МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
“ЛЭТИ” ИМ. В. И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ
по лабораторной работе №1
по дисциплине «Объектно-ориентированное программирование»
Тема: Создание классов, конструкторов и методов классов

Студент гр. 0304 Крицын Д.Р.
Преподаватель Шевская Н.В.

Санкт-Петербург
2021

Цель работы.

Реализовать основные классы будущей игры — игровое поле, состоящее из клеток-объектов, клетки входа и выхода, а также интерфейс элемента клетки.

Задание.

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:

- Реализовать класс поля, который хранит набор клеток в виде двумерного массива.
- Реализовать класс клетки, которая хранит информацию о ее состоянии, а также того, что на ней находится.
- Создать интерфейс элемента клетки.
- Обеспечить появление клеток входа и выхода на поле. Данные клетки не должны быть появляться рядом.
- Для класса поля реализовать конструкторы копирования и перемещения, а также соответствующие операторы.
- Гарантировать отсутствие утечки памяти.

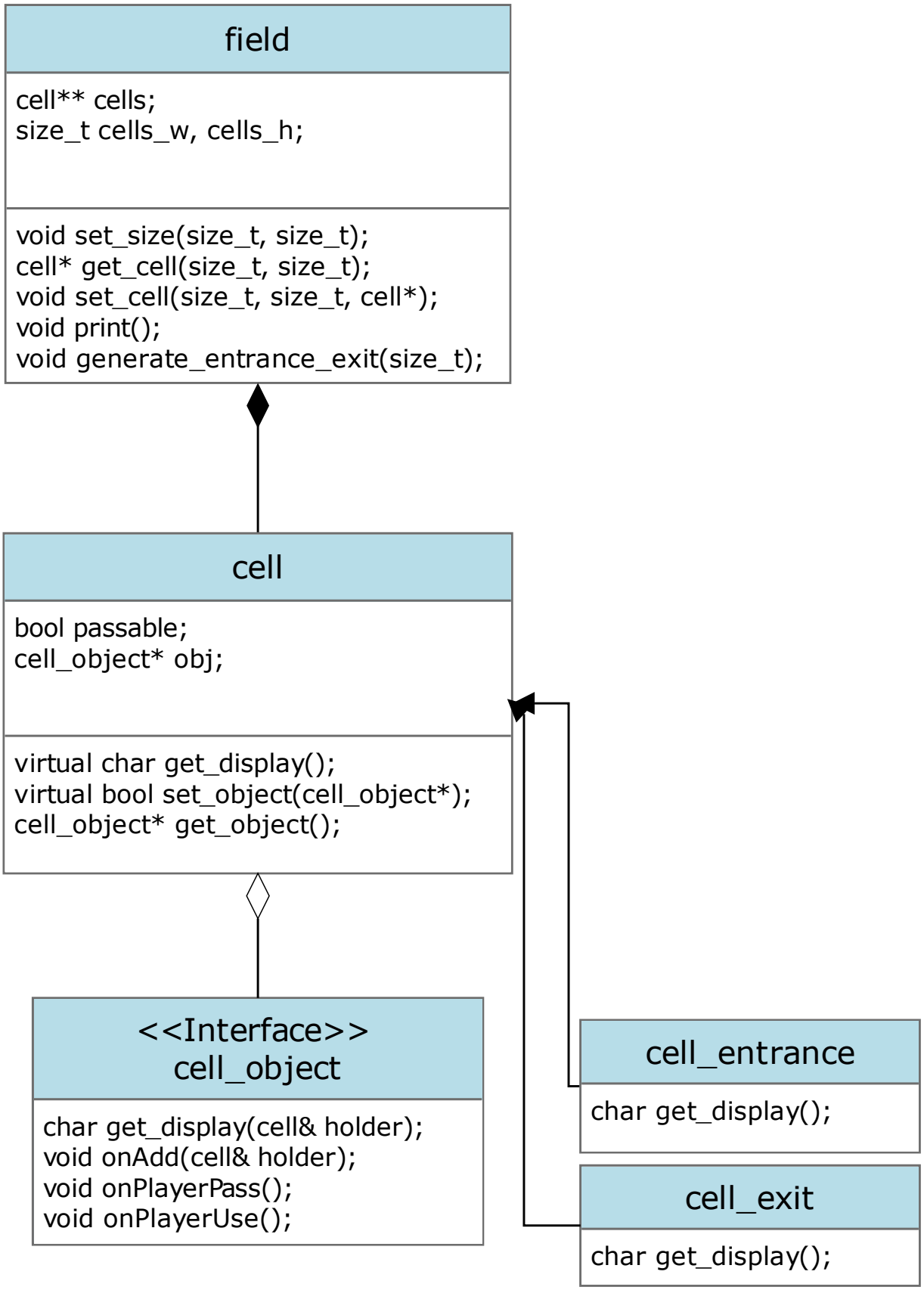
Выполнение работы.

