ЛАБОРАТОРНАЯ РАБОТА №1

ВНИМАНИЕ GIT!

Задание ОБЯЗАТЕЛЬНО должно выполняться под системой контроля версий git

Принцип работы тот же, что и на семинарах, а именно:

- Необходимо завести **публичный** удалённый репозиторий на сервере МИЭМ: https://git.miem.hse.ru/
- Синхронизировать локальный репозиторий с удалённым и настроить конфигурацию так, чтобы:
  - **user.name** = ваши **Фамилия** и **Имя** (на английском с заглавных букв)
  - **user.email** = ваша **корпоративная** почта
- Наличие истории коммитов, **КАЖДЫЙ** из которых:
  - является работоспособной версией программы
  - имеет однострочное сообщение, наглядно передающее суть данного небольшого изменения
  - вы являетесь как его автором, так и коммитером (т.е. в обоих случаях указаны ваши **Фамилия Имя** и **корпоративная почта**)

  (Если в истории всего несколько коммитов, то условие считается невыполненным)
- В Smart LMS нужно загрузить ссылку на удалённый репозиторий с вашей работой (это можно сделать в самом начале работы над лабораторной, чтобы потом не было проблем с дедлайном)

Невыполнение любого из этих условий приводит к тому, что ЛР вовсе не проверяется!

ОЦЕНИВАНИЕ

- За каждый из пунктов можно получить максимум 1 балл, т.е. за полностью выполненное задание можно получить максимум 8 баллов.
- Дополнительные баллы на оценку 9/10 можно получить при выполнении чего-то сверх требуемого в задании.
- Наличие правильно работающего кода без способности объяснить его расценивается как невыполненное задание т.е. 0 баллов.

СПИСЫВАНИЕ

Приводит к **ОБНУЛЕНИЮ** накопленной…

Поэтому лучше сделать меньше, но самостоятельно.

ЗАДАНИЕ

В пространстве имён **linalg** разработать библиотеку линейной алгебры, позволяющий работать с матрицами произвольной размерности. Для этого вам потребуется:

1) Реализовать класс Matrix на основе **ОДНОГО** динамического массива.
   Матрица хранит в себе вещественные числа (т.е. типа double).
   В качестве ресурсов класса хранить только:
   - **m_ptr** – указатель на этот массив;
   - **m_rows** и **m_columns** – актуальная размерность матрицы;

   Предоставить методы для получения информации о состоянии матрицы:
   - **….rows()** возвращает количество строк т.е. **m_rows**
   - **….columns()** возвращает количество колонок т.е. **m_columns**
   - **….empty()** возвращает значение типа bool, т.е. пустая ли матрица

   Предоставить методы для изменения размерности матрицы:
   - **….reshape**(rows, cols) меняет размерность матрицы, не меняя элементы и их количество. Если сохранить количество элементов не получится, выбрасывает исключение.

2) Предоставить пользователю следующие возможности при инициализации:

```
     // Дефолтный конструктор:
linalg::Matrix m0;
     // Конструкторы с параметрами:
// При заполнении элементов использовать дефолтные конструкторы:
linalg::Matrix m1(4);      // матрица вида: 4 строки и 1 столбец
linalg::Matrix m2(4, 6);   // матрица вида: 4 строки и 6 столбцов
     // Конструктор копирования:
linalg::Matrix m3(m1);
     // Конструктор перемещения:
linalg::Matrix m4(std::move(m2));
     // Унифицированная инициализация (использовать std::initializer_list)
linalg::Matrix m5 = { {1, 2, 3}, {4, 5, 6} };    // 2 строки и 3 столбца
linalg::Matrix m6 = { {1, 2, 3, 4, 5, 6} };      // 1 строка и 6 столбцов
linalg::Matrix m7 = { 1, 2, 3, 4, 5, 6 };        // 6 строк  и 1 столбец
linalg::Matrix m8 = { {1},{2},{3},{4},{5},{6} }; // 6 строк  и 1 столбец
```

3) Предоставить пользователю возможность взаимодействия с классом с помощью операторов присваивания и оператора вызова функции для индексирования:
   - Оператор присваивания (копирующий и перемещающий):

```
m1 = m2;  // Копирующее присваивание
m1 = linalg::Matrix{ 1, 2, 3, 4, 5, 6 }; // Перемещающее присваивание
```

   - Оператор вызова функции, чтобы обращаться к элементам (с возможностью их изменения, если это не константная матрица):

```
linalg::Matrix m = { {1.0, 2.0, 3.0}, {4.0, 5.0, 6.0} };
double val = m(0,2); // => 3.0 т.к. это 1-ая строка, 3-ий элемент
m(0,2) = 7.0; // теперь 1-ая строка, 3-ий элемент стал равен 7
```

```
const linalg::Matrix c_m = { {1.0, 2.0, 3.0}, {4.0, 5.0, 6.0} };
double val = c_m(0,2); // => 3.0 т.к. это 1-ая строка, 3-ий элемент
c_m(0,2) = 7.0; // ОШИБКА т.к. нельзя менять константную матрицу
```

4) Для удобства демонстрации результатов тестирования матрицы реализовать перегрузку оператора вывода в поток. Примеры формата вывода:

```
|1  2  3  4  5|
|2  4  6  8 10|
|3  6  9 12 15|
|4  8 12 16 20|
|5 10 15 20 25|
```

```
| 1 333   1   1 333|
|22   1   1 333 333|
| 1   1   1 333   1|
| 1  22   1  22  22|
```

Примечание: элементы должны быть выравнены по столбцам

5) Использовать механизм исключений для обработки нештатных ситуаций (например: размеры матриц не позволяют выполнить арифметическую операцию). Для этого использовать **std**::runtime_error. Также не забывать о спецификаторе noexcept – где это уместно.

