**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра МО ЭВМ**

отчет

**по лабораторной работе №1**

**по дисциплине «Программирование»**

Тема: Создание классов, конструкторов и методов классов

|  |  |  |
| --- | --- | --- |
| Студент гр. 0383 |  | Сабанов П.А. |
| Преподаватель |  | Жангиров Т.Р. |

Санкт-Петербург

2021

**Цель работы.**

Игровое поле представляет из себя прямоугольную плоскость разбитую на клетки. На поле на клетках в дальнейшем будут располагаться игрок, враги, элементы взаимодействия. Клетка может быть проходимой или непроходимой, в случае непроходимой клетки, на ней ничего не может располагаться. На поле должны быть две особые клетки: вход и выход. В дальнейшем игрок будет появляться на клетке входа, а затем выполнив определенный набор задач дойти до выхода.

Требования:

1) Реализовать класс поля, который хранит набор клеток в виде двумерного массива.

2) Реализовать класс клетки, которая хранит информацию о ее состоянии, а также того, что на ней находится.

3) Создать интерфейс элемента клетки (объект, который хранится на клетке).

4) Обеспечить появление клеток входа и выхода на поле. Данные клетки не должны быть появляться рядом.

5) Для класса поля реализовать конструкторы копирования и перемещения, а также соответствующие операторы.

6) Гарантировать отсутствие утечки памяти.

**Ход работы.**

Для реализации поля я решил создать класс matrix, который будет хранить матрицу однотипных объектов. Для его реализации я решил создать класс vector, который будет хранить список однотипных объектов. Для использования в будущем я создал класс vector\_iterator и добавил в класс vector методы, создающие нужный итератор.

Класс vector хранит в себе динамический массив. Имеется возможность добавлять и удалять по произвольному индексу элементы. Имеется возможность изменить размер и вместимость вектора. Вектор обладает свойством random access.

Класс matrix хранит в себе вектор векторов.

Класс field хранит в себе экземпляр класса matrix.

Для представления клетки был создан класс cell, хранящий идентификатор типа инт типа клетки и artifact, представляющий из себя артефакт, лежащий на этой клетке. Каждый артефакт имеет уникальный идентификатор. Артефакт реализует семантику перемещения для перемещения артефакт из одной клетки в другую или в рюкзак игрока, а также семантику копирования для создания копий артефакта (однако идентификатор у нового артефакта будет другим). Для удобства и эффективности реализации все данные артефакта хранятся в его поле \_data типа std::unique\_ptr<data>.

**Исходный код.**

<<vector.h>>

#ifndef OOP\_LAB1\_VECTOR\_H

#define OOP\_LAB1\_VECTOR\_H

#include <iostream>

#include <iterator>

#include "vector\_iterator.h"

template <typename T>

class vector {

public:

using iterator = vector\_iterator<T>;

using const\_iterator = vector\_iterator<const T>;

private:

static constexpr int initial\_capacity = 10;

int \_size = 0;

int \_capacity = 0;

T\* \_arr = nullptr;

void \_\_copy(const vector<T>& other);

void \_\_move(vector<T>&& other);

void \_\_free();

public:

explicit vector(int n = initial\_capacity);

vector(const std::initializer\_list<T>& initializerList);

vector(const vector<T>& other);

vector(vector<T>&& other);

~vector();

void resize(int n);

void reserve(int n);

void add(const T& elem);

void remove(int index);

int size() const;

vector<T>& operator=(const std::initializer\_list<T>& initializerList);

vector<T>& operator=(const vector<T>& other);

vector<T>& operator=(vector<T>&& other);

T& operator[](int index);

const T& operator[](int index) const;

template <typename R>

friend std::ostream& operator<<(std::ostream& out, const vector<R>& v);

iterator begin();

iterator end();

iterator rbegin();

iterator rend();

iterator from(int index);

iterator rfrom(int index);

const\_iterator begin() const;

const\_iterator end() const;

const\_iterator rbegin() const;

const\_iterator rend() const;

const\_iterator from(int index) const;

const\_iterator rfrom(int index) const;

