умножение матрицы на скаляр.

# Постановка задачи

Выполнение работы предполагает решение следующих задач:

1. Реализация методов шаблонного класса `TDynamicVector` согласно заданному интерфейсу.
2. Реализация методов шаблонного класса `TDynamicMatrix` согласно заданному интерфейсу.
3. Обеспечение работоспособности тестов и примера использования.
4. Реализация заготовок тестов, покрывающих все методы классов `TDynamicVector` и `TDynamicMatrix`.
5. Модификация примера использования в тестовое приложение, позволяющее задавать матрицы и осуществлять основные операции над ними.

# Руководство пользователя

test\_tmatrix и test\_tvector – это тесты для проверки классов TDynamicVector и TDynamicMatrix, test\_main сразу проведёт оба теста. При запуске тестов запускается консоль, в которую выводится результат выполнения тестов, их количество и итог, в случае неуспешного прохождения теста выводятся подробности: полученный и ожидаемый результат.

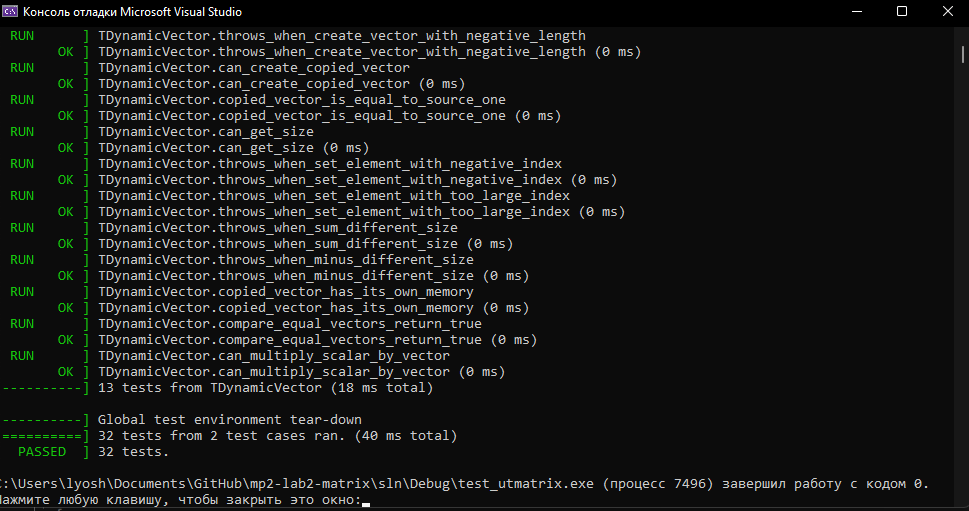


Рис.1 (Вывод при запуске test\_main)

При запуске sample\_matrix запускается консоль, генерируется и выводятся две матрицы и результат их сложения.

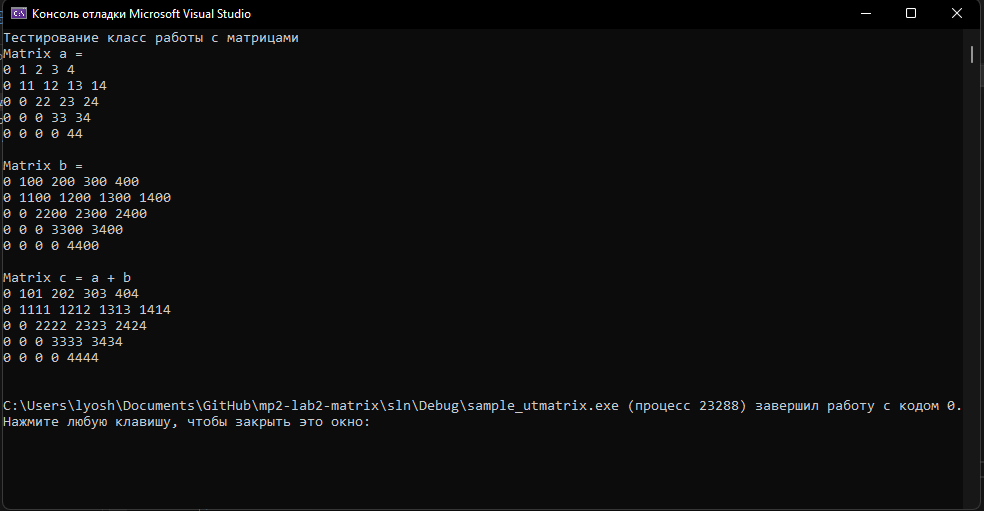


Рис.2 (Вывод при запуске sample\_matrix)

