

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.

Студентка гр. 0382

Кривенцова Л.С.

Преподаватель

Жангиров Т.Р.

Санкт-Петербург

2021

Цель работы.

Изучить понятия класса, его методов и полей, научиться реализовывать простейшие классы и осуществлять межклассовые отношения.

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.

Основные теоретические положения.

ООП

- Концепцию предложили Оле-Йохан Даль и Кристен Ньюгором
- Появилась из языка ALGOL
- Сохранение фрейма в динамической памяти

- Локальные переменные сохранялись после выхода из функции
- Полиморфизм через указатели на функции

Абстракция

- Отображение только существенной информации о мире с сокрытием деталей и реализации
- Выделение интерфейса
- Единицей абстракции может быть класс или файл

Инкапсуляция

- Связь кода и данных
- Защита от внешнего воздействия
- Основа инкапсуляции в ООП - класс

Наследование

- Механизм, с помощью которого один объект перенимает свойства другого
- Позволяет добавлять классу характеристики, делающие его уникальным
- Поддержка понятия иерархической классификации
- Уменьшение количества дублирующего кода

Полиморфизм

- Реализация принципа: Один интерфейс – множество реализаций
- Механизм, позволяющий скрыть за интерфейсом общий класс действий
- Виды полиморфизма:
 - ✓ Статический
 - ✓ Динамический
 - ✓ Параметрический

Методы

- Функции определенные внутри структуры
- Отличие заключается в прямом доступе к полям структуры
- Обращение к методу аналогично обращению к полям

Неявный указатель `this`

- Методы реализованы как обычные функции, имеющие дополнительный параметр
- Неявный параметр является указателем типа структуры и имеет имя `this`
- Можно считать, что настоящая сигнатура методов следующая:
- Позволяет обратиться к полям объекта при перекрытии имен

Объявление и определение методов

- Как и для обычных функций можно разделять объявление и определение
- Объявление выносится в заголовочный файл (`.h`)
- Определение выносится в исходный файл (`.cpp`)

Перегрузка функций

- Определение нескольких функций с одинаковым именем, но отличающимся списком параметров (типами и/или количеством)

Инвариант класса

- Публичный интерфейс – список методов, доступный внешним пользователям класса
- Инвариант класса – набор утверждений, которые должны быть истинны применительно к любому объекту данного класса в любой момент времени, за исключением переходных процессов в методах объекта
- Для сохранения инварианта класса:
- Все поля должны быть закрытыми
- Публичные методы должны сохранять инвариант

Конструкторы

- Специальная функция объявляемая в классе
- Имя функции совпадает с именем класса
- Не имеют возвращаемого значения
- Предназначены для инициализации объектов

Списки инициализации

- Предназначены для инициализации полей

- Инициализации происходит в порядке объявления полей

Деструктор

- Специальные функции, объявляемые в классе
- Имя функции совпадает с именем класса, плюс знак ~ в начале
- Не имеют возвращаемого значения и параметров
- Предназначены для освобождения используемых ресурсов
- Вызывается автоматически при удалении экземпляра класса / структуры

Выполнение работы.

Ход решения:

Используется стандартная библиотека c++ и её заголовочные файлы *iostream*, *cstdlib* и *ctime* (для установки начала последовательности, генерируемой функцией *rand()*, для которой в подключен *cmath*).

Определяется тип данных (*structure*) *Coordinates*. Структура содержит два поля – *x*, *y* для хранения координат клеток игрового поля. Определяется перечисление *TYPE*, хранящее 4 состояния клетки – проходимая, непроходимая, выход и вход (*enum TYPE{PASSABLE, NOPASS, IN, OUT}*).

1. Определяется интерфейс *Cell*, от которого будут наследоваться классы различных клеток игрового поля. Не имеет полей (т.к. является интерфейсом).

а. Чистые виртуальные функции:

virtual TYPE GetType() = 0; - геттер (получает значение полей, имеющих модификатор доступа *privat*) для доступа к полю типа клетки (поле будет реализовано в классах-наследниках).

virtual void SetType(TYPE t) = 0; - сеттер (устанавливает значение для полей, имеющих модификатор доступа *private*) для доступа редактирования в методах другого класса поля типа клетки (это поле будет реализовано в классах-наследниках).

virtual void SetPoint(int x, int y)=0; - сеттер для доступа редактирования поля позиции клетки на игровом поле в методах другого класса (это поле будет реализовано в классах-наследниках).

