

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. 3344

Пачев Д.К.

Преподаватель

Жангиров Т.Р.

Санкт-Петербург

2024

Цель работы.

Освоить принципы объектно-ориентированного программирования через создание классов для игры "Морской бой".

Задание.

а. Создать класс корабля, который будет размещаться на игровом поле. Корабль может иметь длину от 1 до 4, а также может быть расположен вертикально или горизонтально. Каждый сегмент корабля может иметь три различных состояния: целый, поврежден, уничтожен. Изначально у корабля все сегменты целые. При нанесении 1 урона по сегменту, он становится поврежденным, а при нанесении 2 урона по сегменту, уничтоженным. Также добавить методы для взаимодействия с кораблем.

б. Создать класс менеджера кораблей, хранящий информацию о кораблях. Данный класс в конструкторе принимает количество кораблей и их размеры, которые нужно расставить на поле.

с. Создать класс игрового поля, которое в конструкторе принимает размеры. У поля должен быть метод, принимающий корабль, координаты, на которые нужно поставить, и его ориентацию на поле. Корабли на поле не могут соприкасаться или пересекаться. Для игрового поля добавить методы для указания того, какая клетка атакуется. При попадании в сегмент корабля изменения должны отображаться в менеджере кораблей.

Каждая клетка игрового поля имеет три статуса:

- i.неизвестно (изначально вражеское поле полностью неизвестно),
- ii.пустая (если на клетке ничего нет)
- iii.корабль (если в клетке находится один из сегментов корабля).

Для класса игрового поля также необходимо реализовать конструкторы копирования и перемещения, а также соответствующие им операторы присваивания.

Примечания:

- Не забывайте для полей и методов определять модификаторы доступа
- Для обозначения переменной, которая принимает небольшое ограниченное количество значений, используйте enum
- Не используйте глобальные переменные

- При реализации копирования нужно выполнять глубокое копирование
- При реализации перемещения, не должно быть лишнего копирования
- При выделении памяти делайте проверку на переданные значения
- У поля не должно быть методов возвращающих указатель на поле в явном виде, так как это небезопасно

Выполнение работы.

- **Класс Cell**

Описание:

Представляет собой клетку на игровом поле в игре. Хранит информацию о своем статусе (пустая или занятая) и связанности с кораблем, если таковой находится в этой клетке. Позволяет управлять состоянием клетки и взаимодействовать с находящимся в ней кораблем.

Поля класса:

1. *bool is_open*: Логическое значение, указывающее, открыта ли клетка (т.е. была ли она атакована).
2. *Status status*: Поле, представляющее статус клетки, который может принимать значения, такие как Empty (пустая), Occupied (занятая).

Для представления статуса клетки используется enum class Status:

```
enum class Status {Empty, Occupied};
```

3. *Ship* pointer_to_ship*: Указатель на объект класса Ship, который находится в данной клетке. Если клетка пустая, указатель равен nullptr.
4. *int index_of_segment*: Индекс сегмента корабля, который находится в этой клетке, если клетка занята. Если клетка пустая, значение обычно игнорируется.

Методы класса:

1. **Конструктор:**

- *Cell()*: Инициализирует клетку и устанавливает значения по умолчанию.

2. *void display()*: Отображает текущий статус клетки в консоль, используя специальные символы для различных состояний (открыта, пустая, поврежденная и т.д.).

3. `void open()`: Устанавливает значение `is_open` в `true`, что позволяет открыть клетку для дальнейшего взаимодействия.
4. **Геттеры и сеттеры**: Предоставляют доступ к полям класса и позволяют изменять статус, указатель на корабль, индекс сегмента и открытость.

- **Класс Ship**

Описание:

Класс `Ship` представляет собой корабль в игре. Он отвечает за хранение информации о длине корабля, его сегментах и их состоянии. Каждый сегмент может быть в одном из трех состояний: целый, поврежденный или разрушенный. Для представления сегментов класса `Ship` используется класс перечисления:

```
enum class SegmentState{FULL, Damaged, Destroyed}
```

Поля класса:

1. `unsigned short length`: Длина корабля, представляющая количество его сегментов.
2. `std::vector<SegmentState> segments`: Вектор, хранящий состояние каждого сегмента корабля.

