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 классов, методов классов, наследование.

Студент гр. 8304 _____ Воропаев А.О.

Преподаватель _____ Размочаева Н.В.

Санкт-Петербург

2020

Цель работы.

Научиться создавать классы и их конструкторы, реализовать методы классов и познакомиться с наследованием классов.

Задание.

Разработать и реализовать набор классов:

- Класс игрового поля
- Набор классов юнитов

Игровое поле является контейнером для объектов представляющим прямоугольную сетку. Основные требования к классу игрового поля:

- Создание поля произвольного размера
- Контроль максимального количества объектов на поле
- Возможность добавления и удаления объектов на поле
- Возможность копирования поля (включая объекты на нем)
- Для хранения запрещается использовать контейнеры из stl

Юнит является объектом, размещаемым на поля боя. Один юнит представляет собой отряд. Основные требования к классам юнитов:

- Все юниты должны иметь как минимум один общий интерфейс
- Реализованы 3 типа юнитов (например, пехота, лучники, конница)
- Реализованы 2 вида юнитов для каждого типа(например, для пехоты могут быть созданы мечники и копейщики)
- Юниты имеют характеристики, отражающие их основные атрибуты, такие как здоровье, броня, атака.
- Юнит имеет возможность перемещаться по карте

Ход работы.

1) Был реализован класс Field хранящий информацию о поле, а также имеющий методы позволяющие добавлять/удалять объекты на поле. Каждая ячейка поля содержит информацию о типе располагающегося на ней юнита (nullptr если юнита нет). Для поля также реализован конструктор копирования/перемещения и оператор присваивания/перемещения.

2) Был реализован класс Unit хранящий основные методы, применимые для каждого из 3-х типов юнитов. Он является основным классом юнитов. Он содержит виртуальные методы, такие как move и attack.

3) Были реализованы дочерние классы Archer, Knight, Cavalry класса Unit. Они содержат методы применимые только для юнитов соответствующего типа. Также различается и их дальность боя, урон и броня.

Выводы.

Был разработан и реализован набор классов: класс игрового поля, набор классов юнитов, удовлетворяющие требованиям, таким как, создание поля произвольного размера, контроль максимального количества объектов на поле. Возможность добавления, удаления объектов, копирования поля. Юниты имеют один общий интерфейс, есть возможность перемещаться по карте.

Приложение А.

Исходный код.