# Руководство программиста

## Описание структуры программы

Программа состоит из следующих модулей:

* Модуль *utmatirx,* содержащий реализацию классов Вектор и Матрица
* Модуль *sample\_utmatrix*. Пример использования матриц; демонстрирует результат сложения двух матриц*.*
* Модуль *gtest.* Отвечает за Google C++ Testing Framework.
* Модуль *test\_utmatrix.* Набор тестов для библиотек *test\_tbitfield.cpp* и *test\_tset.cpp. test\_main.cpp* запускает тесты. Реализованы они с помощью использования фреймворка Google Test.

## Описание структур данных

*Класс TDynamicVector:*

Поля:

size\_t sz – размер вектора

T \*pMem - память для представления битового поля

Конструкторы и деструктор:

TDynamicVector(size\_t size = 1) : sz(size); - конструктор по умолчанию

TDynamicVector(T\* arr, size\_t s) : sz(s));

TDynamicVector(const TDynamicVector& v); - конструктор копирования

TDynamicVector(TDynamicVector&& v); - конструктор перемещения

~TDynamicVector();

Методы:

size\_t size(); - размер вектора

Перегрузки:

bool operator==(const TDynamicVector &v) const; - сравнение на равенство

bool operator!=(const TDynamicVector &v) const; - сравнение на неравенство

TDynamicVector & operator=(const TDynamicVector &v); - присваивание

TDynamicVector& operator=(TDynamicVector&& v) noexcept – перемещение

T& operator[](size\_t ind) – индексация

const T& operator[](size\_t ind) const – индексация

T& at(size\_t ind) – индексация с контролем

const T& at(size\_t ind) const – индексация с контролем

friend void swap(TDynamicVector& lhs, TDynamicVector& rhs) – функция обмена

TDynamicVector operator+(T val) – скалярное сложение

TDynamicVector operator-(T val) – скалярное вычитание

TDynamicVector operator\*(T val) – скалярное умножение

TDynamicVector operator+( const TDynamicVector& v) – векторное сложение

TDynamicVector operator-( const TDynamicVector& v) – векторное вычитание

TDynamicVector operator\*( const TDynamicVector& v) – векторное умножение

friend istream &operator>>(istream &istr, TDynamicVector &v); - ввод с клавиатуры

friend ostream &operator<<(ostream &ostr, const TDynamicVector &bf); - вывод на экран

*Класс TDynamicMatrix: (унаследован от класса TDynamicVector)*

Поля:

TDynamicVector<TDynamicVector<T>>::sz – размер вектора (унаследовано от базового класса)

TDynamicVector<TDynamicVector<T>>::pMem - память для представления битового поля (унаследовано от базового класса)

Конструкторы:

TDynamicMatrix(size\_t s = 1) (стандартный конструктор)

Методы:

TDynamicVector<TDynamicVector<T>>::size - размер матрицы (унаследовано от базового класса);

Перегрузки:

using TDynamicVector<TDynamicVector<T>>::operator[]; - индексация (унаследовано от базового класса)

bool operator==(const TDynamicMatrix& m) const; - сравнение на равенство

bool operator!=(const TDynamicMatrix& m) const; - сравнение на неравенство

TDynamicMatrix operator\*(const T& val) – умножение матрицы на скаляр

TDynamicVector<T> operator\*(const TDynamicVector<T>& v) – умножение матрицы на вектор

TDynamicMatrix operator+(const TDynamicMatrix& m) – сложение матриц

TDynamicMatrix operator-(const TDynamicMatrix& m) – вычитание матриц

TDynamicMatrix operator\*(const TDynamicMatrix& m) – умножение матриц

TDynamicMatrix operator/(const TDynamicMatrix& m) – деление матриц

friend istream &operator>>(istream &istr, TDynamicMatrix& v); - ввод с клавиатуры

friend ostream &operator<<(ostream &ostr, const TDynamicMatrix& v); - вывод на экран

## Описание алгоритмов

В данной работе рассматриваются два типа операций: векторные и скалярные.

Что касается скалярных операций, то все они по своей сути идентичны: алгебраическая операция с определенным числом применяется к каждому элементу вектора (матрицы).