};

template <typename T>

void vector<T>::\_\_copy(const vector<T>& other) {

\_size = other.\_size;

\_capacity = other.\_capacity;

\_arr = new T[\_capacity];

for (int i = 0; i < \_size; ++i)

\_arr[i] = other.\_arr[i];

}

template <typename T>

void vector<T>::\_\_move(vector<T>&& other) {

\_size = other.\_size;

\_capacity = other.\_capacity;

\_arr = other.\_arr;

other.\_size = 0;

other.\_capacity = 0;

other.\_arr = nullptr;

}

template <typename T>

void vector<T>::\_\_free() {

delete[] \_arr;

}

template <typename T>

vector<T>::vector(int n) : \_size(0), \_capacity(n) {

\_arr = new T[n];

}

template <typename T>

vector<T>::vector(const std::initializer\_list<T>& initializerList) {

operator=(initializerList);

}

template <typename T>

vector<T>::vector(const vector<T>& other) {

\_\_copy(other);

}

template <typename T>

vector<T>::vector(vector<T>&& other) {

\_\_move(std::move(other));

}

template <typename T>

vector<T>::~vector() {

\_\_free();

}

template <typename T>

void vector<T>::resize(int n) {

if (n > \_capacity)

reserve(n);

\_size = n;

}

template <typename T>

void vector<T>::reserve(int n) {

if (n > \_capacity) {

T\* new\_arr = new T[n];

for (int i = 0; i < \_size; ++i)

new\_arr[i] = \_arr[i];

delete[] \_arr;

\_capacity = n;

\_arr = new\_arr;

}

}

template <typename T>

void vector<T>::add(const T& elem) {

if (\_size + 1 > \_capacity)

reserve(\_capacity \* 2 + 1);

\_arr[\_size] = elem;

++\_size;

}

template <typename T>

void vector<T>::remove(int index) {

(&\_arr[index])->~T();

for (int i = index + 1; i < \_size; ++i)

\_arr[i-1] = \_arr[i];

--\_size;

}

template <typename T>

int vector<T>::size() const {

return \_size;

}

template <typename T>

vector<T>& vector<T>::operator=(const std::initializer\_list<T>& initializerList) {

\_\_free();

\_size = \_capacity = initializerList.size();

\_arr = new T[\_capacity];

auto iter = initializerList.begin();

for (int i = 0; i < \_size; ++i, ++iter)

\_arr[i] = \*iter;

return \*this;

}

template <typename T>

vector<T>& vector<T>::operator=(const vector<T>& other) {

if (this != &other) {

\_\_free();

\_\_copy(other);

}

return \*this;

}

template <typename T>

vector<T>& vector<T>::operator=(vector<T>&& other) {

if (this != &other) {

\_\_free();

\_\_move(std::move(other));

}

return \*this;

}

template <typename T>

T& vector<T>::operator[](int index) {

return \_arr[index];

}

template <typename T>

const T& vector<T>::operator[](int index) const {

return \_arr[index];

}

template <typename R>

std::ostream& operator<<(std::ostream& out, const vector<R>& v) {

if (v.size() == 0) {

out << "{}";

return out;

}

out << "{ ";

for (int i = 0;;) {

out << v[i];

if (++i == v.size())

break;

out << ", ";

}

out << " }";

return out;

}

template <typename T>

typename vector<T>::iterator vector<T>::begin() {

return iterator(\_arr);

}

template <typename T>

typename vector<T>::iterator vector<T>::end() {

return iterator(\_arr + \_size);

}

template <typename T>

typename vector<T>::iterator vector<T>::rbegin() {

return iterator(\_arr + \_size - 1, true);

}

template <typename T>

typename vector<T>::iterator vector<T>::rend() {

return iterator(\_arr - 1, true);

}

template <typename T>

typename vector<T>::iterator vector<T>::from(int index) {

return iterator(\_arr + index);

}

template <typename T>

typename vector<T>::iterator vector<T>::rfrom(int index) {

return iterator(\_arr + index, true);

}

template <typename T>

typename vector<T>::const\_iterator vector<T>::begin() const {

return const\_iterator(\_arr);