Coordinates GetPoint() final; - геттер для доступа к полю позиции клетки на

игровом поле (это поле будет реализовано в классах-наследниках).

virtual ~Cell(){}; - деструктор.

2. Определяется класс клетки игрового поля *Cellule* (наследник *Cell*).

а. Определяются два поля с модификатором доступа *private*:

Coordinates position{}; - поле представляет собой структуру, хранящую два целых числа - координаты данной клетки (её позиция на игровом поле).

TYPE type; - поле хранит тип клетки - один из четырех доступных (из перечисления) вариантов.

б. Реализуются методы класса с модификатором доступа *public*:

Cellule():position({0,0}), type(PASSABLE){} – конструктор со списком инициализации полей.

TYPE GetType() final; – Возвращает тип данной клетки игрового поля. Предполагается отсутствие дальнейшего переопределения в случае наследования (*final*).

void SetType(TYPE t) final; - Ничего не возвращает, присваивает полю *type* один из доступных типов клетки. Предполагается отсутствие дальнейшего переопределения в случае наследования (*final*).

void SetPoint(int a, int b) final; - Ничего не возвращает, присваивает полю *position* значения координат данной клетки. Предполагается отсутствие дальнейшего переопределения в случае наследования (*final*).

Coordinates GetPoint() final; - Возвращает позицию клетки игрового поля. Предполагается отсутствие дальнейшего переопределения в случае наследования (*final*).

3. Определяется класс внешнего представления клетки игрового поля *CelluleView*. Не является ничьим наследником, однако зависит от класса *Cellule*, так как инициализирует своё поле (*view*) в зависимости от состояния класса *Cellule* (его поля *Type*).

а. Определяется поле с модификатором доступа *private*:

char view; - Хранит символ, являющийся внешнем представлением клетки игрового поля.

b. Определяются методы класса с модификатором доступа *public*:

explicit CelluleView(Cellule& one); - Конструктор класса. Принимает ссылку на объект класса *Cellule*, от которого зависит. С помощью конструкции *switch* инициализирует поле *view* (в зависимости от состояния поля *Cellule::Type*): '0' – если тип клетки - проходимый, '1' – если тип клетки - непроходимый, '^' – тип клетки - вход, '>' – тип клетки - выход. Неявное преобразование этого конструктора запрещено.

char GetView() const; - геттер для доступа к полю внешнего представления игровой клетки. Возвращает символ, не меняет никакие состояния.

virtual ~CelluleView(){}; - чистый виртуальный деструктор.

4. Определяется класс игрового поля *Field*. Межклассовые отношения: связан ассоциацией с классами *Cellule* и *CelluleView*.

a. Определяются поля с модификатором доступа *private*:

*Cellule** field;* - поле, хранящее двумерный массив объектов класса *Cellule* – матрицу игрового поля.

int width; - целое число, хранит ширину игрового поля.

int height; - целое число, хранит длину игрового поля.

Coordinates in{}; - объект структуры координат, хранящий нахождение клетки входа.

Coordinates out{}; - объект структуры координат, хранящий нахождение клетки выхода.

b. Реализуются методы класса с модификатором доступа *public*:

explicit Field(int w = 10, int h = 10); - Конструктор. Принимает два целочисленных числа (размерность игрового поля), их значения также заданы по умолчанию. В конструкторе выделяется память для матрицы (поле *field*) и, с помощью сеттера поля *Cellule::position* каждой игровой клетке этой матрицы устанавливаются координаты. Неявное преобразование этого конструктора запрещено.

~Field(); - Деструктор класса, в котором освобождается память массива (поле *field*).

Field(const Field &other) : width(other.width), height(other.height), field(new Cellule[width])* ; - Конструктор копирования класса. Принимает в качестве входного параметра константную ссылку на объект этого же класса. С помощью списка инициализации и цикла *for* создаваемому объекту присваиваются данные – члены класса из исходного объекта.

Field& operator = (const Field &other); - Перегрузка оператора присваивания копированием. Принимает в качестве входного параметра константную ссылку на объект этого же класса. После проверки на попытку присвоить объект самому себе, освобождается память матрицы(поля объекта класса) с помощью *delete[]*. Затем во вложенном цикле *for* создаваемому объекту присваиваются данные – члены класса из исходного объекта.

Field(Field&& other) ; - Конструктор перемещения класса. Принимает в качестве параметра ссылку *rvalue* на тип класса. С помощью с помощью функции *swap* стандартной библиотеки значения полей исходного объекта присваиваются текущему объекту.