Относительно векторных (матричных) операций, сложение и вычитание векторов (матриц) возможно только при соответствии их размеров. При этом каждый i-тый элемент второго вектора (или i,j-тый элемент матрицы) складывается/вычитается из i-того элемента первого вектора (или i,j-того элемента матрицы).

Чтобы умножить две матрицы, необходимо, чтобы количество столбцов в левой матрице A было равно числу строк в правой матрице B. Результатом умножения матриц является матрица C, которая имеет столько же строк, сколько левая матрица, и столько же столбцов, сколько правая матрица. Для каждой ячейки новой матрицы (i, j) необходимо вычислить значение ячейки путем умножения i-той строки первой матрицы на j-тый столбец второй матрицы. Это означает поочередно умножить каждый элемент i-той строки на соответствующий элемент j-того столбца и сложить полученные произведения.

Умножение матрицы на вектор представляет собой особый случай умножения двух матриц, причем результатом является вектор.

# Эксперименты

Эксперименты проводились на ПК с следующими параметрами:

1. Операционная система: Windows 11
2. Процессор: Intel Core i5-7200, 3100 МГц, я: 6, лог.пр: 6
3. Версия Visual Studio: 2022, 17.5.2

|  |  |  |
| --- | --- | --- |
| Размер матрицы | Время работы оператора сложения (в млс)  O(n) | Время работы оператора умножения (в млс)  O(n3) |
| 10х10 | 0.0456 | 0.0981 |
| 100х100 | 0.7835 | 41.320 |
| 1000х1000 | 23.4626 | 80785.1244 |

Мы использовали классические методы для сложения и умножения квадратных матриц, сравнивая матрицы, число элементов которых различается на два порядка. Их сложности соответственно O(n) и O(n3) (под n подразумевается количество столбцов матрицы).

Мы можем наблюдать соответствие между вычисленной сложностью и реальной: увеличение числа столбцов матрицы в десять раз приводит к увеличению времени выполнения операции сложения в десять раз, а операции умножения – в тысячу раз, что соответствует 10^3.

# Заключение

В данной лабораторной работе мне удалось выполнить все поставленные задачи. Я создал вспомогательный класс TVector и расширил его до класса TMatrix, также реализовал автоматические тесты. Тестовое приложение подтвердило работоспособность реализованных классов. Все тесты были пройдены полностью и успешно. В результате экспериментов стало ясно, что сложность алгоритмов соответствует ожиданиям.

1. Литература
2. Гергель В.П. Методические материалы по курсу «Методы программирования 2»: [<http://www.itmm.unn.ru/files/2018/10/Primer-1.2.-Struktury-hraneniya-matrits-spetsialnogo-vida.pdf>], 2015.
3. Википедия: свободная электронная энциклопедия: на русском языке [https://ru.wikipedia.org/wiki/Матрица\_(математика)]
4. Приложение