Файл unit.h:

```
#pragma once

#include <cmath>
#include <utility>
#include <memory>

class Field;

class Location {
public:
   Location(int x, int y) {
      this->position.first = x;
      this->position.second = y;
   }

   explicit Location(std::pair<int, int> pos) {
      this->position = pos;
   }

   Location() = default;
   ~Location() = default;

   void setPosition(int x, int y) {
      this->position.first = x;
      this->position.second = y;
   }

   void setPosition(std::pair<int, int> pos) {
      this->position = pos;
   }

   std::pair<int, int> getPosition() const {
      return position;
   }

private:
   std::pair<int, int> position;


};


class Health
{
public:
   Health() {
      health = 0;
   }
   ~Health() = default;

   explicit Health(int h) {
      this->health = h;
   }

   int getHealth() {
      return health;
```

```
    }

    void setHealth(int h) {
        this->health = h;
    }

    void actDamaged(int dmg) {

        this->health -= dmg;
    }

    void actHealed(int heal) {
        this->health += heal;
    }

private:
    int health;

};


class Damage
{
public:
    Damage() {
        damage = 0;
        range = 0;
    }
    ~Damage() = default;

    Damage(int d, int r) {
        this->damage = d;
        this->range = r;
    }

    int getDamage() const {
        return damage;

    }

    void setDamage(int d) {
        damage = d;
    }

    int getRange() const {
        return range;

    }

    void setRange(int r) {
        range = r;
    }

    void actDecreaseDmg(int debuff) {

        damage -= debuff;
    }

    void actIncreaseDmg(int buff) {
        damage += buff;
    }

private:
    int damage;
```

```
    int range;

};


class Armor
{
public:

    Armor() {
        armor = 0;
    }
    ~Armor() = default;

    explicit Armor(int a) {
        this->armor = a;
    }

    int getArmor() const {
        return armor;

    }

    void setArmor(int a) {
        armor = a;
    }

    void actDecreaseArmor(int debuff) {

        armor -= debuff;
    }

    void actIncreaseArmor(int buff) {
        armor += buff;
    }

private:
    int armor;

};


class Unit :public Armor, public Health, public Damage, public Location {
public:
    Unit() = default;

    virtual Unit* clone() = 0;

    virtual ~Unit() = 0;

     void move(Field *f, int dX, int dY);

     void attack(Field *f, Unit* target);
};


class Archer : public Unit {
public:
    Archer() = default;

    explicit Archer(int health, int dmg, int range, int armor);

    ~Archer() override = 0;
```

```
};


class MasterArcher : public Archer {
public:
   MasterArcher() : Archer(500, 50, 10, 15)
   {};

   Unit* clone() final{
       auto tmp = new MasterArcher;

       tmp->setHealth(this->getHealth());

        tmp->setDamage(this->getDamage());

        tmp->setRange(this->getRange());

        tmp->setArmor(this->getArmor());

        return (Unit*) tmp;

   }

   ~MasterArcher() final = default;
};


class NoviceArcher : public Archer {

public:
   NoviceArcher() : Archer(300, 30, 10, 8)
   {};

    Unit* clone() final{
        auto tmp = new NoviceArcher;

        tmp->setHealth(this->getHealth());

        tmp->setDamage(this->getDamage());

        tmp->setRange(this->getRange());

        tmp->setArmor(this->getArmor());

        return (Unit*) tmp;

    }

   ~NoviceArcher() final = default;
};


class Knight : public Unit {

public:

   Knight() = default;

   explicit Knight(int health, int dmg, int range, int armor);

   ~Knight() override = 0;

};
```

```
class Swordsman : public Knight {

public:
   Swordsman() : Knight(900, 75, 5, 20)
   {};

    Unit* clone() final{
        auto tmp_for_clone = new Swordsman;

        tmp_for_clone->setHealth(this->getHealth());

        tmp_for_clone->setDamage(this->getDamage());

        tmp_for_clone->setRange(this->getRange());

        tmp_for_clone->setArmor(this->getArmor());

        return (Unit*) tmp_for_clone;

    }

   ~Swordsman() final = default;
};


class Shieldman : public Knight {

public:
   Shieldman() : Knight(1400, 40, 2, 35)
   {};

    Unit* clone() final{
        auto tmp_for_clone = new Shieldman;

        tmp_for_clone->setHealth(this->getHealth());

        tmp_for_clone->setDamage(this->getDamage());

        tmp_for_clone->setRange(this->getRange());

        tmp_for_clone->setArmor(this->getArmor());

        return (Unit*) tmp_for_clone;

    }

   ~Shieldman() final = default;
};



class Cavalry : public Unit {

public:

   Cavalry() = default;

   explicit Cavalry(int health, int dmg, int range, int armor);

   ~Cavalry() override = 0;
};
```

```
class Paladin : public Cavalry {

public:
   Paladin() : Cavalry(800, 60, 8, 20)
   {};

    Unit* clone() final{
        auto tmp_for_clone = new Paladin;

        tmp_for_clone->setHealth(this->getHealth());

        tmp_for_clone->setDamage(this->getDamage());

        tmp_for_clone->setRange(this->getRange());

        tmp_for_clone->setArmor(this->getArmor());

        return (Unit*) tmp_for_clone;

    }

   ~Paladin() final = default;
};


class Cavalier : public Cavalry {

public:
    Cavalier() : Cavalry(600, 55, 8, 20)
    {};

    Unit* clone() final{
        auto tmp = new Cavalier;

        tmp->setHealth(this->getHealth());

        tmp->setDamage(this->getDamage());

        tmp->setRange(this->getRange());

        tmp->setArmor(this->getArmor());

        return (Unit*) tmp;

    }

   ~Cavalier() final = default;
};
```

Файл cell.cpp:

```
#include "Cell.h"

Cell::Cell():Object(nullptr){}
void Cell::clearCell() {
    Object = nullptr;
}
void Cell::setCharacter(Unit* newUnit) {
    if(Object == nullptr)
        Object = newUnit;
```

```
}
Unit* Cell::getCharacter() const{
    return this->Object;
}
Cell::Cell(const Cell& obj):Cell()
{
    this->Object = obj.Object;
}
Cell& Cell::operator=(const Cell& obj) {
    if(this == &obj)
        return *this;
    this->Object = obj.Object;
    return *this;
}
```