Методы класса:

1. **Конструктор**: Инициализирует корабль заданной длины. Проверяет корректность длины с помощью метода `checkLength`. Если длина корректна, создает вектор `segments` с размером `length`, и каждый сегмент инициализируется в состоянии `FULL`. Если длина некорректна, выводится сообщение об ошибке, и длина устанавливается в 0.
2. **Методы работы с повреждениями**: `takeDamage(int index)`: Наносит повреждение сегменту корабля по заданному индексу. Если сегмент

целый (FULL), он становится поврежденным (Damaged); если сегмент поврежден, он становится разрушенным (Destroyed).

`bool isShipDestroyed()` : Проверяет, разрушен ли корабль, возвращая `true`, если все сегменты находятся в состоянии `Destroyed`, иначе `false`.

3. Методы доступа к данным: Геттеры используются для получения доступа к полям.

Таким образом, класс `Ship` обеспечивает необходимую логику для управления состоянием корабля, его сегментами и взаимодействием с игровым полем, поддерживая инкапсуляцию и проверку данных.

- **Класс `ShipManager`**

Описание:

Класс `ShipManager` отвечает за управление коллекцией кораблей в игре. Он отвечает за создание, хранение всех кораблей.

Поля класса:

1. `std::vector<Ship> ships`: Вектор, хранящий объекты класса `Ship`, представляющие все корабли, используемые в игре.

Методы класса:

1. **Конструктор:**

- o `ShipManager(int number_of_ships, const std::vector<int> &ship_sizes)`: Инициализирует `ShipManager`, создавая указанное количество кораблей с заданными размерами. Для каждого корабля создается объект класса `Ship`, который добавляется в вектор `ships`.

2. **Геттеры:**

- `std::vector<Ship>& getShips():` Возвращает ссылку на вектор `ships`, позволяя другим частям программы получить доступ к списку кораблей.
- `std::vector<Ship>& getShipByIndex(int index):` Возвращает корабль по указанному индексу. Если `index` выходит за пределы вектора `ships`, возвращается последний корабль из вектора.

Таким образом, класс `ShipManager` хранит в себе информацию о кораблях, и при нанесении урона изменения в нем отображаются.

• Класс `GameField`

Описание:

Класс `GameField` представляет игровое поле в игре. Он отвечает за хранение клеток поля, управление их состоянием, размещение кораблей и выполнение атак.

Поля класса:

1. `int width:` Ширина игрового поля. (ось X)
2. `int height:` Высота игрового поля. (ось Y)
3. `int number_of_deployed_ships:` Количество размещённых кораблей на поле.
4. `std::vector<std::vector<Cell>> field:` Двумерный вектор клеток, представляющий игровое поле.

Методы класса:

1. Конструктор:

- `GameField(int width, int height):` Инициализирует объект поля заданной ширины и высоты, проверяя их корректность с помощью

метода `checkWidthAndHeight`. Каждая клетка поля создается как объект класса `Cell`.

2. Конструктор копирования:

- Реализует глубокое копирование объекта, копируя каждую клетку поля из другого объекта `GameField`.

3. Оператор присваивания копированием:

- Обеспечивает глубокое копирование объекта поля, копируя все клетки поля.

4. Конструктор перемещения:

- Реализует перемещение объектов, передавая владение ресурсами от одного объекта `GameField` к другому.

5. Оператор присваивания перемещением:

- Перемещает ресурсы от одного объекта к другому, аналогично конструктору перемещения.

6. Метод атаки клетки:

- `attackCell(int x, int y)`: Обрабатывает атаку на клетку поля по заданным координатам. Если клетка занята кораблем, вызывает у корабля метод `takeDamage()`.

7. Метод отображения поля:

- `show()`: Отображает текущее состояние игрового поля, выводя клетки с их статусами.

8. Метод размещения корабля:

- `placeShip(Ship &ship, int x, int y, bool is_vertical)`: Размещает корабль на поле, проверяя на пересечения с другими кораблями. В случае пересечения выводит сообщение об ошибке.