Field& operator=(Field&& other) ; - Перегрузка оператора присваивания перемещением. Принимает в качестве параметра ссылку *rvalue* на тип класса, возвращает ссылку на тип класса. После проверки на попытку присвоить объект самому себе, с помощью функции *swap* стандартной библиотеки значения полей исходного объекта присваиваются текущему объекту. Возвращается ссылка на текущий объект.

void MakeInOut(); - Метод класса, обеспечивающий появление клеток входа и выхода в игровом поле. Значение поля класса координат входа инициализируются с помощью функции *rand()* (в диапазоне размерности поля). Полю объекта *Cellule::Type* (хранящемуся в матрице игрового поля *Field::field* по индексу только что сгенерированных координат) через сеттер присваивается тип клетки «Вход» (*.SetType(IN)*). Аналогично генерируется клетка «Выхода» (в цикле *while* координаты генерируются случайным образом до тех пор, пока потенциальная клетка «Выхода» не будет лежать достаточно далеко от уже установленной клетки «Входа». Дальность – разность координат должна

составлять минимум 20% длины координатной оси (длины и ширины)).

Coordinates GetIn(); - геттер для доступа к полю координат игровой клетки входа. Возвращает символ, не меняет никакие состояния.

void MakeObjects(); - Метод класса, устанавливающий проходимость клеток. В ней запускается вложенный цикл *for*, в котором в каждой итерации создаётся структура координат игровой клетки (*Coordinates*), и с 10% шансом клетке игрового поля с координатами, хранящимися в созданной структуре присваивается (с помощью сеттера) тип клетки *NOPASS* – такая игровая клетка становится непроходимой.

void PrintBorder() const; - Константный метод, является вспомогательным для метода *Print()*, в котором и вызывается. Печатает на экран горизонтальную границу поля. Этот функционал вынесен в отдельный метод только во избежание дублирования кода.

void Print() const; - Константный метод, печатающий актуальное состояние игрового поля на экран, добывающий внешнее представление текущей клетки с помощью геттера *.GetView()*.

void Start(); - Метод класса, отвечающий за запуск игры. Устанавливает начальное состояние для функции *rand()* (*srand(time(NULL))*), вызывает методы, генерирующие состояния клеток *MakeInOut()* и *MakeObjects()* и метод печати поля на экран *Print()*.

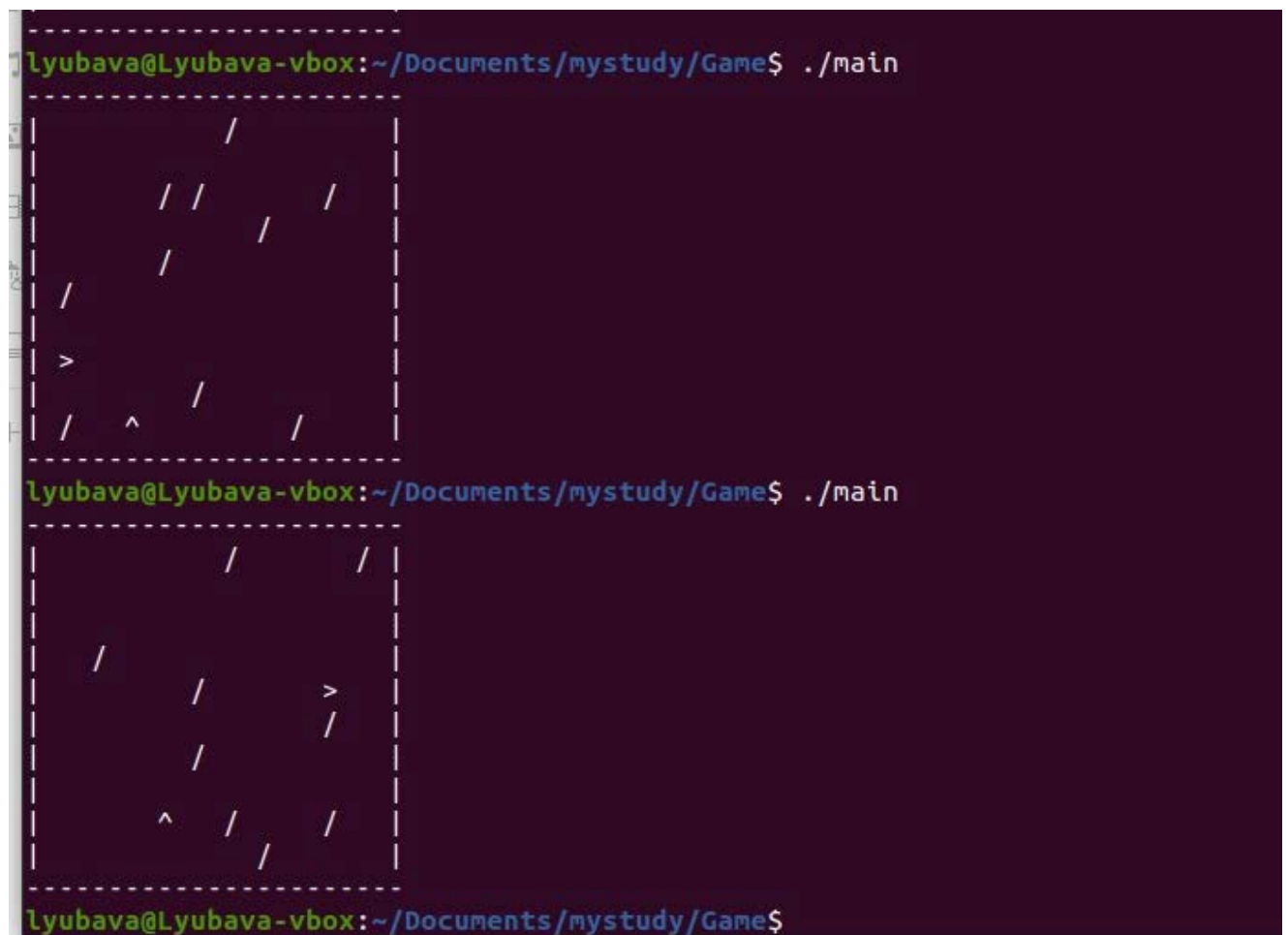
5. Главная функция *main()*.