Файл Unit.cpp:

```
#include <iostream>
#include "Unit.h"
#include "Field.h"
#define DEFAULT 1

Unit::~Unit() = default;

void Unit::attack(Field *f, Unit* target) {
    if (!f){
        std::cout << "Invalid field pointer passed\n";
        return;
    }

    if (!target){
        std::cout << "Invalid target passed\n";
        return;
    }

   if (sqrt(pow(abs(this->getPosition().first - target->getPosition().first), 2)
      + pow(abs(this->getPosition().second - target->getPosition().second), 2)) <= this-
>getRange())
   {
      target->actDamaged(this->getDamage());
      return;
   }

   else{
       std::cout << "Out of attack range\n";
       return;
   }

}


void Unit::move(Field *f, int dX, int dY) {
    if (!f){
        std::cout << "Invalid field passed\n";
        return;
    }

   if (!(f->moveUnit(this->getPosition().first, this->getPosition().second, dX, dY))){
       std::cout << "Move failed\n";
       return;
```

```
    }
    this->setPosition(this->getPosition().first + dX, this->getPosition().second + dY);
}



Archer::~Archer() = default;

Archer::Archer(int health, int dmg, int range, int armor) {
    if (health <= 0){
        std::cout << "Cannot create negative-health unit, health set to DEFAULT\n";
        this->setHealth(DEFAULT);
    }
    else {
        this->setHealth(health);
    }

    if (dmg <= 0){
        std::cout << "Cannot create negative-dmg unit, health set to DEFAULT\n";
        this->setDamage(DEFAULT);
    }
    else {
        this->setDamage(dmg);
    }

   this->setRange(range);
    if (range <= 0){
        std::cout << "Cannot create negative-attack-range unit, health set to DEFAULT\n";
        this->setRange(DEFAULT);
    }
    else {
        this->setRange(range);
    }

   this->setArmor(armor);
    if (armor <= 0){
        std::cout << "Cannot create negative-armor unit, health set to DEFAULT\n";
        this->setArmor(DEFAULT);
    }
    else {
        this->setArmor(armor);
    }
}



Knight::~Knight() = default;

Knight::Knight(int health, int dmg, int range, int armor) {
    if (health <= 0){
        std::cout << "Cannot create negative-health unit, health set to DEFAULT\n";
        this->setHealth(DEFAULT);
    }
    else {
        this->setHealth(health);
    }

    if (dmg <= 0){
        std::cout << "Cannot create negative-dmg unit, health set to DEFAULT\n";
        this->setDamage(DEFAULT);
    }
    else {
        this->setDamage(dmg);
```

```
    }

    this->setRange(range);
    if (range <= 0){
        std::cout << "Cannot create negative-attack-range unit, health set to DEFAULT\n";
        this->setRange(DEFAULT);
    }
    else {
        this->setRange(range);
    }

    this->setArmor(armor);
    if (armor <= 0){
        std::cout << "Cannot create negative-armor unit, health set to DEFAULT\n";
        this->setArmor(DEFAULT);
    }
    else {
        this->setArmor(armor);
    }
}



Cavalry::~Cavalry() = default;

Cavalry::Cavalry(int health, int dmg, int range, int armor) {
    if (health <= 0){
        std::cout << "Cannot create negative-health unit, health set to DEFAULT\n";
        this->setHealth(DEFAULT);
    }
    else {
        this->setHealth(health);
    }

    if (dmg <= 0){
        std::cout << "Cannot create negative-dmg unit, health set to DEFAULT\n";
        this->setDamage(DEFAULT);
    }
    else {
        this->setDamage(dmg);
    }

    this->setRange(range);
    if (range <= 0){
        std::cout << "Cannot create negative-attack-range unit, health set to DEFAULT\n";
        this->setRange(DEFAULT);
    }
    else {
        this->setRange(range);
    }

    this->setArmor(armor);
    if (armor <= 0){
        std::cout << "Cannot create negative-armor unit, health set to DEFAULT\n";
        this->setArmor(DEFAULT);
    }
    else {
        this->setArmor(armor);
    }
}
```

Файл dwarf.cpp:

```
#include "Dwarf.h"

Dwarf::Dwarf(){
    health = 250;
    armor = 250;
    attack = 40;
    attackRange = 2;
}
char Dwarf::getClass() const {
    return 'D';
}
```

Файл dwarf.h:

```
#ifndef OOP_1_DWARF_H

#define OOP_1_DWARF_H
#include "Warrior.h"
class Dwarf: public Warrior{
public:
    Dwarf();
    char getClass() const;
};
#endif //OOP_1_DWARF_H
```