1. Реализация класса *cell*, содержащего интерфейс для определения, является ли ячейка проходимой, а также содержащегося в ней элемента ячейки.
2. Реализация класса *field*, имеющего конструкторы и операторы копирования и перемещения, интерфейс для взаимодействия с размером поля, ячейками на нём, выводом поля на экран, генерацией входа и выхода.
3. Реализация классов *cell_entrance* и *cell_exit*, являющихся классами ячеек входа и выхода на уровень.
4. Реализация класса-интерфейса *cell_object*: описание возможных событий, на которые объект может реагировать.
5. Создание *makefile* для удобной компиляции модулей программы.

Выводы.

В ходе выполнения данной лабораторной работы была построена основа иерархии классов, которая может быть в дальнейшем дополнена и расширена до базовой игровой логики. Были изучены конструкторы операторы копирования и присваивания, такие принципы ООП, как полиморфизм (и в частности наследование).

ПРИЛОЖЕНИЕ А
UML-ДИАГРАММА КЛАССОВ



ПРИЛОЖЕНИЕ Б
ИСХОДНЫЙ КОД ПРОГРАММЫ

field.h

```
#ifndef FIELD_H
#define FIELD_H

#include <cstdlib>

#include "cells/cell.h"

class field
{
    public:

    field(const field& other);
    field(field&& other);
    field& operator=(const field& other);
    field& operator=(field&& other);

    field(size_t w, size_t h);

    // Basic manipulation methods

    size_t get_w() const;
    size_t get_h() const;
    void set_size(size_t w, size_t h);

    cell* get_cell(size_t x, size_t y) const;
    void set_cell(size_t x, size_t y, cell* c);

    // Display

    void print();

    // Generation

    void generate_entrance_exit(size_t min_dist = 3);

    private:

    cell** cells;
    size_t cells_w, cells_h;
};

#endif
```