Field game; - объявляется объект класса игрового поля.

game.Start(); - у созданного объекта класса вызывается метод начала игры

Результат работы программы:

Рис 1. – демонстрация работы программы в терминале Ubuntu.



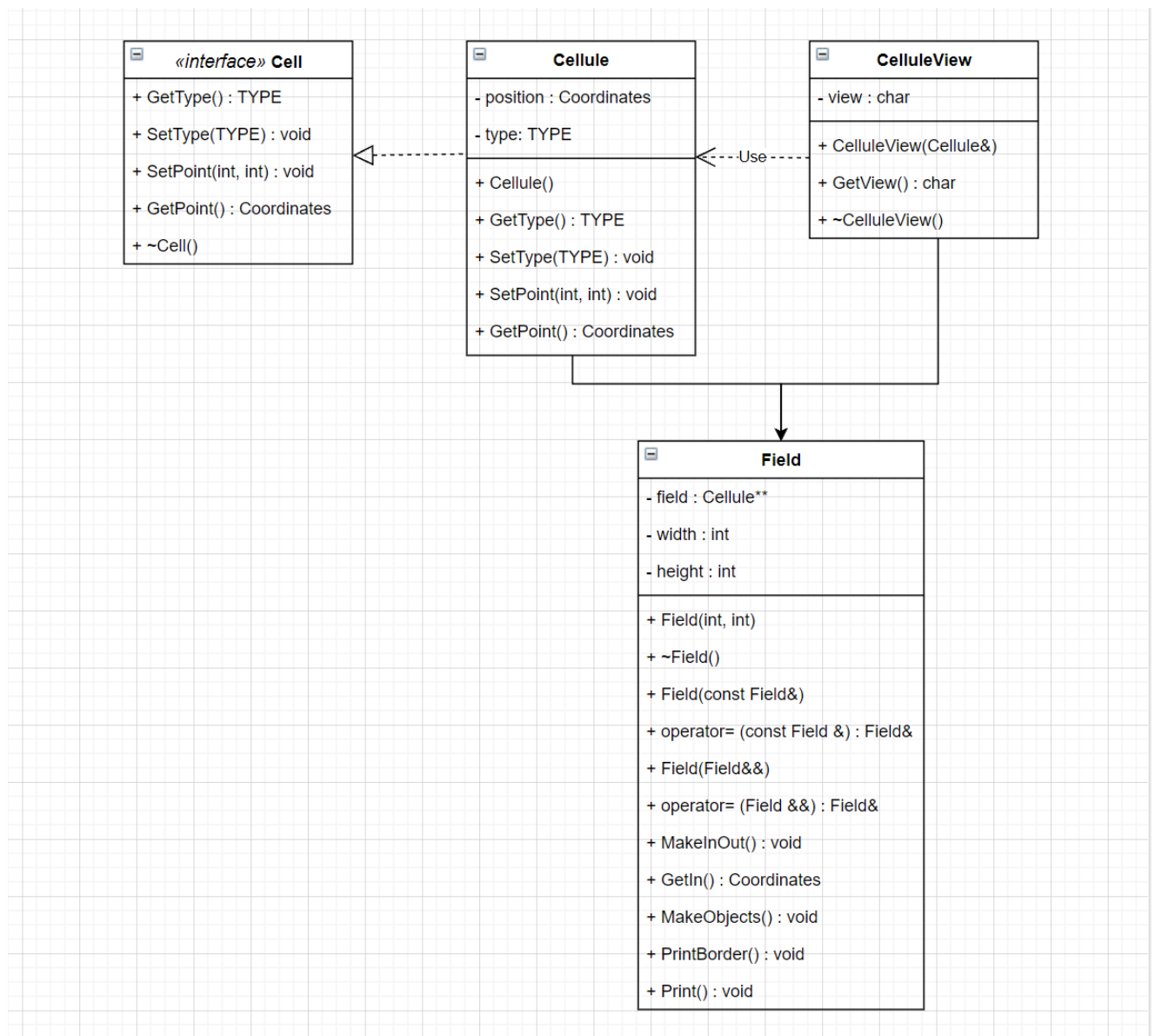
```
lyubava@Lyubava-vbox:~/Documents/mystudy/Game$ ./main
      /
    / /
  / / /
 / / /
> / /
 / ^ /
 / /

lyubava@Lyubava-vbox:~/Documents/mystudy/Game$ ./main
      /      /
    /      /
  /      /
 /      /
>      /
 /      /
 ^      /
 /      /

lyubava@Lyubava-vbox:~/Documents/mystudy/Game$
```