Файл elf.cpp:

```
#include "Elf.h"

Elf::Elf()  {
    health = 125;
    armor = 50;
    attack = 30;
    attackRange = 2;
}
char Elf::getClass() const {
    return 'E';
}
```

Файл elf.h:

```
#ifndef OOP_1_ELF_H

#define OOP_1_ELF_H
#include "Thief.h"
class Elf: public Thief {
public:
    Elf();
    char getClass() const;
};
#endif //OOP_1_ELF_H
```

Файл field.cpp:

```
#include <iostream>
#include "Field.h"
Field::Field(int x, int y) :X(x), Y(y), matrix(new Cell*[X]),
objectsCounter(0), maxObjectsOnField(x*y){
    for(int i = 0; i < X; i++) {
        matrix[i] = new Cell[Y];
        for(int j = 0; j < Y; j++)
            matrix[i][j].setCharacter(nullptr);
    }
}
Field::Field():Field(0,0) {}
Field::~Field()
{
    if(matrix)
    {
        for(int i = 0; i < X; i++)
            if(matrix + i)
                delete[] matrix[i];
        delete[] matrix;
    }
}
void Field::addObject(Unit *Object, int x, int y) {
    if(!Object->isOnField())
        Object->replace(this);
    if(checkPoint(x,y) && !matrix[x][y].getCharacter() &&  objectsCounter <
maxObjectsOnField) {
        matrix[x][y].setCharacter(Object);
        Object->setX(x);
        Object->setY(y);
        objectsCounter++;
    }
}
int Field::getObjectCounter() const{
    return objectsCounter;
}
void Field::deleteObject(Unit *Object) {
    int requiredX = Object->getX();
    int requiredY = Object->getY();
    if(checkPoint(requiredX, requiredY) &&
matrix[requiredX][requiredY].getCharacter() == Object) {
        matrix[requiredX][requiredY].clearCell();
    }
    objectsCounter--;
}
void Field::print() const{
    printf("------------------\n");
    for(int i = 0; i < X;i++){
        for(int j = 0; j < Y; j++)
            if(matrix[i][j].getCharacter() != nullptr)
                std::cout << matrix[i][j].getCharacter()->getClass() << "
";
            else
```

```
                std::cout << ". ";
        std::cout << std::endl;
    }
    printf("------------------\n");
}
bool Field::checkPoint(int x, int y) const {
    return x < X && y < Y;
}
void Field::swapCharacters(int x1, int y1, int x2, int y2) {
    if(x1 == x2 && y1 == y2)
        return;
    //std::swap(matrix[x1][y1].Object, matrix[x2][y2].Object);
    Unit* tmp = matrix[x1][y1].getCharacter();
    matrix[x1][y1].clearCell();
    matrix[x1][y1].setCharacter(matrix[x2][y2].getCharacter());
    matrix[x2][y2].clearCell();
    matrix[x2][y2].setCharacter(tmp);
}
int Field::getX() const {
    if(matrix != nullptr)
        return X;
    return -1;
}
int Field::getY() const {
    if(matrix != nullptr)
        return Y;
    return -1;
}
Field::Field(const Field& obj) : Field(obj.X,obj.Y)
{
    objectsCounter = obj.objectsCounter;
    for(int i = 0; i < X; i++)
        for(int j = 0; j < Y; j++)
            this->matrix[X][Y] = obj.matrix[X][Y];
}
Field::Field(Field&& obj):X(0), Y(0), matrix(nullptr){
    std::swap(matrix, obj.matrix);
    std::swap(X, obj.X);
    std::swap(Y, obj.Y);
    std::swap(objectsCounter, obj.objectsCounter);
    std::swap(maxObjectsOnField, obj.maxObjectsOnField);
}
Field& Field::operator=(const Field& obj) {
    if(this == &obj)
        return *this;
    this->matrix = new Cell*[obj.X];
    this->X = obj.X;
    this->Y = obj.Y;
    this->objectsCounter = obj.objectsCounter;
    this->maxObjectsOnField = obj.maxObjectsOnField;
    for(int i = 0; i < X; i++) {
        this->matrix[i] = new Cell[Y];
        for (int j = 0; j < Y; j++) {
            this->matrix[i][j] = obj.matrix[i][j];
        }
    }
```

```
    return *this;
}
Field& Field::operator=(Field &&obj) {
    if(this == &obj)
        return *this;
    std::swap(matrix, obj.matrix);
    std::swap(X, obj.X);
    std::swap(Y, obj.Y);
    std::swap(objectsCounter, obj.objectsCounter);
    std::swap(maxObjectsOnField, obj.maxObjectsOnField);
    return *this;
}
```