field.cpp

```
#include "field.h"

#include "cells/cell_entrance.h"
#include "cells/cell_exit.h"

#include <cstring>
#include <algorithm>
#include <cstdlib>
#include <ctime>
#include <iostream>

field::field(const field& other)
{
    cells_w = other.cells_w; cells_h = other.cells_h;

    cells = new cell*[cells_w*cells_h];
    std::memcpy(cells, other.cells,
cells_w*cells_h*sizeof(cell));
}
field::field(field&& other)
{
    std::swap(cells_w, other.cells_w);
    std::swap(cells_h, other.cells_h);

    std::swap(cells, other.cells);
}
field& field::operator=(const field& other)
{
    if(this == &other)
        return *this;

    set_size(other.cells_w, other.cells_h);
    std::memcpy(cells, other.cells,
cells_w*cells_h*sizeof(cell));

    return *this;
}
field& field::operator=(field&& other)
{
    if(this == &other)
        return *this;

    std::swap(cells_w, other.cells_w);
    std::swap(cells_h, other.cells_h);
```

```
    std::swap(cells, other.cells);

    return *this;
}

field::field(size_t w, size_t h) : cells_w(w), cells_h(h)
{
    cells = new cell*[w*h];
}

// Display

void field::print()
{
    for(size_t x = 0; x < cells_w; ++x)
    {
        for(size_t y = 0; y < cells_h; ++y)
            if(get_cell(x, y))
                std::cout << get_cell(x, y)->get_display();
            else
                std::cout << ' ';
        std::cout << '\n';
    }
}


// Basic manipulation methods

size_t field::get_w() const { return cells_w; }
size_t field::get_h() const { return cells_h; }
void field::set_size(size_t w, size_t h)
{
    cells_w = w; cells_h = h;
    delete[] cells;
    cells = new cell*[w*h];
}

cell* field::get_cell(size_t x, size_t y) const
{
    if(x >= cells_w || y >= cells_h)
        return nullptr;
    return cells[y*cells_h + x];
}
void field::set_cell(size_t x, size_t y, cell* c)
```

```
{
    cells[y*cells_h + x] = c;
}


// Generation

void field::generate_entrance_exit(size_t min_dist)
{
    if(cells_w == 0 || cells_h == 0)
        return;

    std::srand(std::time(nullptr));

    size_t ex = std::rand() % cells_w, ey = std::rand() %
cells_h;
    set_cell(ex, ey, new cell_entrance());
    size_t x, y;
    if(cells_w > 3 && cells_h > 3){
        min_dist = std::min(min_dist, std::max(ex, cells_w -
1 - ex) + std::max(ey, cells_h - 1 - ey));

        for(x = std::rand() % cells_w, y = std::rand() %
cells_h;
            (x > ex ? x - ex : ex - x) + (y > ey ? y - ey :
ey - y) < min_dist;
            x = std::rand() % cells_w, y = std::rand() %
cells_h) {}
    }
    else if(cells_w > 1 && cells_h > 1)
        for(x = std::rand() % cells_w, y = std::rand() %
cells_h;
            x == ex || y == ey;
            x = std::rand() % cells_w, y = std::rand() %
cells_h) {}
    else
        x = 0, y = 0;

    set_cell(x, y, new cell_exit());
}
```

cell.h

```
#ifndef CELL_H
#define CELL_H

#include "../cell_object.h"
```

```
#include <stdlib.h>

class field;

class cell
{
    public:

    cell();
    cell(bool passable);

    virtual char get_display();

    virtual bool set_object(cell_object* obj);
    cell_object* get_object();

    private:

    bool passable;

    cell_object* obj;
};

#endif
```

cell.cpp

```
#include "cell.h"

cell::cell() : passable(false)
{}
cell::cell(bool passable) : passable(passable)
{}

char cell::get_display() { return ' '; }

bool cell::set_object(cell_object* obj)
{
    if(passable) this->obj = obj;
    else    return false;
    return true;
}
cell_object* cell::get_object()
{
    return obj;
}
```

cell_entrance.h

```
#ifndef CELL_ENTRANCE_H
#define CELL_ENTRANCE_H

#include "cell.h"

class cell_entrance : public cell
{
    char get_display() override;
};

#endif
```

cell_entrance.cpp

```
#include "cell_entrance.h"

char cell_entrance::get_display() { return '['; }
```

cell_exit.h

```
#ifndef CELL_EXIT_H
#define CELL_EXIT_H

#include "cell.h"

class cell_exit : public cell
{
    char get_display();
};

#endif
```

cell_exit.cpp

```
#include "cell_exit.h"

char cell_exit::get_display() { return ']'; }
```

cell_object.h

```
#ifndef CELL_OBJECT_H
#define CELL_OBJECT_H

class cell;

class cell_object
{
    public:

    virtual char get_display(cell& holder);

    // Events
```

```
    virtual void onAdd(cell& holder);

    virtual void onPlayerPass();
    virtual void onPlayerUse();
};

#endif
```