UML-диаграмма межклассовых отношений:

Рис 2. – UML-диаграмма.



Программный код см. в приложении А.

Выводы.

Были изучены понятия класса, его методов и полей, получены навыки реализовывать простейшие классы и осуществлять межклассовые отношения.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.cpp

```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include "structs.h"
#include "Cell.h"
#include "Cellule.h"
#include "Field.h"
int main() {
    Field game;
    game.Start();
    return 0;
}
```

Название файла: structs.h

```
#pragma once
enum TYPE{PASSABLE, NOPASS, IN, OUT};

struct Coordinates{
    int x;
    int y;
};
```

Название файла: Cell.h

```
#pragma once
#include <iostream>
#include <cstdlib>
#include <ctime>
#include "structs.h"
class Cell{
public:
    virtual TYPE GetType() = 0;
    virtual void SetType(TYPE t) = 0;
    virtual void SetPoint(int x, int y)=0;
    virtual Coordinates GetPoint() = 0;
    virtual ~Cell(){};
};
```

Название файла: Cellule.h

```
#pragma once
#include <iostream>
#include <cstdlib>
#include <ctime>
#include "structs.h"
#include "Cell.h"
class Cellule: public Cell {
public:
    Cellule();
    TYPE GetType() final;
    void SetType(TYPE t) final;
    void SetPoint(int a, int b) final;
    Coordinates GetPoint() final;
private:
```

```

    Coordinates position{};
    TYPE type;
};

```

Название файла: Cellule.cpp

```

#include "Cellule.h"
Cellule::Cellule():position({0,0}), type(PASSABLE){}

TYPE Cellule::GetType() {
    return this->type;
}

void Cellule::SetType(TYPE t) {
    this->type = t;
}

void Cellule::SetPoint(int a, int b) {
    this->position.x = a;
    this->position.y = b;
}
Coordinates Cellule::GetPoint() {
    return this->position;
}

```

Название файла: CelluleView.h

```

#pragma once
#include <iostream>
#include <cstdlib>
#include <ctime>
#include "structs.h"
#include "Cell.h"
#include "Cellule.h"
class CelluleView {
public:
    explicit CelluleView(Cellule& one);
    char GetView() const;
    virtual ~CelluleView();
private:
    char view;
};

```

Название файла: CelluleView.cpp

```

#include "CelluleView.h"
CelluleView::CelluleView(Cellule& one) {
    switch (one.GetType())
    {
        case PASSABLE:
            this->view = ' ';
            break;
        case NOPASS:
            this->view = '/';
            break;
        case OUT:
            this->view = '>';
            break;
    }
}

