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

Студент гр. 2300 _____ Жохов К.С.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург

2023

Цель работы.

Изучить базовые принципы объектно-ориентированной парадигмы программирования. Познакомиться с понятием класса. Разработать класс игрока, содержащий некоторые характеристики и методы для работы с ними. Также разработать класс, ответственный за передвижение игрока по игровому полю и осуществляющий некоторую работу с его характеристиками.

Задание.

а) Создать класс игрока. У игрока должны быть поля, которые определяют его характеристики, например кол-во жизней, очков и.т.д. Также в классе игрока необходимо реализовать ряд методов для работы с его характеристиками. Данные методы должны контролировать значения характеристик (делать проверку на диапазон значений).

б) Создать класс, передвигающий игрока по полю и работу с характеристиками. Данный класс всегда должен знать об объекте игрока, которым управляет, но не создавать класс игрока. В следующих лаб. работах данный класс будет проводить проверку, может ли игрок совершить перемещение по карте.

Выполнение работы.

1. Класс `Player`. Представляет собой модель игрока, информация о жизненной способности и игровом счёте игрока содержится в приватных полях health и score соответственно (приватность полей не даёт пользователю менять их значение без дополнительной проверки).

- Метод `checkOption()`. Модификатор доступа – `private (т.к. метод требуется лишь для проверки потенциальных значений полей класса)`. Принимает на вход один аргумент (потенциальное значение поля), проверяет его валидность, возвращает тип `bool`.
- Конструктор `Player()`. Модификатор доступа – `public`. Принимает на вход необязательные параметры (потенциальные значения полей `health` и

score), проверяет их валидность. Если аргументы имеют неприемлемые значения, бросается ошибка std::invalid_argument, в противном случае поля будут инициализированы, переданными аргументами.

- Метод getHealth(). Модификатор доступа – public. Не принимает аргументов. Возвращает информацию о жизненной способности игрока.
- Метод getScore(). Модификатор доступа – public. Не принимает аргументов. Возвращается информацию о количестве очков игрока.
- Метод setHealth(). Модификатор доступа – public. Принимает необязательный аргумент, проверяет: входит ли он в подходящий диапазон значений. В зависимости от результата проверки бросается ошибка или устанавливается новое значение поля health.
- Метод setScore(). Модификатор доступа – public. Принимает необязательный аргумент, проверяет: входит ли он в подходящий диапазон значений. В зависимости от результата проверки бросается ошибка или устанавливается новое значение поля score.

2. enum class Direction. Класс перечислений, требуемый для задания направления движения.

3. Класс Controller. Ответственен за передвижения игрока по игровому полю, способен изменять характеристики игрока. Содержит два приватных поля: ссылку на объект игрока и координаты игрока (приватность полей не даёт пользователю менять их значения без дополнительной проверки).

- Метод checkCoordinates(). Модификатор доступа - private (т.к. метод используется лишь для проверки потенциального значения поля класса). Принимает на вход две потенциально новые координаты игрока, проверяет: содержатся ли они на игровом поле. В зависимости от результата проверки возвращает тип bool.
- Конструктор Controller(). Модификатор доступа – public. Принимает на вход ссылку на объект игрока и ещё два необязательных аргумента (координаты объекта игрока). Проверяет валидность, полученных координат, и в

зависимости от результата проверки бросается ошибка или инициализируются поля.

- Метод `setCoordinates()`. Модификатор доступа – `public`. Принимает на вход два необязательный аргумента (новые координаты). Проверяется валидность координат. Бросается ошибка или устанавливаются новые координаты.
- Метод `getCoordinates()`. Модификатор доступа – `public`. Не принимает аргументов. Возвращает значения поля, содержащего координаты.
- Метод `changeCoordinates()`. Модификатор доступа – `public`. Принимает два необязательных аргумента (изменение по каждой из координат). Вызывает метод `setCoordinates()`, подав ему координаты с учётом желаемого изменения.
- Метод `changeHealth()`. Модификатор доступа – `public`. Принимает необязательный аргумент (величина изменения единиц здоровья). Вызывает у объекта класса игрока метод `setHealth()`, подав в качестве аргумента величину здоровья с учётом желаемого изменения.
- Метод `changeScore()`. Модификатор доступа – `public`. Принимает необязательный аргумент (величина изменения очков). Вызывает у объекта класса игрока метод `setScore()`, передав в качестве аргумента количество очков с учётом желаемых изменений.
- Метод `move()`. Модификатор доступа – `public`. Принимает на вход одну из констант класса перечислений `Direction`. В зависимости от значения аргумента произойдёт перемещение игрока на игровом поле (изменение одной из координат).