9. Проверка столкновений кораблей:

- `check_collide(int x, int y, Ship* pointer_to_ship)`: Проверяет пересечения между двумя кораблями на игровом поле по их координатам. Смотрит, чтобы в области 8 клеток вокруг каждой части корабля не находилась часть другого корабля.

10. Геттеры для ширины и высоты поля:

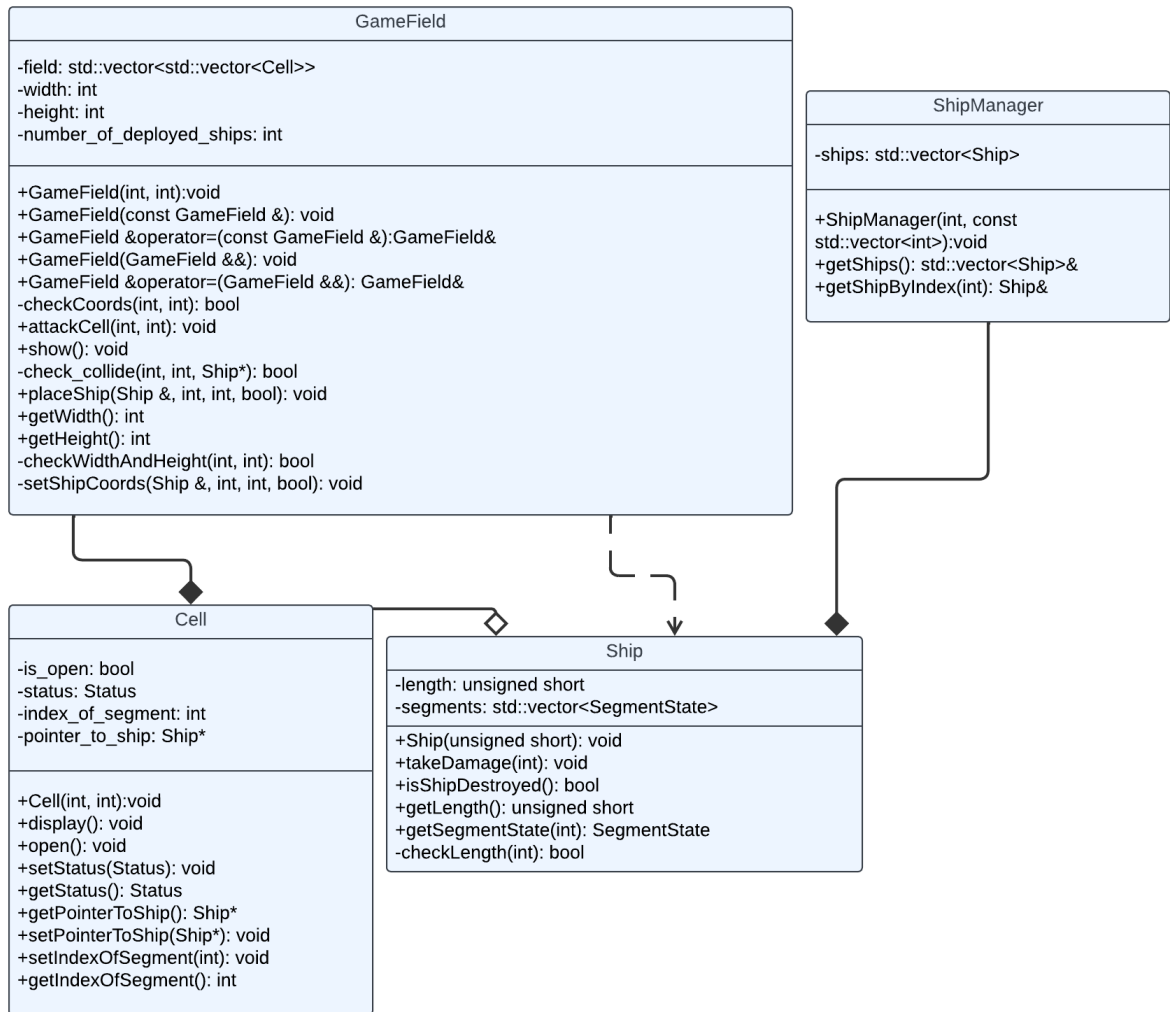
- `int getWidth()` : Возвращает ширину игрового поля.
- `int getHeight()` : Возвращает высоту игрового поля.