## tmatrix.h

1. #ifndef \_\_TDynamicMatrix\_H\_\_
2. #define \_\_TDynamicMatrix\_H\_\_
3. #include <iostream>
4. using namespace std;
5. const int MAX\_VECTOR\_SIZE = 100000000;
6. const int MAX\_MATRIX\_SIZE = 10000;
7. // Динамический вектор -
8. // шаблонный вектор на динамической памяти
9. template<typename T>
10. class TDynamicVector
11. {
12. protected:
13. size\_t sz;
14. T\* pMem;
15. public:
16. TDynamicVector(size\_t size = 1) : sz(size)
17. {
18. if (sz == 0)
19. throw out\_of\_range("Vector size should be greater than zero");
20. if (sz > MAX\_VECTOR\_SIZE)
21. throw length\_error("Exceeded max vector size");
22. pMem = new T[sz]();// {}; // У типа T д.б. констуктор по умолчанию
23. }
24. TDynamicVector(T\* arr, size\_t s) : sz(s)
25. {
26. // assert(arr != nullptr && "TDynamicVector ctor requires non-nullptr arg");
27. if (sz > MAX\_VECTOR\_SIZE)
28. throw length\_error("Exceeded vector max size");
29. pMem = new T[sz];
30. std::copy(arr, arr + sz, pMem);
31. }
32. TDynamicVector(const TDynamicVector& v)
33. {
34. if (v.pMem == nullptr) {
35. sz = 0;
36. pMem = nullptr;
37. }
38. else {
39. sz = v.sz;
40. pMem = new T[sz];
41. std::copy(v.pMem, v.pMem + sz, pMem);
42. }
43. }
44. TDynamicVector(TDynamicVector&& v) noexcept
45. {
46. sz = v.sz;
47. pMem = nullptr;
48. std::swap(pMem, v.pMem);
49. }
50. ~TDynamicVector()
51. {
52. if (pMem != nullptr)
53. delete[] pMem;
54. pMem = nullptr;
55. sz = 0;
56. }
57. TDynamicVector& operator=(const TDynamicVector& v)
58. {
59. if (this == &v)
60. return \*this;
61. if (sz != v.sz)
62. {
63. if (pMem != nullptr)
64. delete[] pMem;
65. sz = v.sz;
66. pMem = new T[sz];
67. }
68. std::copy(v.pMem, v.pMem + v.sz, pMem);
69. return \*this;
70. }
71. TDynamicVector& operator=(TDynamicVector&& v) noexcept
72. {
73. if (this == &v)
74. return \*this;
75. if (pMem != nullptr) {
76. delete[] pMem;
77. pMem = nullptr;
78. }
79. sz = v.sz;
80. std::swap(pMem, v.pMem);
81. return \*this;
82. }
83. size\_t size() const noexcept { return sz; }
84. // индексация
85. T& operator[](size\_t ind)
86. {
87. return pMem[ind];
88. }
89. const T& operator[](size\_t ind) const
90. {
91. return pMem[ind];
92. }
93. // индексация с контролем
94. T& at(size\_t ind)
95. {
96. if (ind > sz || ind < 0) throw "Index not in range";
97. return pMem[ind];
98. }
99. const T& at(size\_t ind) const
100. {
101. if (ind > sz || ind < 0) throw "Index not in range";
102. return pMem[ind];
103. }
104. // сравнение
105. bool operator==(const TDynamicVector& v) const noexcept
106. {
107. if (sz != v.sz) return false;
108. for (int i = 0; i < sz; i++) {
109. if (pMem[i] != v.pMem[i]) return false;
110. }
111. return true;
112. }
113. bool operator!=(const TDynamicVector& v) const noexcept
114. {
115. return !(\*this == v);
116. }
117. // скалярные операции
118. TDynamicVector operator+(T val)
119. {
120. TDynamicVector result(sz);
121. for (int i = 0; i < sz; i++) {
122. result.pMem[i] = pMem[i] + val;
123. }
124. return result;
125. }
126. TDynamicVector operator-(T val)
127. {
128. TDynamicVector result(sz);
129. for (int i = 0; i < sz; i++) {
130. result.pMem[i] = pMem[i] - val;
131. }
132. return result;
133. }
134. TDynamicVector operator\*(T val)
135. {
136. TDynamicVector result(sz);
137. for (int i = 0; i < sz; i++) {
138. result.pMem[i] = pMem[i] \* val;
139. }
140. return result;
141. }
142. // векторные операции
143. TDynamicVector operator+(const TDynamicVector& v)
144. {
145. if (sz != v.sz) throw "Error len vector";
146. TDynamicVector result(sz);
147. for (int i = 0; i < sz; i++) {
148. result.pMem[i] = pMem[i] + v.pMem[i];
149. }
150. return result;
151. }
152. TDynamicVector operator-(const TDynamicVector& v)
153. {
154. if (sz != v.sz) throw "Error len vector";
155. TDynamicVector result(sz);
156. for (int i = 0; i < sz; i++) {
157. result.pMem[i] = pMem[i] - v.pMem[i];
158. }