```

```

        case IN:
            this->view = '^';
            break;
        default:
            this->view = '?';
            break;
    }
}

char CelluleView::GetView() const {
    return this->view;
}
CelluleView::~CelluleView(){};

```

Название файла: Field.h

```

#pragma once
#include <iostream>
#include <cstdlib>
#include <ctime>
#include "structs.h"
#include "Cellule.h"
#include "CelluleView.h"
class Field {
public:
    explicit Field(int w = 10, int h = 10);
    ~Field();
    Field(const Field &other);
    Field& operator = (const Field &other);
    Field(Field&& other);
    Field& operator=(Field&& other);
    void MakeInOut();
    Coordinates GetIn();
    void MakeObjects();
    void PrintBorder() const;
    void Print() const;
    void Start();

private:
    Cellule** field;
    int width;
    int height;
    Coordinates in{};
    Coordinates out{};
};

```

Название файла: Field.cpp

```

#include "Field.h"
#include <cmath>
Field::Field(int w, int h){
    width = w;
    height = h;
    field = new Cellule*[height];
    for (int i = 0; i < height; i++){
        field[i] = new Cellule[width];
        for (int j = 0; j < width; j++){

```

```

        field[i][j].SetPoint(j, i);
    }
}
}
Field::~Field() {
    for (int i = 0; i < height; i++) {
        delete[] field[i];
    }
    delete[] field;
}

Field::Field(const Field& other) : width(other.width),
height(other.height),
                                in(other.in), out(other.out), field(new
Cellule*[height]) { // Конструктор копирования
    for (int i = 0; i < height; i++) {
        field[i] = new Cellule[width];
        for (int j = 0; j < width; j++){
            field[i][j] = other.field[i][j];
        }
    }
}

Field & Field:: operator = (const Field &other) { // Оператор
присваивания копированием
    if (this != &other){
        for (int i = 0; i < height; i++){
            delete[] field[i];
        }
        delete[] field;
        width = other.width;
        height = other.height;
        in = other.in;
        out = other.out;
        field = new Cellule* [height];
        for (int i = 0; i < height; i++) {
            field[i] = new Cellule[width];
            for (int j = 0; j < width; j++){
                field[i][j] = other.field[i][j];
            }
        }
    }
    return* this;
}

Field::Field(Field&& other) { // Конструктор перемещения
    std::swap(this->width, other.width);
    std::swap(this->height, other.height);
    std::swap(this->field, other.field);
    std::swap(this->in, other.in);
    std::swap(this->out, other.out);
}

Field & Field:: operator=(Field&& other) { // Оператор присваивания
перемещением

```

```

        if (this != &other) {
            std::swap(this->width, other.width);
            std::swap(this->height, other.height);
            std::swap(this->field, other.field);
            std::swap(this->in, other.in);
            std::swap(this->out, other.out);
        }
        return* this;
    }

    void Field::MakeInOut() {
        in = {rand() % width, rand() % height};
        field[in.y][in.x].SetType(IN);
        out = {rand() % width, rand() % height};
        while (abs(out.x - in.x) < std::round(0.2 * width) || abs(out.y -
in.y) < std::round(0.2 * height) ){

            out = {rand() % width, rand() % height};
        }
        field[out.y][out.x].SetType(OUT);
    }

    Coordinates Field::GetIn() {
        return this->in;
    }

    void Field::MakeObjects() {
        int chance;
        for (int i = 0; i < height; i++) {
            for (int j = 0; j < width; j++) {
                Coordinates any = {j, i};
                chance = rand() % 100 + 1;
                if ((chance <= 10) && (field[any.y][any.x].GetType() != IN)
&& (field[any.y][any.x].GetType() != OUT)) //10% chance
                    field[any.y][any.x].SetType(NOPASS);
            }
        }
    }

    void Field::PrintBorder() const {
        for (int i = 0; i < width + 1; i++)
            std::cout << "--";
        std::cout << '-' << std::endl;
    }

    void Field::Print() const {
        PrintBorder();
        for (int i = 0; i < height; i++){
            std::cout << "| ";
            for (int j = 0; j < width; j++) {
                std::cout << CelluleView(this->field[i][j]).GetView() << '
';
            }

            std::cout << '|' << std::endl;
        }
    }

```



```
        }  
        PrintBorder();  
    }  
  
    void Field::Start() {  
        srand(time(NULL));  
        MakeInOut();  
        MakeObjects();  
        Print();  
    }
```