}

template <typename T>

typename vector<T>::const\_iterator vector<T>::end() const {

return const\_iterator(\_arr + \_size);

}

template <typename T>

typename vector<T>::const\_iterator vector<T>::rbegin() const {

return const\_iterator(\_arr + \_size - 1, true);

}

template <typename T>

typename vector<T>::const\_iterator vector<T>::rend() const {

return const\_iterator(\_arr - 1, true);

}

template <typename T>

typename vector<T>::const\_iterator vector<T>::from(int index) const {

return const\_iterator(\_arr + index);

}

template <typename T>

typename vector<T>::const\_iterator vector<T>::rfrom(int index) const {

return const\_iterator(\_arr + index, true);

}

#endif //OOP\_LAB1\_VECTOR\_H

<<vector\_iterator.h>>

#ifndef OOP\_LAB1\_FINAL\_VECTOR\_ITERATOR\_H

#define OOP\_LAB1\_FINAL\_VECTOR\_ITERATOR\_H

#include "vector.h"

template <typename T>

class vector\_iterator : std::iterator<std::random\_access\_iterator\_tag, T> {

template <typename R>

friend class vector;

bool \_reverse;

T\* \_cur;

explicit vector\_iterator(T\* start, bool reverse = false);

public:

vector\_iterator<T>& reverse();

T& operator\*();

vector\_iterator<T>& operator++();

vector\_iterator<T> operator++(int);

vector\_iterator<T>& operator+=(int n);

vector\_iterator<T>& operator--();

vector\_iterator<T> operator--(int);

vector\_iterator<T>& operator-=(int n);

bool operator==(const vector\_iterator<T>& other) const;

bool operator!=(const vector\_iterator<T>& other) const;

};

template <typename T>

vector\_iterator<T>::vector\_iterator(T\* start, bool reverse) : \_cur(start), \_reverse(reverse) {}

template <typename T>

vector\_iterator<T>& vector\_iterator<T>::reverse() {

\_reverse = !\_reverse;

return \*this;

}

template <typename T>

T& vector\_iterator<T>::operator\*() {

return \*\_cur;

}

template <typename T>

vector\_iterator<T>& vector\_iterator<T>::operator++() {

if (\_reverse)

--\_cur;

else

++\_cur;

return \*this;

}

template <typename T>

vector\_iterator<T> vector\_iterator<T>::operator++(int) {

vector\_iterator<T> tmp(\_cur, \_reverse);

if (\_reverse)

--\_cur;

else

++\_cur;

return tmp;

}

template <typename T>

vector\_iterator<T>& vector\_iterator<T>::operator+=(int n) {

if (\_reverse)

\_cur -= n;

else

\_cur += n;

return \*this;

}

template <typename T>

vector\_iterator<T>& vector\_iterator<T>::operator--() {

if (\_reverse)

++\_cur;

else

--\_cur;

return \*this;

}

template <typename T>

vector\_iterator<T> vector\_iterator<T>::operator--(int) {

vector\_iterator<T> tmp(\_cur, \_reverse);

if (\_reverse)

++\_cur;

else

--\_cur;

return tmp;

}

template <typename T>

vector\_iterator<T>& vector\_iterator<T>::operator-=(int n) {

if (\_reverse)

\_cur += n;

else

\_cur -= n;

return \*this;

}

template <typename T>

bool vector\_iterator<T>::operator==(const vector\_iterator<T>& other) const {

return \_cur == other.\_cur && \_reverse == other.\_reverse;

}

template <typename T>

bool vector\_iterator<T>::operator!=(const vector\_iterator<T>& other) const {

return !operator==(other);