159. return result;
160. }
161. T operator\*(const TDynamicVector& v) noexcept(noexcept(T()))
162. {
163. if (sz != v.sz) throw "Error len vector";
164. T result = T();
165. for (int i = 0; i < sz; i++) {
166. result += pMem[i] \* v.pMem[i];
167. }
168. return result;
169. }
170. friend void swap(TDynamicVector& lhs, TDynamicVector& rhs) noexcept
171. {
172. std::swap(lhs.sz, rhs.sz);
173. std::swap(lhs.pMem, rhs.pMem);
174. }
175. // ввод/вывод
176. friend istream& operator>>(istream& istr, TDynamicVector& v)
177. {
178. for (size\_t i = 0; i < v.sz; i++)
179. istr >> v.pMem[i]; // требуется оператор>> для типа T
180. return istr;
181. }
182. friend ostream& operator<<(ostream& ostr, const TDynamicVector& v)
183. {
184. for (size\_t i = 0; i < v.sz; i++)
185. ostr << v.pMem[i] << ' '; // требуется оператор<< для типа T
186. return ostr;
187. }
188. };
189. // Динамическая матрица -
190. // шаблонная матрица на динамической памяти
191. template<typename T>
192. class TDynamicMatrix : public TDynamicVector<TDynamicVector<T>>
193. {
194. using TDynamicVector<TDynamicVector<T>>::pMem;
195. using TDynamicVector<TDynamicVector<T>>::sz;
196. public:
197. TDynamicMatrix(size\_t s = 1) : TDynamicVector<TDynamicVector<T>>(s)
198. {
199. if (sz > MAX\_MATRIX\_SIZE)
200. throw length\_error("Exceeded matrix max size");
201. for (size\_t i = 0; i < sz; i++)
202. pMem[i] = TDynamicVector<T>(sz);
203. }
204. TDynamicMatrix(T\* arr, size\_t s) : sz(s)
205. {
206. if (sz > MAX\_MATRIX\_SIZE)
207. throw length\_error("Exceeded matrix max size");
208. assert(arr != nullptr && "TDynamicMatrix ctor requires non-nullptr arg");
209. pMem = new T[sz];
210. std::copy(arr, arr + sz, pMem);
211. }
212. using TDynamicVector<TDynamicVector<T>>::operator[];
213. // сравнение
214. bool operator==(const TDynamicMatrix& m) const noexcept
215. {
216. if (sz != m.sz) return false;
217. for (int i = 0; i < sz; i++) {
218. if (pMem[i] != m.pMem[i]) return false;
219. }
220. return true;
221. }
222. // матрично-скалярные операции
223. TDynamicMatrix operator\*(const T& val)
224. {
225. TDynamicMatrix result(sz);
226. for (int i = 0; i < size(this); i++) {
227. result.pMem[i] = pMem[i] \* val;
228. }
229. return result;
230. }
231. // матрично-векторные операции
232. TDynamicVector<T> operator\*(const TDynamicVector<T>& v)
233. {
234. if (sz != v.sz) throw "Matrices sizes are not equal";
235. TDynamicVector<T> result(sz);
236. for (int i = 0; i < sz; i++) {
237. for (int j = 0; j < sz; j++) {
238. result[i] += pMem[i][j] \* v[j];
239. }
240. }
241. return result;
242. }
243. // матрично-матричные операции
244. TDynamicMatrix operator+(const TDynamicMatrix& m)
245. {
246. if (sz != m.sz) throw "Matrices sizes are not equal";
247. TDynamicMatrix result(sz);
248. for (int i = 0; i < sz; i++) {
249. result.pMem[i] = pMem[i] + m.pMem[i];
250. }
251. return result;
252. }
253. TDynamicMatrix operator-(const TDynamicMatrix& m)
254. {
255. if (sz != m.sz) throw "Matrices sizes are not equal";
256. TDynamicMatrix result(sz);
257. for (int i = 0; i < sz; i++) {
258. result.pMem[i] = pMem[i] - m.pMem[i];
259. }
260. return result;
261. }
262. TDynamicMatrix operator\*(const TDynamicMatrix& m)
263. {
264. if (sz != m.sz) throw "Matrices sizes are not equal";
265. TDynamicMatrix result(sz);
266. for (int i = 0; i < sz; i++) {
267. for (int j = 0; j < sz; j++) {
268. for (int k = 0; k < sz; k++) {
269. result[i][j] += pMem[i][k] \* m.pMem[k][j];
270. }
271. }
272. }
273. return result;
274. }
275. // ввод/вывод
276. friend istream& operator>>(istream& istr, TDynamicMatrix& v)
277. {
278. for (int i = 0; i < v.sz; i++) {
279. istr >> v.pMem[i];
280. }
281. return istr;
282. }
283. friend ostream& operator<<(ostream& ostr, const TDynamicMatrix& v)
284. {
285. for (int i = 0; i < v.sz; i++) {
286. ostr << v.pMem[i] << endl;
287. }
288. return ostr;
289. }
290. };
291. #endif