11. Методы проверки:

- `bool checkWidthAndHeight(int width, int height)` : Проверяет, что ширина и высота положительны.
- `bool checkCoords(int x, int y)` : Проверяет, что обе координаты находятся в допустимых пределах.

Таким образом, `GameField` является ключевым компонентом архитектуры игры, предоставляющим функциональность для управления клетками, кораблями и взаимодействиями на поле.

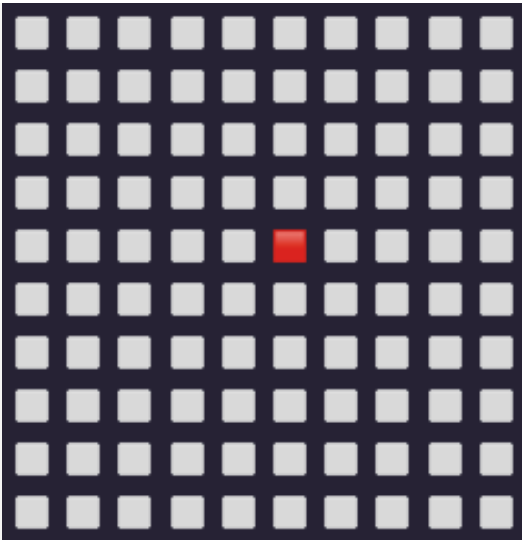
UML – диаграммы классов представлены на картинке ниже

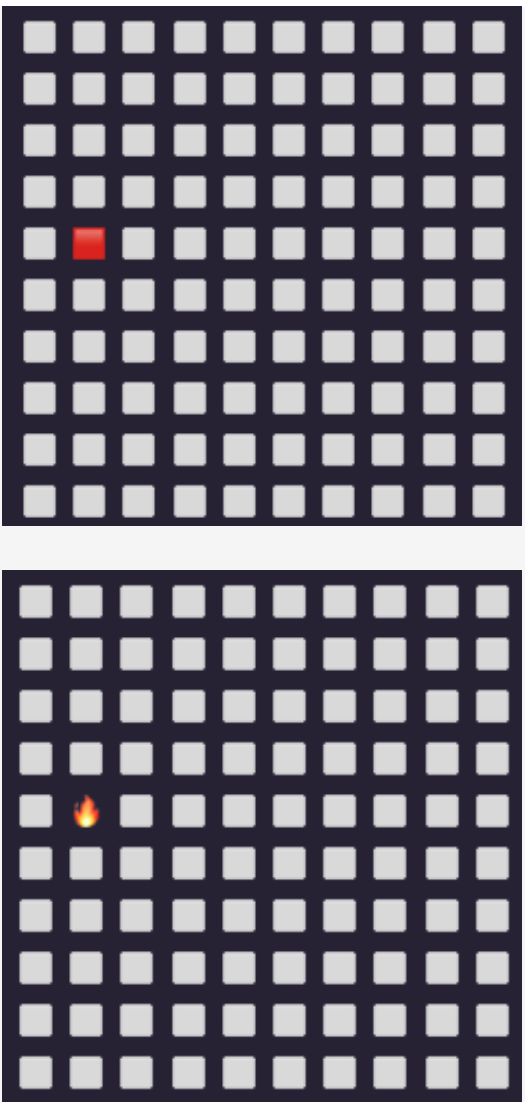


Тестирование

Результаты тестирования представлены в Таблице 1

Таблица 1 - Результаты тестирования

№ п/п	Входные данные	Выходные данные	Комментарии
1.	<pre> GameField field{ 10, 10}; auto ship_manager = ShipManager(3, {3, 1, 2}); int ships_coords[3][2] = {{5, 4},{1, 4},{9, 1}}; auto &ships = ship_manager.getShips() ; bool positions[] = {false, false, true}; for (int i = 0; i < 3; i++) { field.placeShip(ships[i], ships_coords[i][0], ships_coords[i][1], positions[i]); } field.attackCell(5, 4); field.show(); </pre>		Верно
2.	<pre> auto ship_manager = ShipManager(3, {3, 1, 2}); auto &ships = ship_manager.getShips() ; std::cout<<ships[0].getL ength()<<"\n"; </pre>	3	Верно

