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. 2383 _____ Борисов И.В.

Преподаватель _____ Сорокумов С.В.

Санкт-Петербург

2023

Цель работы.

Целью данной лабораторной работы является создание класса игрока, создание удобного интерфейса взаимодействия с ним, а также создание классов, которые будут иметь контроль над игроком, следить за его передвижением по карте и пр.

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.

Примечания:

- Не забывайте для полей и методов определять модификатор доступа
- Для указания направления движения можно использовать перечисление enum или дополнительную систему классов. Использования чисел или строк является для указания направления является плохой практикой
- Делать отдельный метод под каждое направление делает класс перегруженным, и в будущем ограничивает масштабирование класса

Основные теоретические положения.

Наследование от виртуального класса в программировании означает создание нового класса, который наследует свойства и методы от виртуального (абстрактного) класса. Виртуальный класс - это класс, который содержит хотя бы одну виртуальную функцию. Виртуальная функция - это функция, которая может быть переопределена в производных классах.

Singleton class - это класс, который позволяет создать только один свой экземпляр и предоставляет глобальную точку доступа к этому экземпляру. Это шаблон проектирования, обеспечивающий создание только одного экземпляра класса и предоставляющий возможность доступа к нему из любой точки программы. Паттерн singleton обычно используется в сценариях, когда необходимо ограничить инстанцирование класса одним объектом, например, для управления общим ресурсом или поддержания глобального состояния.

В языке C++ виртуальный деструктор используется для обеспечения корректного уничтожения объектов при удалении объекта производного класса по указателю типа базового класса. Если базовый класс имеет невиртуальный деструктор, а объект производного класса удаляется по указателю базового класса, то это приводит к неопределенному поведению.

Выполнение работы.

Изначально был создан интерфейс класса сущности *EntityInterface*. В дальнейшем от данного интерфейса будут наследоваться все типы сущностей: игрок, а также все типы ботов. У данного класса есть ряд виртуальных методов (геттеры и сеттеры) которые позволяют взаимодействовать с данными класса, такими как *health*, *armor, attack*, *defence*. Все методы являются pure virtual и обязаны быть переопределены. Также у всех геттеров присутствует атрибут *[[nodiscard]]* который показывает что возвращаемое значение не должно быть проигнорированно.

Класс *Player* наследуется от интерфейса *EntityInterface* и перегружает методы получения и установки параметров при помощи ключевого слова override. Также у класса игрока есть конструкторы копирования, перемещения, а также соответствующие им операторы присваивания. При помощи функции *void adjust(int32_t &value, int32_t limit);* значение по value приводится к диапозону значений [0;limit]. Таким образом контролируется валидность параметров игрока.

Создан класс *Direction* с ипользованием паттерна Singleton class. Данный класс отвечает за вычисление новой позиции при перемещении сущности в данном направлении функция для вычисления новой позиции *Position calculate_position(const Position &old, DIRECTION direction);*. Возможные направления перечисленны в enum *DIRECTION*. Для избегания копирования кода и упрощения дальнешего развития программы изменение позиции происходит без условие, т.е. к старому вектору прибавляется вектор соответствующий данному направлению. Таким образом для вычисления новой позиции достаточно сложить два вектора.

Созданы удобные псевдонимы для класса *Vector2<int32_t>*для лучшей читаемости кода. Псевдонимы описаны в файле Aliases.hpp.

Так как позиция состоит из 2-х чисел, то было решено создать шаблонный класс Vector2, который содержит 2 числа и реализует интерфейс работы с вектором, перегружены операторы математических операций, а также сравнения. Так как данный класс является шаблонным, то реализация функий не отведена в отдельный .cpp файл.

Наконец был создан класс *PlayerHandler*, который отвечает за перемещение игрока по карте. Данный класс в конструкторе принимает указатель на класс игрока, если этот указатель невалидный (nullptr), то выбрасывается стандартное исключение *std::invalid_argument("Nullptr passed to PlayerHandler").* В деструкторе данный класс удаляет переданный ему во владение класс игрока, потому передаваемый класс игрока обязан быть

выделен в кучи при помощи оператора new. Помимо реализации доступа к полям класса имеется функция *void move_by_direction(DIRECTION direction, int32_t multiplier)* которая отвечает за перемещение игрока по карте в данном направлении на multiplier клеток. В будущем метод будет выполнять проверку на проходимость клетки на карте.

Для тестирования программы были использованы гугл тесты.

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 player;	Player player; EXPECT_EQ(player.get_health(), 100); EXPECT_EQ(player.get_armor(), 0); EXPECT_EQ(player.get_attack(), 10); EXPECT_EQ(player.get_defense(), 10); EXPECT_EQ(player.get_points(), 0); EXPECT_EQ(player.get_position(), Position(0, 0));	Дефолтный конструктор игрока.
2.	Player player(Position(1, 1), 100, 15);	EXPECT_EQ(player.get_health(), 100); EXPECT_EQ(player.get_armor(), 15); EXPECT_EQ(player.get_attack(), 10);	Использование конструктора с параметрами.