}

#endif //OOP\_LAB1\_FINAL\_VECTOR\_ITERATOR\_H

<<matrix.h>>

#ifndef OOP\_LAB1\_FINAL\_MATRIX\_H

#define OOP\_LAB1\_FINAL\_MATRIX\_H

#include <iostream>

#include "../vector/vector.h"

template <typename T>

class matrix {

public:

using column = vector<T>;

private:

int \_width = 0, \_height = 0;

vector<column> \_matr = nullptr;

void \_\_copy(const matrix<T>& other);

void \_\_move(matrix<T>&& other);

public:

matrix(int width, int height);

matrix(const std::initializer\_list<column>& initializerList);

matrix(const matrix<T>& other);

matrix(matrix<T>&& other);

int width() const;

int height() const;

column& operator[](int index);

const column& operator[](int index) const;

matrix<T>& operator=(const std::initializer\_list<vector<T>>& initializerList);

matrix<T>& operator=(const matrix<T>& other);

matrix<T>& operator=(matrix<T>&& other);

template <typename R>

friend std::ostream& operator<<(std::ostream& out, const matrix<R>& m);

};

template <typename T>

void matrix<T>::\_\_copy(const matrix<T>& other) {

\_width = other.\_width;

\_height = other.\_height;

\_matr = other.\_matr;

}

template <typename T>

void matrix<T>::\_\_move(matrix<T>&& other) {

\_width = other.\_width;

\_height = other.\_height;

\_matr = std::move(other.\_matr);

other.\_width = 0;

other.\_height = 0;

}

template <typename T>

matrix<T>::matrix(int width, int height) : \_width(width), \_height(height), \_matr(width) {

\_matr.resize(width);

for (column& c : \_matr)

c.resize(height);

}

template <typename T>

matrix<T>::matrix(const std::initializer\_list<column>& initializerList) {

operator=(initializerList);

}

template <typename T>

matrix<T>::matrix(const matrix<T>& other) {

\_\_copy(other);

}

template <typename T>

matrix<T>::matrix(matrix<T>&& other) {

\_\_move(std::move(other));

}

template <typename T>

int matrix<T>::width() const {

return \_width;

}

template <typename T>

int matrix<T>::height() const {

return \_height;

}

template <typename T>

typename matrix<T>::column& matrix<T>::operator[](int index) {

return \_matr[index];

}

template <typename T>

const typename matrix<T>::column& matrix<T>::operator[](int index) const {

return \_matr[index];

}

template <typename T>

matrix<T>& matrix<T>::operator=(const std::initializer\_list<vector<T>>& initializerList) {

if (initializerList.size() == 0) {

\_width = \_height = 0;

\_matr.resize(0);

} else {

\_width = initializerList.size();

\_height = (\*initializerList.begin()).size();

\_matr.resize(\_width);

auto iter = initializerList.begin();

for (int i = 0; i < \_width; ++i, ++iter)

\_matr[i] = std::move(\*iter);

}

}

template <typename T>

matrix<T>& matrix<T>::operator=(const matrix<T>& other) {

if (this != &other) {

\_\_copy(other);

}

return \*this;

}

template <typename T>

matrix<T>& matrix<T>::operator=(matrix<T>&& other) {

if (this != &other) {

\_\_move(std::move(other));

}

return \*this;

}

template <typename R>

std::ostream& operator<<(std::ostream& out, const matrix<R>& m) {

out << "{ width=" << m.\_width << ", height=" << m.\_height << ", elements=" << m.\_matr << " }";

return out;

}

#endif //OOP\_LAB1\_FINAL\_MATRIX\_H

<<pair.h>>

#ifndef OOP\_LAB1\_FINAL\_PAIR\_H

#define OOP\_LAB1\_FINAL\_PAIR\_H

#include <utility>

#include <string.h>

template <typename T, typename R = T>

class pair {

void \_\_copy(const pair<T,R>& other);

void \_\_move(pair<T,R>&& other);

public:

T first;

R second;

pair(T f, R s);

pair(const pair<T,R>& other);

pair(pair<T,R>&& other);

pair<T,R>& operator=(const pair<T,R>& other);

pair<T,R>& operator=(pair<T,R>&& other);

};

template <typename T, typename R>

void pair<T,R>::\_\_copy(const pair<T,R>& other) {

first = other.first;

second = other.second;

}

template <typename T, typename R>

void pair<T,R>::\_\_move(pair<T,R>&& other) {

first = other.first;

second = other.second;

memset(&other.first, 0, sizeof(other.first));

memset(&other.second, 0, sizeof(other.second));

