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)

ВНИМАНИЕ CMAKE!

Использовать автоматизированную систему сборки cmake для организации проекта.

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 для данной ЛР состоит из двух частей:

1) Доработать Matrix из прошлой ЛР-1.
2) Разработать Graph для дальнейшей работы с ним в ЛР-3.

Оба программных решения необходимо оформить как header-only библиотеки в отдельных репозиториях => в SmartLMS необходимо будет загрузить обе ссылки.

ЗАДАНИЕ (часть 1)

1) Доработать класс Matrix и операции над ним из ЛР1 так, чтобы была возможность работать с произвольным типом T, а не только с double, т.е. сделать матрицу шаблонной с параметром шаблона – тип элемента матрицы.

2) Реализовать работу со вместимостью матрицы аналогично тому, как это сделано в контейнере std::vector из STL.
   Для этого в качестве ресурсов класса хранить:
   - **m_capacity** – количество элементов, под которые зарезервирована память.

   Также предоставить методы для работы со вместимостью:
   - ….**capacity**() – возвращает вместимость матрицы
   - ….**reserve**(*n*) – резервирует память под матрицу из *n* элементов
   - ….**shrink_to_fit**() – перевыделит память в соответствии с текущим размером матрицы: **m_rows * m_columns**
   - ….**clear**() – очищает матрицу т.е. меняет её актуальный размер (но вместимость не изменяется)

3) Обеспечить возможность работы с матрицами, хранящими в себе элементы несовпадающих типов.

```
     // int список инициализации в double матрицу
linalg::Matrix<double> m_d = { {1, 2}, {3, 4}, {5, 6} };
     // double список инициализации в int матрицу
linalg::Matrix<int> m_i = { {1.1, 2.2, 3.3}, {4.4, 5.5, 6.6} };
     // инициализация short матрицы с помощью double матрицы
linalg::Matrix<short> m_s = m_d;
     // присваивание int матрицы в short матрицу
m_s = m_i;
     // Арифметические операции между матрицами с разными типами:
m_s += m_i;
m_s -= m_i;
m_s *= 3.14;
m_s + m_i - m_s;
m_d * m_i;
m_d * 0;
3.14  * m_i;
     // Сравнение матриц с разными типами:
m_d == m_i;
m_d != m_i;
```

Примечание: заодно это решает следующую проблему (но почему?):

```
linalg::Matrix<double> m = { {1}, {2}, {3}, {4}, {5}, {6} };
// т.е. теперь нет неоднозначности, как это было обнаружено ранее в ЛР-1…
```

4) Обеспечить быструю и безопасную работу с матрицей, для этого:
   - Самостоятельный контроль запуска конструкторов и деструкторов при выделении и освобождении памяти под элементы матрицы. Для этого использовать размещающий new (…), чтобы запустить нужный нам конструктор (а не дефолтный, который автоматически запускается на втором этапе new[]). Это может понадобиться, например, в конструкторе копирования матрицы, когда нужно вызывать конструкторы копирования элементов типа T (вдруг у типа T вообще нет дефолтного конструктора…).
   - Инвариант матрицы нигде не должен нарушаться (например: если при инициализации матрицы какой-то из конструкторов типа T выбросит исключение, то это не должно привести к утечке памяти или несогласованности **m_rows**, **m_columns** и **m_capacity** с размером памяти, которая хранится по указателю **m_ptr**).

ЗАДАНИЕ (часть 2)

5) В пространстве имён **graph** разработать шаблонный класс Graph для представления ориентированного графа, который внутри себя хранит информацию:
   - Об уникальных ключах, по которым можно пройти в вершины (название вершины);
   - О данных, которые хранятся в вершине;
   - О направленных рёбрах, которые связывают вершины (с весом).