		EXPECT_EQ(player.get_defense(), 10); EXPECT_EQ(player.get_points(), 0); EXPECT_EQ(player.get_position(), Position(1, 1));	

3.	Player player;
	
	player.set_position(Position(10, 10));
	
	player.set_points(500);
	
	player.set_points(-1);
		EXPECT_EQ(player.get_position(), Position(0, 0));
		EXPECT_EQ(player.get_points(), 0);
		EXPECT_EQ(player.get_position(), Position(10, 10));
		EXPECT_EQ(player.get_points(), 500);
		player.set_points(-1);
		EXPECT_EQ(player.get_points(), 0);
			Проверка setter методов.

Выводы.

Были изучены основные управляющие конструкции языка C++ для программирования в объектно ориентированном стиле, создан класса игрока с интерфейсом взаимодействия с ним, а также создан классов, который будет иметь контроль над игроком, отвечать за его перемещение.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: stc/Entities/Interface/EntityInterface.hpp

```
#pragma once
#include "../../Math/Vector2.hpp"
#include "../../Movement/Aliases.hpp"

class EntityInterface {
 public:
  virtual ~EntityInterface() = default;

  [[nodiscard]] virtual Position get_position() const = 0;
  [[nodiscard]] virtual int32_t get_health() const = 0;
  [[nodiscard]] virtual int32_t get_armor() const = 0;
  [[nodiscard]] virtual int32_t get_attack() const = 0;
  [[nodiscard]] virtual int32_t get_defense() const = 0;

  virtual void set_position(const Position &new_position) = 0;
  virtual void set_health(int32_t new_value) = 0;
  virtual void set_armor(int32_t new_value) = 0;
  virtual void set_attack(int32_t new_value) = 0;
  virtual void set_defense(int32_t new_value) = 0;
};
```

Название файла: stc/Entities/Player/Player.hpp

```
#pragma once
#include "../Interface/EntityInterface.hpp"

class Player : public EntityInterface {
 private:
  const int32_t health_lim_ = 200;
  const int32_t armor_lim_ = 100;
  const int32_t attack_lim_ = 50;
  const int32_t defence_lim_ = 50;
  const int32_t points_lim_ = 1000;

  Position position_;

  int32_t health_;
  int32_t armor_;
  int32_t attack_;
  int32_t defence_;
  int32_t points_;

  void adjust(int32_t &value, int32_t limit);
 public:
  Player(const Player &player);
```

```
  Player(Player &&player) noexcept;
  Player &operator=(const Player &player);
  Player &operator=(Player &&player) noexcept;

  explicit Player(const Position &position = Position(0, 0),
                  int32_t health = 100,
                  int32_t armor = 0,
                  int32_t attack = 10,
                  int32_t defence = 10,
                  int32_t points = 0);

  [[nodiscard]] Position get_position() const override;
  [[nodiscard]] int32_t get_health() const override;
  [[nodiscard]] int32_t get_armor() const override;
  [[nodiscard]] int32_t get_attack() const override;
  [[nodiscard]] int32_t get_defense() const override;
  [[nodiscard]] int32_t get_points() const;

  void set_position(const Position &new_value) override;
  void set_health(int32_t new_value) override;
  void set_armor(int32_t new_value) override;
  void set_attack(int32_t new_value) override;
  void set_defense(int32_t new_value) override;
  void set_points(int32_t new_value);
};
```

