ЛАБОРАТОРНАЯ РАБОТА №3

Допуск к защите лабораторной работы:

1. Создать документацию к ЛР с помощью Doxygen (комментарии + UML 2.0 диаграмма)

2. Загрузить документацию и сам код (h-файлы и cpp) в репозиторий GitLab не позднее 09 марта 23:59.

Разработать шаблонный класс Graph для представления ориентированного графа, который внутри себя хранит информацию:

1) Об уникальных ключах, по которым можно пройти в вершины (название вершины);
2) О данных, которые хранятся в вершине;
3) О направленных рёбрах, которые связывают вершины (с весом).

Задание 1 (1 балл):

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 (т.е. это композиция)
- в качестве ресурсов содержит map из пар ключей и узлов

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

Задание 2 (1 балл):

У класса Graph реализовать следующие конструкторы:
- дефолтный конструктор (создаёт пустой граф т.е. в нём нет ни рёбер, ни узлов)
- конструктор копирования (чтобы скопировать в себя другой граф)
- конструктор перемещения (чтобы перемещать в себя другой граф)

И операторы присваивания:
- оператор копирующего присваивания (скопировать другой граф, вместо старого)
- оператор перемещающего присваивания (подменить свои ресурсы на чужие)

Совет: для реализации понадобятся аналогичные конструкторы у Node

Задание 3 (1 балл):

Основной интерфейс Node, который необходимо предоставить пользователю:

```
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
```

Основной интерфейс Graph, который необходимо предоставить пользователю:
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(g) // как метод класса (т.е. внутри) и глобальная реализация (т.е. вне класса)

Задание 4 (1 балл):

Итерирование по Graph:
begin() **end**() **cbegin**() **cend**() // => для итерирования по узлам (т.е. по элементам map, которая хранится в ресурсах Graph)
Совет: для итерирования удобно ввести псевдонимы: iterator и const_iterator
Итерирование по Node:
begin() **end**() **cbegin**() **cend**() // => для итерирования по рёбрам (т.е. по элементам map, которая хранится в ресурсах Node)
Совет: для итерирования удобно ввести псевдонимы: iterator и const_iterator

Задание 5 (1 балл):

Работа с графом через ключ в аргументах:
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?
Совет: если key не найден среди узлов, то выкидывать исключение
[key]// => возвращает ссылку на значение узла (работает только для неконст граф)
(не нашёл key => создал новый Node с помощью дефолтного конструктора)
at(key) // => возвращает ссылку на значение узла (не нашёл key => кидает исключение)

Задание 6 (1 балл):

Вставка узлов и рёбер в граф (по аналогии как в map):
insert_node(key, val)// => вернёт пару: [iterator, bool]
insert_or_assign_node(key, val)// => вернёт пару: [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: [iterator, bool]
insert_edge({key_from, key_to}, weight)
insert_or_assign_edge({key_from, key_to}, weight)
Совет: при реализации удобно создать в private секции Node вспомогательные:
insert_edge(key, weight) // => вернёт пару: [iterator, bool]
insert_or_assign_edge(key, weight) // => вернёт пару: [iterator, bool]

Задание 7 (1 балл):

Удаление узлов и рёбер из Graph:
clear_edges()// => ничего не возвращает. Удаляет все рёбра в графе (останутся узлы)
erase_edges_go_from(key) // удалить все рёбра, выходящие из узла с ключом key
// => true если всё успешно => false если не нашлось узла с ключом key
erase_edges_go_to(key) // удалить все рёбра, входящие в узел с ключом key
// => true если всё успешно => false если не нашлось узла с ключом key
erase_node(key)// => bool успешное ли удаление узла, ключ у которого был key
// => true если узел был найден и удален => false если такого узла и не было
Совет: при удалении узла, не забудьте удалить рёбра идущие к нему из других узлов
Удаление рёбер из Node:
erase_edge(key)// => bool успешное ли удаление ребра, который исходит из этого узла и заканчивается в том, который называется key
//=> true если ребро было найдено и удалено => false если такого ребра и не было

Задание 8 (1 балл):

При выполнении следующих заданий использовать шаблонные классы Matrix и Graph.

Подключить разработанные библиотеки шаблонных классов в проект.

```
// Добавить пути в Visual Studio можно следующим образом:
Project -> Properties -> VC++ Directories -> Include Directories
// Теперь в проекте можно:
#include <Matrix_file.h>
#include <Graph.h>
```

Задание 9 (2 балла):

Алгоритм Дейкстры: возвращает пару из веса кратчайшего маршрута и сам маршрут (от key_from до key_to)

```
std::pair<weight_t, route_t> dijkstra(const graph_t& graph, node_name_t key_from, node_name_t key_to)
```

В случае нештатных ситуаций кидать исключения:
- матрица из аргументов не квадратная;
- не все веса рёбер в матрице положительные;
- какой-то из вершин из аргументов не найдено в графе;
- запрашиваемого пути не нашлось (т.е. эти вершины не связаны).