## test\_tmatrix.cpp

1. #include "tmatrix.h"
2. #include <gtest.h>
3. TEST(TDynamicMatrix, can\_create\_matrix\_with\_positive\_length)
4. {
5. ASSERT\_NO\_THROW(TDynamicMatrix<int> m(5));
6. }
7. TEST(TDynamicMatrix, cant\_create\_too\_large\_matrix)
8. {
9. ASSERT\_ANY\_THROW(TDynamicMatrix<int> m(MAX\_MATRIX\_SIZE + 10));
10. }
11. TEST(TDynamicMatrix, throws\_when\_create\_matrix\_with\_negative\_length)
12. {
13. ASSERT\_ANY\_THROW(TDynamicMatrix<int> m(-10));
14. }
15. TEST(TDynamicMatrix, can\_create\_copied\_matrix)
16. {
17. TDynamicMatrix<int> m(5);
18. ASSERT\_NO\_THROW(TDynamicMatrix<int> m1(m));
19. }
20. TEST(TDynamicMatrix, copied\_matrix\_is\_equal\_to\_source\_one)
21. {
22. TDynamicMatrix<int> m1(2);
23. for (size\_t i = 0; i < 2; i++)
24. for (size\_t j = 0; j < 2; j++)
25. m1[i][j] = i + j;
26. TDynamicMatrix<int> m2(m1);
27. EXPECT\_EQ(m1, m2);
28. }
29. TEST(TDynamicMatrix, copied\_matrix\_has\_its\_own\_memory)
30. {
31. TDynamicMatrix<int> m1(2);
32. for (size\_t i = 0; i < 2; i++)
33. for (size\_t j = 0; j < 2; j++)
34. m1[i][j] = i + j;
35. TDynamicMatrix<int> m2(std::move(m1));
36. EXPECT\_NE(&m1, &m2);
37. }
38. TEST(TDynamicMatrix, can\_get\_size)
39. {
40. TDynamicMatrix<int> m(5);
41. EXPECT\_EQ(5, m.size());
42. }
43. TEST(TDynamicMatrix, can\_set\_and\_get\_element)
44. {
45. TDynamicMatrix<int> m(4);
46. ASSERT\_NO\_THROW(m[0][0] = 10);
47. }
48. TEST(TDynamicMatrix, can\_assign\_matrix\_to\_itself)
49. {
50. TDynamicMatrix<int> m(4);
51. ASSERT\_NO\_THROW(m = m);
52. EXPECT\_EQ(m, m = m);
53. }
54. TEST(TDynamicMatrix, can\_assign\_matrices\_of\_equal\_size)
55. {
56. TDynamicMatrix<int> m(4), m1(4);
57. for (size\_t i = 0; i < 4; i++)
58. for (size\_t j = 0; j < 4; j++)
59. m[i][j] = i;
60. ASSERT\_NO\_THROW(m1 = m);
61. EXPECT\_EQ(m, m1);
62. }
63. TEST(TDynamicMatrix, assign\_operator\_change\_matrix\_size)
64. {
65. TDynamicMatrix<int> m(2), m1(4);
66. m = m1;
67. EXPECT\_EQ(m1, m.size());
68. }
69. TEST(TDynamicMatrix, can\_assign\_matrices\_of\_different\_size)
70. {
71. TDynamicMatrix<int> m(4), m1(2);
72. for (size\_t i = 0; i < 4; i++)
73. for (size\_t j = 0; j < 4; j++)
74. m[i][j] = i;
75. ASSERT\_NO\_THROW(m1 = m);
76. EXPECT\_EQ(m, m1);
77. }
78. TEST(TDynamicMatrix, compare\_equal\_matrices\_return\_true)
79. {
80. TDynamicMatrix<int> m(2), m1(2);
81. EXPECT\_EQ(true, m == m1);
82. }
83. TEST(TDynamicMatrix, compare\_matrix\_with\_itself\_return\_true)
84. {
85. TDynamicMatrix<int> m1(2);
86. EXPECT\_EQ(true, m1 == m1);
87. }
88. TEST(TDynamicMatrix, matrices\_with\_different\_size\_are\_not\_equal)
89. {
90. TDynamicMatrix<int> m1(2), m2(4);
91. EXPECT\_EQ(false, m1 == m2);
92. }
93. TEST(TDynamicMatrix, can\_add\_matrices\_with\_equal\_size)
94. {
95. TDynamicMatrix<int> m(4), m1(4), m2(4);
96. for (size\_t i = 0; i < 4; i++)
97. for (size\_t j = 0; j < 4; j++) {
98. m[i][j] = i;
99. m1[i][j] = j;
100. m2[i][j] = i + j;
101. }
102. EXPECT\_EQ(m2, m + m1);
103. }
104. TEST(TDynamicMatrix, cant\_add\_matrices\_with\_not\_equal\_size)
105. {
106. TDynamicMatrix<int> m(4), m1(2);
107. ASSERT\_ANY\_THROW(m + m1);
108. }
109. TEST(TDynamicMatrix, can\_subtract\_matrices\_with\_equal\_size)
110. {
111. TDynamicMatrix<int> m(4), m1(4), m2(4);
112. for (size\_t i = 0; i < 4; i++) {
113. for (size\_t j = 0; j < 4; j++) {
114. m[i][j] = i;
115. m1[i][j] = j;
116. m2[i][j] = i - j;
117. }
118. }
119. EXPECT\_EQ(m2, m - m1);
120. }
121. TEST(TDynamicMatrix, cant\_subtract\_matrixes\_with\_not\_equal\_size)
122. {
123. TDynamicMatrix<int> m(4), m1(2);
124. ASSERT\_ANY\_THROW(m - m1);
125. }