Название файла: stc/Entities/Player/Player.cpp

```
#include "Player.hpp"

Player::Player(const Position &position,
               int32_t health,
               int32_t armor,
               int32_t attack,
               int32_t defence,
               int32_t points) :
               position_(position),
               health_(health),
               armor_(armor),
               attack_(attack),
               defence_(defence),
               points_(points)
{
}
Player::Player(const Player &player)
{
     *this = player;
}
Player::Player(Player &&player) noexcept
{
     *this = std::move(player);
}
Player &Player::operator=(const Player &player)
{
     if (this != &player)
```

```
	{
		position_ = player.position_;
		health_ = player.health_;
		armor_ = player.armor_;
		attack_ = player.attack_;
		defence_ = player.defence_;
		points_ = player.points_;
	}

	return *this;
}
Player &Player::operator=(Player &&player) noexcept
{
	if (this != &player)
	{
		position_ = std::move(player.position_);
		health_ = std::move(player.health_);
		armor_ = std::move(player.armor_);
		attack_ = std::move(player.attack_);
		defence_ = std::move(player.defence_);
		points_ = std::move(player.points_);
	}

	return *this;
}
Position Player::get_position() const
{
	return position_;
}
int32_t Player::get_health() const
{
	return health_;
}
int32_t Player::get_armor() const
{
	return armor_;
}
int32_t Player::get_attack() const
{
	return attack_;
}
int32_t Player::get_defense() const
{
	return defence_;
}
int32_t Player::get_points() const
{
	return points_;
}
void Player::set_position(const Position &new_value)
{
	position_ = new_value;
}
void Player::set_health(int32_t new_value)
{
```

```
        health_ = new_value;
        adjust(health_, health_lim_);
    }
    void Player::set_armor(int32_t new_value)
    {
        armor_ = new_value;
        adjust(armor_, armor_lim_);
    }
    void Player::set_attack(int32_t new_value)
    {
        attack_ = new_value;
        adjust(attack_, attack_lim_);
    }
    void Player::set_defense(int32_t new_value)
    {
        defence_ = new_value;
        adjust(defence_, defence_lim_);
    }
    void Player::set_points(int32_t new_value)
    {
        points_ = new_value;
        adjust(points_, points_lim_);
    }
    void Player::adjust(int32_t &value, int32_t limit)
    {
        value = std::max(0, value);
        value = std::min(value, limit);
    }
```

Название файла: stc/Handlers/PlayerHandler/PlayerHandler.hpp

```
    #pragma once
    #include "../../Entities/Player/Player.hpp"
    #include "../../Movement/Direction.hpp"

    class PlayerHandler {
     private:
      Player *player_;
     public:
      ~PlayerHandler();
      PlayerHandler() = delete;
      explicit PlayerHandler(Player *player);

      [[nodiscard]] Position get_position() const;
      [[nodiscard]] int32_t get_health() const;
      [[nodiscard]] int32_t get_armor() const;
      [[nodiscard]] int32_t get_attack() const;
      [[nodiscard]] int32_t get_defense() const;
      [[nodiscard]] int32_t get_points() const;

          void   move_by_direction(DIRECTION   direction,   int32_t
multiplier);
      void set_position(const Position &new_value);
```

```
  void set_health(int32_t new_value);
  void set_attack(int32_t new_value);
  void set_defense(int32_t new_value);
  void set_points(int32_t new_value);
};
```

Название файла: stc/Handlers/PlayerHandler/PlayerHandler.cpp

```
#include "PlayerHandler.hpp"

PlayerHandler::PlayerHandler(Player *player) : player_(player)
{
	if (player_ == nullptr)
	{
		throw	std::invalid_argument("Nullptr	passed	to
PlayerHandler");
	}
}
PlayerHandler::~PlayerHandler()
{
	delete player_;
}
Position PlayerHandler::get_position() const
{
	return player_->get_position();
}
int32_t PlayerHandler::get_health() const
{
	return player_->get_health();
}
int32_t PlayerHandler::get_armor() const
{
	return player_->get_armor();
}
int32_t PlayerHandler::get_attack() const
{
	return player_->get_attack();
}
int32_t PlayerHandler::get_defense() const
{
	return player_->get_defense();
}
int32_t PlayerHandler::get_points() const
{
	return player_->get_points();
}
void PlayerHandler::set_position(int32_t new_value)
{
	player_->set_position(new_value);
}
void PlayerHandler::set_health(int32_t new_value)
{
	player_->set_health(new_value);
```

```
}
void PlayerHandler::set_attack(int32_t new_value)
{
     player_->set_attack(new_value);
}
void PlayerHandler::set_defense(int32_t new_value)
{
     player_->set_defense(new_value);
}
void PlayerHandler::set_points(int32_t new_value)
{
     player_->set_points(new_value);
}
void    PlayerHandler::move_by_direction(DIRECTION    direction,
int32_t multiplier)
{
     for (int32_t i = 0; i < multiplier; ++i)
     {

set_position(Direction::getInstance().calculate_position(player_-
>get_position(), direction));
     }
}
```