}

template <typename T, typename R>

pair<T,R>::pair(T f, R s) : first(f), second(s) {}

template <typename T, typename R>

pair<T,R>::pair(const pair<T,R>& other) {

\_\_copy(other);

}

template <typename T, typename R>

pair<T,R>::pair(pair<T,R>&& other) {

\_\_move(std::move(other));

}

template <typename T, typename R>

pair<T,R>& pair<T,R>::operator=(const pair<T,R>& other) {

if (this != &other) {

\_\_copy(other);

}

return \*this;

}

template <typename T, typename R>

pair<T,R>& pair<T,R>::operator=(pair<T,R>&& other) {

if (this != \*other) {

\_\_move(other);

}

return \*this;

}

#endif //OOP\_LAB1\_FINAL\_PAIR\_H

<<field.h>>

#ifndef OOP\_LAB1\_FINAL\_FIELD\_H

#define OOP\_LAB1\_FINAL\_FIELD\_H

#include <iostream>

#include "cell.h"

#include "../../utils/containers/matrix/matrix.h"

#include "../../utils/containers/pair/pair.h"

class field {

private:

matrix<cell> \_cells;

void \_\_copy(const field& other);

void \_\_move(field&& other);

public:

using entry\_exit\_generator = pair<pair<int>,pair<int>> (\*)(const field&);

static pair<pair<int>,pair<int>> default\_entry\_exit\_generator(const field& f);

field(int width, int height, entry\_exit\_generator generator = default\_entry\_exit\_generator);

field(const field& other);

field(field&& other);

int width() const;

int height() const;

cell& get(int x, int y);

const cell& get(int x, int y) const;

field& operator=(const field& other);

field& operator=(field&& other);

friend std::ostream& operator<<(std::ostream& out, const field& f);

};

#endif //OOP\_LAB1\_FINAL\_FIELD\_H

<<field.cpp>>

#include "field.h"

pair<pair<int>, pair<int>> field::default\_entry\_exit\_generator(const field &f) {

return pair<pair<int>,pair<int>>({ 0, f.height()-1 }, { f.width()-1, 0 });

}

void field::\_\_copy(const field &other) {

\_cells = other.\_cells;

}

void field::\_\_move(field &&other) {

\_cells = std::move(other.\_cells);

}

field::field(int width, int height, entry\_exit\_generator generator) : \_cells(width, height) {

pair<pair<int>,pair<int>> entry\_and\_exit = generator(\*this);

get(entry\_and\_exit.first.first, entry\_and\_exit.first.second).set\_type(CELL\_ENTRY);

get(entry\_and\_exit.second.first, entry\_and\_exit.second.second).set\_type(CELL\_EXIT);

}

field::field(const field &other) : /\*cap\*/\_cells(0, 0) {

\_\_copy(other);

}

field::field(field &&other) : /\*cap\*/\_cells(0, 0) {

\_\_move(std::move(other));

}

int field::width() const {

return \_cells.width();

}

int field::height() const {

return \_cells.height();

}

cell& field::get(int x, int y) {

return \_cells[x][y];

}

const cell& field::get(int x, int y) const {

return \_cells[x][y];

}

field& field::operator=(const field &other) {

if (this != &other) {

\_\_copy(other);

}

return \*this;

}

field& field::operator=(field &&other) {

if (this != &other) {

\_\_move(std::move(other));

}

return \*this;

}

std::ostream& operator<<(std::ostream& out, const field& f) {

out << "{ cells=" << f.\_cells << " }";

return out;

}

<<cell.h>>

#ifndef OOP\_LAB1\_FINAL\_CELL\_H

#define OOP\_LAB1\_FINAL\_CELL\_H

#include <iostream>

#include <memory>

#include "../items/artifact.h"

enum cell\_types {

CELL\_NOTHING = 0,

CELL\_ENTRY = 1,

CELL\_EXIT = 2,

};

class cell {

int \_type = 0;

artifact \_item;

void \_\_copy(const cell& other);

void \_\_move(cell&& other);

public:

cell(int type = CELL\_NOTHING, const artifact& item = artifact());

cell(const cell& other);

cell(cell&& other);

int type() const;

void set\_type(int type);

const artifact& item() const;

void move\_item(const artifact& item);

cell& operator=(const cell& other);

cell& operator=(cell&& other);

friend std::ostream& operator<<(std::ostream& out, const cell& c);