## 8.3 test\_tvector.cpp

#include "tmatrix.h"

#include <gtest.h>

TEST(TDynamicVector, can\_create\_vector\_with\_positive\_length)

{

ASSERT\_NO\_THROW(TDynamicVector<int> v(5));

}

TEST(TDynamicVector, cant\_create\_too\_large\_vector)

{

ASSERT\_ANY\_THROW(TDynamicVector<int> v(MAX\_VECTOR\_SIZE + 1));

}

TEST(TDynamicVector, throws\_when\_create\_vector\_with\_negative\_length)

{

ASSERT\_ANY\_THROW(TDynamicVector<int> v(-5));

}

TEST(TDynamicVector, can\_create\_copied\_vector)

{

TDynamicVector<int> v(10);

ASSERT\_NO\_THROW(TDynamicVector<int> v1(v));

}

TEST(TDynamicVector, copied\_vector\_is\_equal\_to\_source\_one)

{

TDynamicVector<int> v(4);

ASSERT\_NO\_THROW(TDynamicVector<int> g(v));

}

TEST(TDynamicVector, can\_get\_size)

{

TDynamicVector<int> v(4);

EXPECT\_EQ(4, v.size());

}

//TEST(TDynamicVector, can\_set\_and\_get\_element)

//{

// TDynamicVector<int> v(4);

// v[0] = 4;

//

// EXPECT\_EQ(4, v[0]);

//}

TEST(TDynamicVector, throws\_when\_set\_element\_with\_negative\_index)

{

TDynamicVector<int> v(6);

ASSERT\_ANY\_THROW(v.at(-5));

}

TEST(TDynamicVector, throws\_when\_set\_element\_with\_too\_large\_index)

{

TDynamicVector<int> v(4);

ASSERT\_ANY\_THROW(v.at(99));

}

TEST(TDynamicVector, throws\_when\_sum\_different\_size)

{

TDynamicVector<int> v(4);

TDynamicVector<int> g(10);

ASSERT\_ANY\_THROW(v + g);

}

TEST(TDynamicVector, throws\_when\_minus\_different\_size)

{

TDynamicVector<int> v(4);

TDynamicVector<int> g(10);

ASSERT\_ANY\_THROW(v - g);

}

TEST(TDynamicVector, copied\_vector\_has\_its\_own\_memory)

{

int\* arr1 = new int[3] { 1, 2, 3 };

TDynamicVector<int> v1(arr1, 3), v3(std::move(v1));

EXPECT\_NE(&v1, &v3);

}

TEST(TDynamicVector, compare\_equal\_vectors\_return\_true)

{

int\* arr1 = new int[3] { 1, 2, 3 };

int\* arr3 = new int[3] { 1, 2, 3 };

TDynamicVector<int> v1(arr1, 3), v3(arr3, 3);

EXPECT\_EQ(v1, v3);

delete[] arr1;

delete[] arr3;

}

TEST(TDynamicVector, can\_multiply\_scalar\_by\_vector)

{

int\* arr1 = new int[3] { 1, 2, 3 };

int\* arr3 = new int[3] { 3, 6, 9 };

TDynamicVector<int> v1(arr1, 3), v3(arr3, 3);

EXPECT\_EQ(v3, v1 \* 3);

delete[] arr1;

delete[] arr3;

}