Внутреннее устройство класса Graph:
- три шаблонных параметра, определяющие типы: ключа, значения и веса
- для них введены псевдонимы: key_type, value_type и weight_type соответственно
- содержит вложенный класс узла Node
- в качестве ресурсов содержит unordered_map из пар ключей (key_type) и узлов (Node)

Внутреннее устройство класса Node:
- в качестве ресурсов содержит значение, хранимое в этом узле, и рёбра
- рёбра хранятся как unordered_map, состоящий из пар ключей (к какому узлу) и весов

6) Предоставить пользователю следующий интерфейс для работы с классом Node:

```
// Конструкторы: дефолтный, с параметром value_type, копирования и перемещения
// Операторы: копирующее и перемещающее присваивание
empty() // => bool пустой ли набор рёбер? (т.е. true если рёбер у этого узла нет)
size() // => size_t кол-во рёбер исходящих из этого узла
value()// => ссылка на хранимое в узле значение
     т.е. пользователь сможет менять его: node.value() = new_value;
     либо просто «подсматривать» если node константная
clear()// => ничего не возвращает. Удаляет все рёбра, исходящие из этого узла
// Итерирование по исходящим из этого узла рёбрам: begin(), end(), cbegin(), cend()
и вести псевдонимы: iterator, const_iterator
```

7) Предоставить пользователю следующий интерфейс для работы с классом Graph:

```
// Конструкторы: дефолтный, копирования и перемещения
// Операторы: копирующее и перемещающее присваивание
empty() // => bool пустой ли набор узлов?(т.е. true если узлов у этого графа нет)
size() // => size_t кол-во узлов имеется у этого графа
clear()// => ничего не возвращает. Удаляет все узлы (т.е. в результате граф пустой)
swap(…) // как метод класса (т.е. внутри) и глобальная реализация (т.е. вне класса)
// Итерирование по содержащимся в графе узлам: begin(), end(), cbegin(), cend()
и вести псевдонимы: iterator, const_iterator
// Работа с графом через ключ в аргументах:
[key]// => возвращает ссылку на значение узла (работает только для не const графов)
          (не нашёл key => создал новый Node с помощью дефолтного конструктора)
at(key) // => возвращает ссылку на значение узла (не нашёл key => кидает исключение)
degree_in(key); // => size_t степень входа т.е. кол-во рёбер входит в этот узел
degree_out(key); // => size_t степень выхода т.е. кол-во рёбер выходит из этого узла
loop(key) // => bool есть ли петля у узла с таким ключом?
// Использовать механизм исключений для обработки нештатных ситуаций (например:
по запрашиваемому ключу не найдена вершина). Для этого использовать класс из
стандартной библиотеки: std::runtime_error.
```

8) Предоставить пользователю интерфейс для вставки узлов в граф (аналогично тому, как это реализовано в map и unordered_map):

```
insert_node(key, val)// => вернёт пару: [Graph::iterator, bool]
insert_or_assign_node(key, val)// => вернёт пару: [Graph::iterator, bool]
// Далее: в 1ом арг принимают пару на ключи откуда и до куда нужно построить ребро:
// Если хотя бы один из ключей не валидный (т.е. не найден), то кидает исключение
// => вернёт пару: [Graph::Node::iterator, bool]
insert_edge({key_from, key_to}, weight)
insert_or_assign_edge({key_from, key_to}, weight)
```

Примечания и рекомендации:

1) Будьте внимательны с выражениями, в которых участвует тип `T` – его напишет пользователь вашей матрицы, а от него можно ждать что угодно: исключения могут вылететь из любого конструктора/оператора/метода элемента…
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) 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 в репозиториях:
     - **matrix.h** и **matrix.hpp** – репозиторий с матрицей
     - **graph.h** и **graph.hpp** – репозиторий с графом

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

1) 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
```

2) Работа с памятью с помощью аллокатора (второй шаблонный параметр)
3) Реализуйте тестирование с помощью сторонних средств, например: **CTest**, **Google Test**, **Native Unit Tests** (VS) и т.д.
4) При работе с памятью использовать стандартные алгоритмы на неинициализированной памяти из библиотеки STL
5) Специализация матрицы для элементов типа `bool` (аналогично тому, как это сделано у `std::vector`, т.е. элемент в памяти занимает 1 бит, а не 1 байт)
6) 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… Нужно просто проверять, является ли операция, которую планируем провести `noexcept`, и заниматься копированием только если это не так (аналогично тому, как это сделано в **STL**)
7) Расширить функционал графа (например, продумать и предоставить пользователю интерфейс для удаления узлов и рёбер из графа).
8) Отрисовать исследуемый граф.