

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 `int` типа клетки и `artifact`, представляющий из себя артефакт, лежащий на этой клетке. Каждый артефакт имеет уникальный идентификатор. Артефакт реализует семантику перемещения для перемещения артефакта 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.