3.	<pre> GameField field{ 10, 10}; auto ship_manager = ShipManager(3, {3, 1, 2}); int ships_coords[3][2] = {{5, 4},{1, 4},{9, 1}}; auto &ships = ship_manager.getShips() ; auto& s = ship_manager.getShipBy Index(9); bool positions[] = {false, false, true}; for (int i = 0; i < 3; i++) { field.placeShip(ships[i], ships_coords[i][0], ships_coords[i][1], positions[i]); } field.attackCell(1, 4); field.show(); field.attackCell(1, 4); field.show(); </pre>		верно

Исходный код программы см. в Приложении А.

Вывод:

В ходе лабораторной работы были освоены принципы объектно-ориентированного программирования, разработана программа на языке C++, которая позволяет создавать поле для игры в «Морской бой», расставлять корабли и взаимодействовать с ними.

ПРИЛОЖЕНИЕ А.

ИСХОДНЫЙ КОД ПРОГРАММЫ.

Cell.cpp:

```
#include "Cell.h"

Cell::Cell() : is_open(false), status(Status::Empty),
pointer_to_ship(nullptr),
index_of_segment(-1) {}

void Cell::display() {
    if (is_open) {
        if (status == Status::Empty)
            std::cout << "O";
        else {
            if (pointer_to_ship->isShipDestroyed()) {
                std::cout << "□";
                return;
            }
            SegmentState segment_state = pointer_to_ship-
>getSegmentState(index_of_segment);
            if (segment_state == SegmentState::Damaged) {
                std::cout << u8"\U0001F7E5";
            }
            else if (segment_state == SegmentState::FULL){
                std::cout<<"!";
            }
            else {
                std::cout << "X";
            }
        }
    }
    else
        std::cout << "□\uFE0F";
}

void Cell::open() {
    is_open = true;
}

void Cell::setStatus(Status status) {
    this->status = status;
}

Status Cell::getStatus() {
    return status;
}

Ship *Cell::getPointerToShip() {
    return pointer_to_ship;
}

void Cell::setPointerToShip(Ship *pointer_to_ship) {
    this->pointer_to_ship = pointer_to_ship;
}
```

```
void Cell::setIndexOfSegment(int index) {  
    index_of_segment = index;  
}  
  
int Cell::getIndexOfSegment() {  
    return index_of_segment;  
}
```


Cell.h:

```
#ifndef OOP_LAB1_CELL_H
#define OOP_LAB1_CELL_H

#include "Ship.h"

enum class Status {
    Empty, Occupied
};

class Cell {
private:
    bool is_open;
    Status status;
    int index_of_segment;
    Ship *pointer_to_ship;

public:
    Cell();

    void display();

    void open();

    void setStatus(Status status);

    Status getStatus();

    Ship *getPointerToShip();

    void setPointerToShip(Ship *pointer_to_ship);

    void setIndexofSegment(int index);

    int getIndexofSegment();
};

#endif //OOP_LAB1_CELL_H
```

