Учреждение образования
«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ»

Кафедра интеллектуальных информационных технологий

Лабораторная работа № 4 по курсу

«Проектирование программ в интеллектуальных системах»

Выполнили студенты группы 921701: **Соловьев А.М.**

Проверил: Садовский М.Е

МИНСК

2020

Тема: Обобщенное программирование. Стандартная библиотека шаблонов (STL).

Цель работы: Получить навыки обобщенного программирования с использованием шаблонов.

При выборе этого варианта необходимо реализовать класс контейнера для представления неориентированного графа с использованием матрицы инцидентности. Класс не должен раскрывать способ представления графа, а должен предлагать typedef'ы методы и итераторы для работы.

Реализованный шаблон класс представления неориентированного графа должен соответствовать следующим требованиям (общие требования см. в общей части задания):

- проверка присутствия вершины в графе
- проверка присутствия ребра между вершинами в графе
- получение количества вершин
- получение количества ребер
- вычисление степени вершины
- вычисление степени ребра
- добавление вершины
- добавление ребра
- удаление вершины
- удаление ребра
- двунаправленный итератор для перебора вершин
- двунаправленный итератор для перебора ребер (совет: обратите внимание на класс std::pair)
- двунаправленный итератор для перебора ребер, инцидентных вершине
- двунаправленный итератор для перебора вершин, смежных вершине
- удаление вершины по итератору на вершину
- удаление ребра по итератору на ребро
- обратные (reverse) модификации для всех итераторов
- константные модификации для всех итераторов

Необходимость создания шаблонных классов-оберток для представления вершин и ребер на усмотрение разработчика. Дополнительная литература:

- Способы представления графов

Ход и результаты работы:

В программе представлен класс контейнер для неориентированного графа (способ представления – матрица инцидентности). Также в программе для более удобного восприятия реализованы классы Vertex и Edge.

```
class Edge {
public:
    pair<Vertex*, Vertex*> edge;
    Edge() { edge = { NULL,NULL }; }
    Edge(Vertex* first_vertex, Vertex* second_vertex) : edge(make_pair(first_vertex, second_vertex)) { }
    friend ostream& operator<<(ostream& out, Edge& pedge) {
        out << *pedge.edge.first << " " << *pedge.edge.second;
        return out;
    }
    friend ostream& operator<<(ostream& out, Edge* pedge) {
        out << *(pedge->edge.first) << " " << *(pedge->edge.second);
        return out;
    }
    friend ostream& operator<<(ostream& out, const Edge& pedge) {
        out << *pedge.edge.first << " " << *pedge.edge.second;
        return out;
    }
};

class Vertex {
public:
    string id;
    vector<bool> edge;
    vector<Vertex*> adjacent_vertex;
    vector <Edge> incidental_edge;
    Vertex() :id("") {}
    Vertex(string name_of_vertex) :id(name_of_vertex) {
```

Класс Ungraph имеет ряд методов, такие как добавление, удаление, нахождение степени и т.д.

```
    void add_vertex(string name_of_vertex);
    void delete_vertex(string name_of_vertex);

    void add_edge(string first_vertex, string second_vertex);
    void delete_edge(string first_vertex, string second_vertex);

    bool check_vertex(string name_of_vertex);
    bool check_edge(string first_vertex, string second_vertex);

    int numb_of_vertex() { return vertex.size() - 2; }
    int number_of_edge() { return edge.size() - 2; }

    int degree_of_vertex(string name_of_vertex);
    int degree_of_edge(string first_vertex, string seconde_vertex);
```

А также самой объёмной частью программы является реализация итераторов. Всего их 4 основных типа:

1. Итератор по вершинам графа
2. Итератор по рёбрам графа
3. Итератор по вершинам смежным данной вершине
4. Итератор по рёбрам инцидентным данной вершине