};

#endif //OOP\_LAB1\_FINAL\_CELL\_H

<<cell.cpp>>

#include "cell.h"

void cell::\_\_copy(const cell& other) {

\_type = other.\_type;

\_item = other.\_item;

}

void cell::\_\_move(cell&& other) {

\_type = other.\_type;

\_item = std::move(other.\_item);

other.\_type = CELL\_NOTHING;

}

cell::cell(int type, const artifact& item) : \_type(type), \_item(std::move(item)) {}

cell::cell(const cell& other) {

\_\_copy(other);

}

cell::cell(cell&& other) {

\_\_move(std::move(other));

}

int cell::type() const {

return \_type;

}

void cell::set\_type(int type) {

\_type = type;

}

const artifact& cell::item() const {

return \_item;

}

void cell::move\_item(const artifact& item) {

\_item = std::move(item);

}

cell& cell::operator=(const cell& other) {

if (this != &other) {

\_\_copy(other);

}

return \*this;

}

cell& cell::operator=(cell&& other) {

if (this != &other) {

\_\_move(std::move(other));

}

return \*this;

}

std::ostream& operator<<(std::ostream& out, const cell& c) {

out << "{ type=" << c.\_type << ", item=" << c.\_item << " }";

return out;

}

<<artifact.h>>

#ifndef OOP\_LAB1\_FINAL\_ARTIFACT\_H

#define OOP\_LAB1\_FINAL\_ARTIFACT\_H

#include <iostream>

#include <memory>

enum artifact\_types {

ART\_NOTHING = 0,

};

class artifact {

class data {

friend class artifact;

const int \_id = get\_id();

int \_type;

public:

data(int type);

data(const data& other);

};

std::unique\_ptr<data> \_data;

void \_\_copy(const artifact& other);

void \_\_move(artifact&& other);

static int get\_id();

public:

artifact();

artifact(int type);

artifact(const artifact& other);

artifact(artifact&& other);

int type() const;

int id() const;

bool is\_none() const;

artifact& operator=(const artifact& other);

artifact& operator=(artifact&& other);

friend std::ostream& operator<<(std::ostream& out, const artifact& art);

};

#endif //OOP\_LAB1\_FINAL\_ARTIFACT\_H

<<artifact.cpp>>

#include "artifact.h"

artifact::data::data(int type) : \_type(type) {

}

artifact::data::data(const artifact::data& other) : \_type(other.\_type) {

}

void artifact::\_\_copy(const artifact& other) {

if (other.\_data == 0)

\_data = 0;

else

\_data = std::unique\_ptr<data>(new data(other.type()));

}

void artifact::\_\_move(artifact&& other) {

\_data = std::move(other.\_data);

}

int artifact::get\_id() {

static int id = 0;

return id++;

}

artifact::artifact() : \_data(nullptr) {

}

artifact::artifact(int type) : \_data(new data(type)) {

}

artifact::artifact(const artifact& other) {

\_\_copy(other);

}

artifact::artifact(artifact&& other) {

\_\_move(std::move(other));

}

int artifact::type() const {

return \_data->\_type;

}

int artifact::id() const {

return \_data->\_id;

}

bool artifact::is\_none() const {

return \_data == 0;

}

artifact& artifact::operator=(const artifact& other) {

if (this != &other) {

\_\_copy(other);

}

return \*this;

}

artifact& artifact::operator=(artifact&& other) {

if (this != &other) {

\_\_move(std::move(other));

}

return \*this;