Файл field.h:

```
#ifndef OOP_1_FIELD_H
#define OOP_1_FIELD_H
#include "Cell.h"
#include "Unit.h"
class Cell;
class Unit;
class Field{
    int X, Y;
    Cell** matrix;
    int objectsCounter;
    int maxObjectsOnField;
public:
    Field();
    Field(int x, int y);
    ~Field();
    int getX() const;
    int getY() const;
    bool checkPoint(int x, int y) const;
    void swapCharacters(int x1, int y1, int x2, int y2);
    int getObjectCounter() const;
    void addObject(Unit* Object, int x, int y);
    void deleteObject(Unit* Object);
    void print() const;
    Field(const Field& obj); // copy constructor
    Field(Field&& obj); // move constructor
    Field& operator=(const Field& obj);
    Field& operator=(Field&& obj);
};
#endif //OOP_1_FIELD_H
```

Файл flamen.cpp:

```
#include "Flamen.h"
Flamen::Flamen(){
    health = 125;
    armor = 0;
```

```
    attack = 15;
    attackRange = 5;
}
char Flamen::getClass() const {
    return 'F';
}
```

Файл flamen.h:

```
#ifndef OOP_1_FLAMEN_H
#define OOP_1_FLAMEN_H
#include "Wizard.h"
class Flamen: public Wizard {
public:
    Flamen();
    char getClass() const;
};
#endif //OOP_1_FLAMEN_H
```

Файл FieldIterator.cpp:

```
#include "FieldIterator.h"
#include "Unit.h"
#include "Field.h"


Iterator::Iterator(const Field& field) : gameField(field) {
    this->i = 0;
    this->j = 0;
    this->width = field.getSize().width;
    this->height = field.getSize().height;
}


bool Iterator::hasNext() const
{
    return i < gameField.getSize().height && j < gameField.getSize().width;

}


void Iterator::first()
{
    j = 0;
    i = 0;
}


const Iterator& operator++(Iterator& it)
{
    if (it.j + 1 < it.width) {
        ++it.j;
    }
    else {
        ++it.i;
        it.j = 0;
```

```
    }

    return it;
}

const Iterator& operator--(Iterator& it)
{
    if (it.j - 1 >= 0) {
        --it.j;
    }
    else {
        --it.i;
        it.j = it.width - 1;
    }

    return it;
}


Unit* Iterator::operator*() const
{
    return ((gameField.GetHead())[i * this->width + j]);
}
```

Файл FieldIterator.h:

```
#include <cstdio>


class Field;
class Unit;

class Iterator
{
public:
    explicit Iterator(const Field& field);

    bool hasNext() const;
    void first();
    friend const Iterator& operator--(Iterator& it);
    friend const Iterator& operator++(Iterator& it);
    Unit* operator*() const;


private:
    size_t i;
    size_t j;
    size_t width;
    size_t height;
    const Field& gameField;
};
```