Название файла: stc/Math/Vector2.hpp

```
#pragma once
#include <algorithm>

template<typename T>
class Vector2 {
 private:
  T x_, y_;
 public:
  // getters/setters
  T get_y() const;
  T get_x() const;

  void set_y(const T &y);
  void set_x(T x);

  // constructors
  Vector2(const T &ix, const T &iy);
  Vector2(const Vector2 &other);
  Vector2();

  // operators
  Vector2 operator+(const Vector2 &other) const;
  Vector2 operator-(const Vector2 &other) const;
  Vector2 operator*(const Vector2 &other) const;
  Vector2 operator/(const Vector2 &other) const;
  Vector2 &operator+=(const Vector2 &other);
  Vector2 &operator-=(const Vector2 &other);
```

```
  Vector2 &operator*=(const Vector2 &other);
  Vector2 &operator/=(const Vector2 &other);
  Vector2 &operator=(const Vector2 &other);
  Vector2 &operator=(Vector2 &&other) noexcept;
  bool operator==(const Vector2 &other) const;
  bool operator<(const Vector2 &other) const;
};

template<typename T>
Vector2<T>::Vector2(const T &ix, const T &iy): x_(ix), y_(iy)
{
}
template<typename T>
Vector2<T>::Vector2(): Vector2(0, 0)
{
}
template<typename T>
Vector2<T>::Vector2(const Vector2 &other)
{
     *this = other;
}
template<typename T>
Vector2<T> Vector2<T>::operator+(const Vector2<T> &other) const
{
     Vector2<T> temp(*this);
     temp += other;
     return temp;
}
template<typename T>
Vector2<T> Vector2<T>::operator-(const Vector2<T> &other) const
{
     Vector2<T> temp(*this);
     temp -= other;
     return temp;
}
template<typename T>
Vector2<T> Vector2<T>::operator*(const Vector2<T> &other) const
{
     Vector2<T> temp(*this);
     temp *= other;
     return temp;
}
template<typename T>
Vector2<T> Vector2<T>::operator/(const Vector2<T> &other) const
{
     Vector2<T> temp(*this);
     temp /= other;
     return temp;
}
template<typename T>
Vector2<T> &Vector2<T>::operator+=(const Vector2<T> &other)
{
     x_ += other.x_;
     y_ += other.y_;
     return *this;
```

```
}
template<typename T>
Vector2<T> &Vector2<T>::operator-=(const Vector2<T> &other)
{
	x_ -= other.x_;
	y_ -= other.y_;
	return *this;
}
template<typename T>
Vector2<T> &Vector2<T>::operator*=(const Vector2<T> &other)
{
	x_ *= other.x_;
	y_ *= other.y_;
	return *this;
}
template<typename T>
Vector2<T> &Vector2<T>::operator/=(const Vector2<T> &other)
{
	if (other.x_ != 0 && other.y_ != 0)
	{
		x_ /= other.x_;
		y_ /= other.y_;
	}
	return *this;
}
template<typename T>
T Vector2<T>::get_y() const
{
	return y_;
}
template<typename T>
void Vector2<T>::set_y(const T &y)
{
	y_ = y;
}
template<typename T>
T Vector2<T>::get_x() const
{
	return x_;
}
template<typename T>
void Vector2<T>::set_x(T x)
{
	x_ = x;
}
template<typename T>
Vector2<T> &Vector2<T>::operator=(const Vector2 &other)
{
	if (this != &other)
	{
		x_ = other.x_;
		y_ = other.y_;
	}
	return *this;
}
```

```
template<typename T>
Vector2<T> &Vector2<T>::operator=(Vector2 &&other) noexcept
{
     if (this != &other)
     {
          x_ = std::move(other.x_);
          y_ = std::move(other.y_);
     }
     return *this;
}
template<typename T>
bool Vector2<T>::operator==(const Vector2 &other) const
{
     return x_ == other.x_ && y_ == other.y_;
}
template<typename T>
bool Vector2<T>::operator<(const Vector2 &other) const
{
     if (x_ == other.x_)
     {
          return y_ < other.y_;
     }
     return x_ < other.x_;
}
```

Название файла: stc/Movement/Aliases.hpp

```
#pragma once
#include "../Math/Vector2.hpp"
#include <cstdint>

using Position = Vector2<int32_t>;
using Dimension = Vector2<int32_t>;
```

Название файла: stc/Movement/Direction.hpp

```
#pragma once
#include "../Math/Vector2.hpp"
#include "../Entities/Interface/EntityInterface.hpp"
#include <vector>

enum DIRECTION {
     none,  left_up,  up,  right_up,  right,  right_bottom,  down,
left_down, left
};

class Direction {
 private:
  const std::vector<Vector2<int32_t>> possible_moves_ = {{0,  0},

       {-1, -1},

       {-1, 0},
```

```
            {-1, 1},

            {0,  1},

            {1,  1},

            {1,  0},

            {1,  -1},

            {0,  -1}};
        Direction() = default;
       public:
        Direction(const Direction &) = delete;
        void operator=(const Direction &) = delete;

        static Direction &getInstance()
        {
             static Direction instance;
             return instance;
        }
          Position  calculate_position(const  Position  &old,  DIRECTION
direction);
     };
```

Название файла: stc/Movement/Direction.cpp

```
     #include "Direction.hpp"

     Position   Direction::calculate_position(const   Position   &old,
DIRECTION direction)
     {
          return old + possible_moves_[direction];
     }
```