Подсказка к Заданию 7:

```cpp
struct Point { double x, y, z; };
std::ostream& operator << (std::ostream& out, Point p) {
	std::cout << '(' << p.x << ',' << p.y << ',' << p.z << ')';
	return out;
}
```

```cpp
template<typename Graph>
void print(const Graph& graph) {
	if (graph.empty()) {
		std::cout << "> This graph is empty!" << std::endl;
		return;
	}
	std::cout << "> Size of graph: " << graph.size() << std::endl;

	for (const auto& [key, node] : graph) {
		std::cout  << '[' << key << "] stores: " << node.value()
				<< " and matches with:" << std::endl;
		for (const auto& [key, weight] : node)
			std::cout  << "\t[" << key << "]\t with weight: "
					<< weight << std::endl;
	}
}
```

```cpp
Graph<std::string, Point, double> graph;

graph["zero"]; // Заполнится точкой, которая заполнится нулями

auto [it1, flag1] = graph.insert_node("first", {1, 1, 1});
std::cout << std::boolalpha << flag1 << std::endl; // => true

graph["second"]; // Заполнится точкой, которая заполнится нулями
auto [it2, flag2] = graph.insert_or_assign_node("second", {2, 2, 2}); //
перезаполнит
std::cout << std::boolalpha << flag2 << std::endl; // => false

graph["third"] = Point{ 3, 3, 3 };
auto [it3, flag3] = graph.insert_node("third", {1, 1, 1}); // бездействует
std::cout << std::boolalpha << flag3 << std::endl; // => false

graph["fourth"]; // Заполнится точкой, которая заполнится нулями
graph.at("fourth") = Point{ 4, 4, 4 };

try { graph.at("fifth"); }
catch (std::exception& ex) { std::cout << ex.what() << std::endl; }

auto [it4, flag4] = graph.insert_edge({ "first", "second" }, 44.44);
std::cout << std::boolalpha << flag4 << std::endl; // => true

auto [it5, flag5] = graph.insert_edge({ "first", "second" }, 55.55);
std::cout << std::boolalpha << flag5 << std::endl; // => false

auto [it6, flag6] = graph.insert_or_assign_edge({ "first", "second" }, 66.66);
std::cout << std::boolalpha << flag6 << std::endl; // => false

auto [it7, flag7] = graph.insert_or_assign_edge({ "second", "first" }, 77.77);
std::cout << std::boolalpha << flag7 << std::endl; // => true

print(graph);

auto graph_other = graph; // Конструктор копирования
auto graph_new = std::move(graph); // Конструктор перемещения
graph = std::move(graph_new); // Перемещающее присваивание
```

```
graph_new = graph; // Копирующее присваивание
graph.swap(graph_new); // Поменять местами содержимое графов
swap(graph, graph_new); // Поменять местами содержимое графов
print(graph);

for (auto& [key, node] : graph) {
     std::cout << "Is here no edges?" << std::boolalpha << node.empty() <<
std::endl;
     std::cout << "How many edges are going from here?" << node.size() <<
std::endl;
     node.value() = Point{ 1,2,3 }; // могу поменять вес
     for (auto& [key, weight] : node) {
          // key = "new key"; // ОШИБКА: нельзя менять ключ
          weight = 11.11; // могу задать новый вес у этого ребра
     }
     bool flag = node.erase_edge("first");      // => true, если удалил
                                               // => false, если такого ребра нет
}
print(graph);

bool flag8 = graph.erase_node("new name");
std::cout << std::boolalpha << flag8 << std::endl;
bool flag9 = graph.erase_node("first");
std::cout << std::boolalpha << flag9 << std::endl;
print(graph);

graph.insert_edge({ "second", "zero" }, 4.4);
graph.insert_edge({ "third", "second" }, 6.6);
print(graph);

bool flag10 = graph.erase_edges_go_from("new name");
std::cout << std::boolalpha << flag10 << std::endl;
bool flag11 = graph.erase_edges_go_from("second");
std::cout << std::boolalpha << flag11 << std::endl;
print(graph);

graph.erase_edges_go_to("second");
print(graph);

graph.insert_edge({ "second", "zero" }, 4.4);
graph.insert_edge({ "third", "second" }, 6.6);
graph.insert_edge({ "third", "third" }, 6.6);
graph.insert_edge({ "third", "zero" }, 6.6);
print(graph);

std::cout << graph.degree_in("second") << std::endl;
std::cout << graph.degree_in("third") << std::endl;
std::cout << graph.degree_out("second") << std::endl;
std::cout << graph.degree_out("third") << std::endl;
std::cout << std::boolalpha << graph.loop("second") << std::endl;
std::cout << std::boolalpha << graph.loop("third") << std::endl;
bool flag12 = graph.erase_node("new name");
std::cout << std::boolalpha << flag12 << std::endl;
bool flag13 = graph.erase_node("second");
std::cout << std::boolalpha << flag13 << std::endl;
print(graph);

graph.clear_edges(); // Очистить все рёбра
print(graph);
```

```
graph.clear(); // Очистить все вершины (очевидно, вместе с рёбрами)
print(graph);
```