Класс итератора является шаблонным и итерируется при помощи указателя. Шаблонный класс даёт гибкость, при помощи которой мы можем итерироваться по любому типу данных. Это обстоятельство дало возможность использования модификаций const и reverse для каждого вида итераторов.

```
template<class Iter>
class Graph_reverse_iterator {
    friend class Ungraph;
    typedef Iter iterator_type;
    typedef input_iterator_tag iterator_category;
    typedef iterator_type value_type;
    typedef ptrdiff_t difference_type;
    typedef iterator_type* pointer;
    typedef iterator_type& reference;
    Iter* value;
public:
    Graph_reverse_iterator() { value = NULL; }
    Graph_reverse_iterator(Iter* p) :value(p) {}
```

```
typedef Graph_iterator<graph_sistem::Edge> Edge_iterator;
typedef Graph_iterator<const graph_sistem::Edge> Edge_citerator;
typedef Graph_reverse_iterator<graph_sistem::Edge> Edge_riterator;

typedef Graph_iterator<graph_sistem::Edge> Incidental_iterator;
typedef Graph_iterator<const graph_sistem::Edge> Incidental_citerator;
typedef Graph_reverse_iterator<graph_sistem::Edge> Incidental_riterator;
```

В самом классе итератора перегружен ряд операторов, среди которых есть обязательные для любого итератора: ++, --, ==, != и *.

```
bool operator==(Graph_reverse_iterator const& other) { return value == other.value; }
bool operator!=(Graph_reverse_iterator const& other) { return value != other.value; }
Graph_reverse_iterator& operator=(Graph_reverse_iterator const& other) { value = other.val

typename Graph_reverse_iterator::reference operator*() const { ... }

Graph_reverse_iterator& operator++() { ... }
Graph_reverse_iterator& operator--() { ... }

Graph_reverse_iterator& operator+(int number) { ... }
Graph_reverse_iterator& operator-(int number) { ... }

Graph_reverse_iterator& operator +=(int number) { ... }
Graph_reverse_iterator& operator -=(int number) { ... }
```

Чтобы контролировать выход за пределы контейнера в классе существуют методы begin() и end().

Begin() возвращает итератор на первый элемент контейнера, а end() на последний(теневой, стоящий после действительного последнего).

```
Vertex_iterator vertex_begin() {
    if (!vertex.size() == 2)throw  Access_error("Vector has no elements");
    return Vertex_iterator(&vertex[1]);
}
Vertex_iterator vertex_end() {
    if (!vertex.size() == 2)throw Access_error("Vector has no elements");
    return Vertex_iterator(&vertex[vertex.size() - 1]);
}

Vertex_citerator vertex_cbegin() {
    if (!vertex.size() == 2)throw Access_error("Vector has no elements");
    return Vertex_citerator(&vertex[1]);
}
Vertex_citerator vertex_cend() {
    if (!vertex.size() == 2)throw Access_error("Vector has no elements");
    return Vertex_citerator(&vertex[vertex.size() - 1]);
}

Vertex_riterator vertex_rbegin() {
    if (!vertex.size() == 2)throw Access_error("Vector has no elements");
    return Vertex_riterator(&vertex[vertex.size() - 2]);
}
```

Также для проверки алгоритмов в программу добавлено 22 юнит теста

```
public:

    TEST_METHOD(Adding_a_vertex)
    {
        Ungraph G;
        G.add_vertex("First_vertex");
        Assert::IsTrue(G.check_vertex("First_vertex"));
    }
    TEST_METHOD(Adding_a_edge)
    {
        Ungraph G;
        G.add_vertex("First_vertex");
        G.add_vertex("Second_vertex");
        G.add_edge("First_vertex", "Second_vertex");
        Assert::IsTrue(G.check_edge("First_vertex", "Second_vertex"));
    }
    TEST_METHOD(Delete_vertex)
    {
        Ungraph G;
        G.add_vertex("First_vertex");
        G.delete_vertex("First_vertex");
        Assert::IsTrue(!G.check_vertex("First_vertex"));
    }
```

Вывод

В ходе лабораторной работы я разбирался в принципе работы stl контейнеров. Данные контейнеры крайне удобны для работы в разных областях it-индустрии. Однако если разработчик делает свой контейнер, ему необходимо также реализовать методы, а также итераторы, позволяющие работать с данным контейнером.

В данной лабораторной работе я реализовал контейнер для неориентированного графа. Данной библиотекой можно пользоваться для решения задач с графами.