6) Предоставить пользователю возможность работы с матрицами с помощью операторов:

Слева	Оператор	Справа	Описание	Результат
Matrix	+	Matrix	Поэлементное сложение матриц	rvalue
Matrix	+=	Matrix	Поэлементное сложение матриц	lvalue
Matrix	-	Matrix	Поэлементное вычитание матриц	rvalue
Matrix	-=	Matrix	Поэлементное вычитание матриц	lvalue
Matrix	*	Matrix	Перемножение **совместимых** матриц	rvalue
double	*	Matrix	Поэлементное перемножение на число	rvalue
Matrix	*	double	Поэлементное перемножение на число	rvalue
Matrix	*=	Matrix	Перемножение **совместимых** матриц	lvalue
Matrix	*=	double	Поэлементное перемножение на число	lvalue
Matrix	==	Matrix	Проверка матриц на совпадение	bool
Matrix	!=	Matrix	Проверка матриц на НЕ совпадение	bool

7) Расширьте возможности вашей матрицы с помощью следующих методов:

Описание:	Пример использования:
Норма (Фробениуса)	`double result = matr.norm()`
След	`double result = matr.trace()`
Определитель	`double result = matr.det()`
Прямой ход метода Гаусса	`std::cout << matr.gauss_forward()`
Обратный ход метода Гаусса	`std::cout << matr.gauss_backward()`
Ранг	`int result = matr.rank()`

Примечание: прямой и обратный метод Гаусса меняют матрицу, у которой они вызваны и возвращают её (а не копию, т.е. аналогично составным присваиваниям).

8) Расширьте возможности вашей матрицы с помощью следующих функций:

Описание:	Пример использования:
Соединить правую и левую	`Matrix result = concatenate(matr1, matr2)`
Транспонирование	`Matrix result = transpose(matr)`
Обратная матрица	`Matrix result = invert(matr)`
Возведение в степень	`Matrix result = power(matr, 4)`
Решение системы уравнений вида $A * \overline{x} = \overline{f}$ (методом Гаусса)	`Matrix result = solve(matr_A, vec_f);`

Примечание: каждая из предложенных функций не меняет матрицы из аргументов, а создаёт новые, которые в итоге возвращает в качестве результата.

Примечания и рекомендации:

1) Можно использовать другой стиль, например **rows_**, **columns_**, …
2) В рамках лабораторной нужно организовать **нестрогую** гарантию безопасности, т.е. если что-то пошло не так (вылетело исключение), то инвариант матрицы не должен быть нарушен (но сама матрица измениться может).
3) Не забывайте про тестирование. С помощью него вы сами сможете обнаружить свои ошибки ещё до того, как начнёте сдавать работу. Например, убедитесь в том, что:
   a. $A^0 = E$
   b. $(A^T)^{-1} = (A^{-1})^T$
   c. $(A_1 * A_2)^{-1} = A_2^{-1} * A_1^{-1}$
   d. $(A^{-1})^6 = (A^{-2})^3$
4) Будьте внимательны при сравнении вещественных чисел (`0.3*3 != 0.9`)
5) Важно уделить внимание архитектуре кода (за плохую архитектуру оценка будет снижаться):
   - Не забывайте, что наш мозг может комфортно удерживать внимание на какой-то сущности, только если она «влезает полностью на экран». Поэтому не нужно писать реализацию методов прямо в классе. (допускается исключение из правила, если реализация уж слишком маленькая, например, занимает всего одну строку). Если разрабатываемая вами процедура не влезает на экран, значит в 99% случаев её можно либо упростить, либо разбить на составные части, которые организованы в виде отдельных функций/методов.
   - При распределении вашего кода по файлам используйте идею наименьших включений.
   - Предложение по разбиению на файлы:
     - **matrix.h** – класс матрицы и интерфейсы функций
     - **matrix.cpp** – реализации методов матрицы и функций
     - **test.h** и **test.cpp** – тесты
     - **main.cpp** – точка входа в программу (запуск тестов)

     Помните, что в **main.cpp** должно попасть только то, что и правда достойно туда попасть.

Предложения для получения дополнительных баллов (это лишь некоторые идеи):

1) Добавить ещё возможностей в вашу библиотеку, например: унарный минус, построение минора матрицы и т.д.
2) Организовать **строгую** гарантию безопасности: если что-то пошло не так (вылетело исключение), то матрица должна остаться в её исходном состоянии (т.е. в том, в котором она была до начала операции). При чем это не значит, что постоянно нужно работать с копией матрицы…
3) Реализуйте тестирование с помощью сторонних средств, например: **CTest**, **Google Test**, **Native Unit Tests** (VS) и т.д.
4) Сделать матрицу шаблонной (т.е. тип элементов матрицы будет задаваться на этапе компиляции).
5) Организовать работу с памятью, используя вместимость, аналогично тому как это сделано в **STL** у `std::vector` (т.е. не нужно перевыделять память, для того, чтобы сохранить в себя матрицу меньшего размера, ведь памяти достаточно…)
6) Предоставить доступ к элементам матрицы с помощью оператора индексирования (для этого можно использовать прокси-класс, аналогично тому как это сделано в **STL** для специализации `std::vector<bool>`):

```
m[0][1] // элемент из 1-й строки 2-го столбца
```