Makefile. С целью сделать код более читаемым и для удобства модификации программа была разделена на смысловые части. Для сборки программы написан Makefile.

Разработанный программный код см. в приложении А.

Тестирование.

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

№ п/п	Входные данные	Выходные данные	Комментарии

1.	Player pl;
	pl.setHealth(92);
	pl.setScore(85);
	std::cout << pl.getHealth() << '\t' << pl.getScore() << '\n';
		92 85	Устанавливаются значения полей игрока

2.	Player pl; pl.setHealth(101);	Health should be from 0 to 100!	Попытка установить недопустимое значение поля

3.	Player pl;
	Player& my_pl = pl;
	Controller ctrl(my_pl, 5, 7);
	ctrl.move(Direction::up);
	std::pair<int, int> crds = ctrl.getCoordinates();
	std::cout << crds.first << '\t' << crds.second << '\n';
		5 8	Проверка основополагающего метода передвижения игрока

4.	Player pl;
	Player& my_pl = pl;
	Controller ctrl(my_pl, 5, 7);
	ctrl.changeCoordinates(-2, -4);
	std::pair<int, int> crds = ctrl.getCoordinates();
	std::cout << crds.first << '\t' << crds.second << '\n';
		3 3	Проверка метода, изменяющего координаты

5.	Player pl; Player& my_pl = pl; Controller ctrl(my_pl, 5, 7);	10 12	Проверка метода, устанавливающего координаты

	ctrl.setCoordinates(10, 12);
	std::pair<int, int> crds = ctrl.getCoordinates();
	std::cout << crds.first << '\t' << crds.second << '\n';		

6.	Player pl;
	pl.setHealth(98);
	pl.setScore(0);
	//std::cout << pl.getHealth() << '\t' << pl.getScore() << '\n';
	Player& my_pl = pl;
	Controller ctrl(my_pl, 5, 7);
	//ctrl.move(Direction::up);
	ctrl.changeHealth(-9);
	ctrl.changeScore(45);
	std::cout << pl.getHealth() << '\t' << pl.getScore() << '\n';
		89 45	Проверка метода, методов, изменяющих характеристики игрока

Выводы.

Получены знания об общих принципах и понятиях объектно-ориентированной парадигмы программирования. Разработан класс игрока, имеющий несколько характеристик и методы для работы с ними, а также класс, имитирующий управление передвижением игрока по игровому полю и осуществляющий желаемые (но допустимые) пользователем изменения характеристик игрока.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: Player.hpp

```
#ifndef PLAYER
#define PLAYER

#include <iostream>

class Player
{
    int health;
    int score;          // случайное кол-во очков случайным образом разбросаны по полю, игрок получает очки, перейдя в клетку; max 100 очков отражает полноту прохождения уровня

    bool checkOption(int value); // функция проверки валидности полей здоровья или очков

public:
    Player(int hp=100, int points=0);           //unsigned int x, unsigned int y

    int getHealth();

    int getScore();

    void setHealth(int value=0);

    void setScore(int value=0);
};

#endif
```

Название файла: Player.cpp

```
#include "Player.hpp"

bool Player::checkOption(int value)
{
    if (value >= 0 && value <= 100)
        return true;
    return false;
}

Player::Player(int hp, int points)
{
    if (hp < 0 || hp > 100 || points < 0 || points > 100) // maybe used checkOption
        throw std::invalid_argument("Incorrect arguments!\n");
    health = hp;
```

```
        score = points;
    }

    int Player::getHealth()
    {
        return health;
    }

    int Player::getScore()
    {
        return score;
    }

    void Player::setHealth(int value)
    {
        if (!checkOption(value))
            throw  std::invalid_argument("Health  should  be  from  0  to
100!\n");
        health = value;
    }

    void Player::setScore(int value)
    {
        if (!checkOption(value))
            throw  std::invalid_argument("Score  should  be  from  0  to
100!\n");
        score = value;
    }
```