Файл Field.cpp:

```
#include <iostream>
#include "Field.h"
#include "Unit.h"


Field::~Field() {
   for (int i = 0; i < size.width * size.height; ++i)
```

```
      delete head[i];

   delete[] head;
}


Field::Field(const Field& field) {
    if (this != &field) {

        this->maxQuantity = field.maxQuantity;

        this->currentQuantity = field.currentQuantity;

        this->size = field.size;

        delete[] head;

        this->head = new Unit *[field.size.width * field.size.height];
        for (int i = 0; i < this->size.width * this->size.height; ++i) {
            head[i] = nullptr;

            for (int j = 0; j < field.size.width * field.size.height; ++j) {
                if (!field.head[j])
                    continue;
                else
                    this->head[j] = field.head[j]->clone();
            }
        }
    }
}


Field::Field(int width, int height) {
    if ((width <= 0) || (height <= 0)){
        std::cout << "Empty/Negative size field cannot be created\n";
        return;
    }

   this->head = new Unit*[width * height];
   this->size.width = width;
   this->size.height = height;

   for (int i = 0; i < this->size.width * this->size.height; ++i) {
      head[i] = nullptr;
   }

}


Field::Field(Field&& other) noexcept {

    this->currentQuantity = other.currentQuantity;
    this->size = other.size;
    this->maxQuantity = other.maxQuantity;

   this->head = other.head;
   other.head = nullptr;

}


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

   if (this != &field) {
```

```
        this->maxQuantity = field.maxQuantity;

        this->currentQuantity = field.currentQuantity;

        this->size = field.size;

        delete[] head;

        this->head = new Unit* [field.size.width * field.size.height];

        for (int i = 0; i < field.size.width * field.size.height; ++i)
              this->head[i] = field.head[i]->clone();

    }
    return *this;
}


bool Field::moveUnit(int xPos, int yPos, int dX, int dY) {
    if (((dX > (this->getSize().width - 1) - xPos) || (dY > (this->getSize().height - 1) -
yPos))
        || ((xPos + dX < 0) || (yPos + dY < 0))) {
        std::cout << "Cant step out of field's borders\n";
        return false;
    }

    if (head[yPos * size.width + xPos] == nullptr ||
        head[(yPos + dY) * size.width + xPos + dX] != nullptr) {
        std::cout << "Can't move non-existent object / Can't move to a taken position\n";
        return false;
    }

    head[(yPos + dY) * size.width + xPos + dX] = head[yPos * size.width + xPos];
    head[yPos * size.width + xPos] = nullptr;
    return true;
}


void Field::addObj(Unit* object, int xPos, int yPos) {
    if(object == nullptr) {
        std::cout << "If you want to delete an object use removeObject instead\n";
        return;
    }
    if (currentQuantity + 1 <= maxQuantity) {
        if (head[size.width * yPos + xPos] == nullptr) {
            head[size.width * yPos + xPos] = object;
            ++currentQuantity;
            ((Unit *) object)->setPosition(xPos, yPos);
        }
        else {
            std::cout << "This position is already taken by another unit\n";
            return;
        }
    }
    else {
        std::cout << "Field contains maximum quantity of objects\n";
    }

}


void Field::removeObj(Location pos) {
    if(head[(pos.getPosition().second - 1) * this->size.width + pos.getPosition().first] ==
```

```
nullptr){
        std::cout << "Invalid position of unit\n";
        return;
    }
   head[(pos.getPosition().second - 1) * this->size.width + pos.getPosition().first] =
nullptr;
   --currentQuantity;
}


f_size Field::getSize() const {
   return this->size;
}


int Field::getMax() const {
   return this->maxQuantity;
}


void Field::setLimit(int lim) {
    if(lim < 0) {
        std::cout << "Field cannot contain negative quantity of objects\n";
    }
   this->maxQuantity = lim;
}


int Field::getQuantity() const {
   return this->currentQuantity;
}

Unit **Field::GetHead() const {
    return this->head;
}


Iterator* Field::getIterator() const {
    return (new Iterator(*this));
}
```

Файл Field.h:

```
#pragma once
#include <variant>
#include <utility>
#include <memory>
#include "Unit.h"
#include "FieldIterator.h"
#include <cstring>

typedef struct fieldSize {
   int width, height;

   fieldSize(int width, int height) {
      this->width = width;
      this->height = height;
   }

   fieldSize(const fieldSize &f) {
      this->width = f.width;
      this->height = f.height;
   }
```

```
    fieldSize() = default;

}f_size;


class Field
{
public:

    Field() = default;

    ~Field();

    Field(const Field &f);

    Field(int width, int height);

    Field(Field&& other) noexcept;

    Field& operator=(const Field &field);

    bool moveUnit(int xPos, int yPos, int dX, int dY);

    void addObj(Unit* object, int xPos, int yPos);

    void removeObj(Location pos);

    f_size getSize() const;

    int getMax() const;

    void setLimit(int lim);

    int getQuantity() const;

    Unit** GetHead() const;

     Iterator* getIterator() const;


private:
    f_size size = { 0, 0 };

    Unit** head = nullptr;

    int maxQuantity = 15;

    int currentQuantity = 0;

};
```

Файл mian.cpp:

```
#include <iostream>
#include "Unit.h"
#include "Field.h"
#include "FieldIterator.h"

void testCopyConstructor(Field fi){

};
```

```
int main()
{

   Field field(10, 10);

   auto* archer = new NoviceArcher;
   auto* paladin = new Paladin;


   field.addObj((Unit*) archer, 1, 1);
   field.addObj((Unit*) paladin, 2, 2);

   archer->attack(&field, paladin);

   paladin->move(&field, -1, -1);

    Iterator it(field);

    it.first();

    ++it;
    --it;

    auto tmp = (*it);

    testCopyConstructor(field);

   return 0;
}
```