}

std::ostream& operator<<(std::ostream& out, const artifact& art) {

out << "{ type=";

if (art.\_data == 0)

out << "null";

else

out << art.type();

out << ", id=";

if (art.\_data == 0)

out << "null";

else

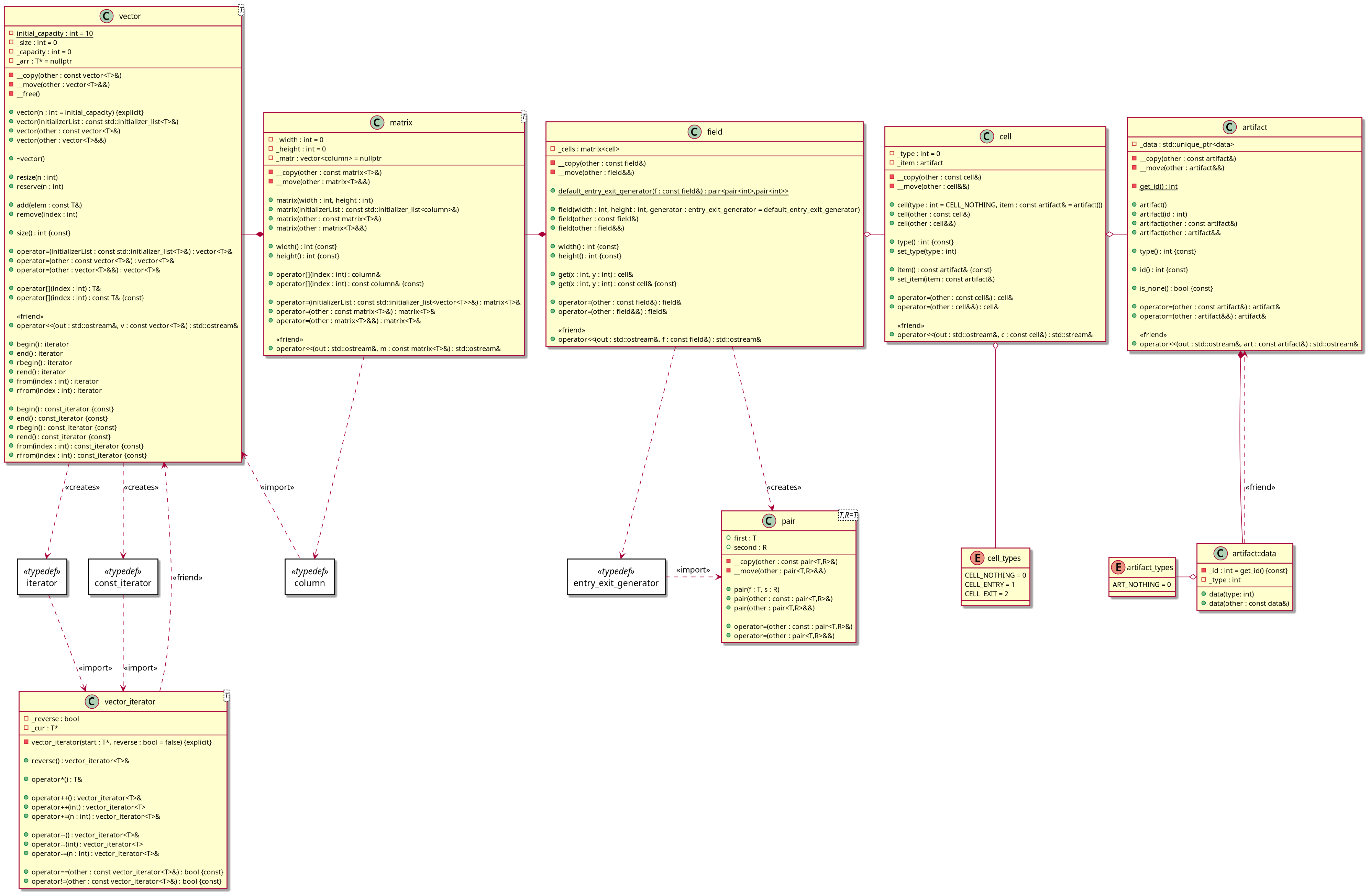
out << art.id();

out << " }";

return out;

}

**UML-диаграмма.**



**Выводы.**

Были реализованы классы vector, vector\_iterator, matrix, pair, field, cell и artifact. Была составлена uml-диаграмма классов для них. Было реализовано взаимодействие между классами.