Название файла: Controller.hpp

```
#ifndef CONTROLLER
#define CONTROLLER


#include "Player.hpp"


enum class Direction {up, down, left, right};


class Controller
{
    Player& player;
    std::pair <int, int> coordinates;


    bool checkCoordinates(int x, int y);


    public:
        Controller(Player& ref, int x=0, int y=0);


        void setCoordinates(int x=0, int y=0);
```

```
        std::pair<int, int> getCoordinates();

        void changeCoordinates(int delta_x=0, int delta_y=0);

        void changeHealth(int delta_hp=0);

        void changeScore(int delta_pts=0);

        void move(Direction step);
};

#endif
```

Название файла: Controller.cpp

```
#include "Controller.hpp"


bool Controller::checkCoordinates(int x, int y)
{
    if (x >= 0 && y >= 0)       //*Checking the validality of x and y* (в последующих работах, когда будут известны размеры игрового поля, будет разработана проверка)
        return true;
    return false;
}


Controller::Controller(Player& ref, int x, int y):player(ref)
{
    if (x < 0 || y < 0)  // *Checking the validality of x and y* may be used checkCoordinates
        throw std::invalid_argument("Coordinates should be inside the playing field!\n");
    coordinates = std::make_pair(x, y);
}

void Controller::setCoordinates(int x, int y)
```

```
    {
        if (!checkCoordinates(x, y))
            throw std::invalid_argument("Coordinates should be inside
the playing field!\n");
        coordinates = std::make_pair(x, y);
    }

    std::pair<int, int> Controller::getCoordinates()
    {
        return coordinates;
    }

    void Controller::changeCoordinates(int delta_x, int delta_y)
    {
        setCoordinates(coordinates.first + delta_x, coordinates.second
+ delta_y);
    }

    void Controller::changeHealth(int delta_hp)
    {
        player.setHealth(player.getHealth() + delta_hp);
    }

    void Controller::changeScore(int delta_pts)
    {
        player.setScore(player.getScore() + delta_pts);
    }

    void Controller::move(Direction step)
    {
        switch(step)
        {
            case Direction::up:
                changeCoordinates(0, 1);
                break;
            case Direction::down:
                changeCoordinates(0, -1);
                break;
            case Direction::left:
```

```
                changeCoordinates(-1, 0);
                break;
            case Direction::right:
                changeCoordinates(1, 0);
                break;
        }
    }
```

Название файла: main.cpp

```
#include "Player.hpp"
#include "Controller.hpp"

int main()
{
    try
    {
        Player pl;
        pl.setHealth(98);
        pl.setScore(0);
        //std::cout << pl.getHealth() << '\t' << pl.getScore() << '\n';
        Player& my_pl = pl;
        Controller ctrl(my_pl, 5, 7);
        //ctrl.move(Direction::up);

        ctrl.changeHealth(-9);
        ctrl.changeScore(45);
        std::cout << pl.getHealth() << '\t' << pl.getScore() << '\n';
        //ctrl.changeCoordinates(-2, -4);
        //ctrl.setCoordinates(10, 12);
        //std::pair<int, int> crds = ctrl.getCoordinates();
        //std::cout << crds.first << '\t' << crds.second << '\n';
    }
    catch(std::invalid_argument& err)
    {
        std::cout << err.what();
    }
```

```
    return 0;
}
```

Название файла: Makefile

```
lb1: main.o Player.o Controller.o
	g++ main.o Player.o Controller.o -o lb1

main.o: main.cpp Player.hpp Controller.hpp
	g++ -c main.cpp

Player.o: Player.cpp Player.hpp
	g++ -c Player.cpp

Controller.o: Controller.cpp Controller.hpp Player.hpp
	g++ -c Controller.cpp

clean:
	rm *.o
```