GameField.cpp:

```
#include "GameField.h"

GameField::GameField(int width, int height) {
    if (!checkWidthAndHeight(width, height)) {
        std::cerr << "Width and height of the field must be positive"
        << '\n';
        width = 0;
        height = 0;
    }
    this->width = width;
    this->height = height;
    number_of_deployed_ships = 0;
    field.resize(height, std::vector<Cell>(width));
    for (int i = 0; i < height; ++i) {
        for (int j = 0; j < width; ++j) {
            field[i][j] = Cell();
        }
    }
}

GameField::GameField(const GameField &other)
    : width(other.width), height(other.height),
    number_of_deployed_ships(other.number_of_deployed_ships) {
    field.resize(height);
    for (int i = 0; i < height; ++i) {
        field[i].resize(width);
        for (int j = 0; j < width; ++j) {
            field[i][j] = other.field[i][j];
        }
    }
}

GameField &GameField::operator=(const GameField &other) {
    if (this != &other) {
        width = other.width;
        height = other.height;
        number_of_deployed_ships = other.number_of_deployed_ships;
        field.resize(height);
        for (int i = 0; i < height; ++i) {
            field[i].resize(width);
            for (int j = 0; j < width; ++j) {
                field[i][j] = other.field[i][j];
            }
        }
    }
    return *this;
}

GameField::GameField(GameField &&other)
    : width(other.width), height(other.height),
    number_of_deployed_ships(other.number_of_deployed_ships) {
    field = std::move(other.field);
    other.width = 0;
    other.height = 0;
    other.number_of_deployed_ships = 0;
}
```

```

GameField &GameField::operator=(GameField &&other) {
    if (this != &other) {
        width = other.width;
        height = other.height;
        number_of_deployed_ships = other.number_of_deployed_ships;
        field = std::move(other.field);
        other.width = 0;
        other.height = 0;
        other.number_of_deployed_ships = 0;
    }
    return *this;
}

void GameField::attackCell(int x, int y) {
    if (!checkCoords(x, y)) {
        std::cerr << "Cell can not be attack. One of coords is not
correct" << '\n';
        return;
    }
    field[y][x].open();
    if (field[y][x].getStatus() == Status::Occupied) {
        auto pointer_to_ship = field[y][x].getPointerToShip();
        auto index = field[y][x].getIndexofSegment();
        pointer_to_ship->takeDamage(index);
    }
}

void GameField::show() {
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            field[i][j].display();
            std::cout << ' ';
        }
        std::cout << '\n';
    }
    std::cout << '\n';
}

bool GameField::checkCollide(int x, int y, Ship* pointer_to_ship) {
    int row_above = y-1;
    int row_below = y+1;
    int left_col = x-1;
    int right_col = x+1;
    for (int i = row_above; i < row_below+1; i++) {
        for (int j = left_col; j < right_col+1; j++) {
            if ((j == x && i == y) || (j < 0 || i < 0 || i >= height ||
j >= width))
                continue;
            else {
                if (field[i][j].getStatus() == Status::Occupied &&
pointer_to_ship != field[i][j].getPointerToShip())
                    return false;
            }
        }
    }
    return true;
}

```

```

}

void GameField::setShipCoords(Ship &ship, int x, int y, bool
is_vertical) {
    for (int i = 0; i < ship.getLength(); i++) {
        if (is_vertical) {
            field[y + i][x].setStatus(Status::Occupied);
            field[y + i][x].setPointerToShip(&ship);
            field[y + i][x].setIndexOfSegment(i);
        } else {
            field[y][x + i].setStatus(Status::Occupied);
            field[y][x + i].setPointerToShip(&ship);
            field[y][x + i].setIndexOfSegment(i);
        }
    }
}

}

void GameField::placeShip(Ship &ship, int x, int y, bool is_vertical)
{
    auto ship_length = ship.getLength();
    int x_coord, y_coord;
    for (int i = 0; i < ship_length; i++) {
        if (is_vertical) {
            x_coord = x;
            y_coord = y + i;
        } else {
            x_coord = x + i;
            y_coord = y;
        }
        if (!checkCoords(x_coord, y_coord)) {
            std::cerr << "Ship can not be placed. Coordinates are out of
bounds" << '\n';
            return;
        }
        if (!checkCollide(x_coord, y_coord, &ship)) {
            std::cerr << "Ships collide\n";
            return;
        }
    }
    setShipCoords(ship, x, y, is_vertical);
}

int GameField::getWidth() {
    return width;
}

int GameField::getHeight() {
    return height;
}

bool GameField::checkWidthAndHeight(int width, int height) {
    if (width < 0 || height < 0)
        return false;
    return true;
}

```

```
bool GameField::checkCoords(int x, int y) {  
    if (x < 0 || x >= width || y < 0 || y >= height)  
        return false;  
    return true;  
}
```

GameField.h:

```
#ifndef OOP_LAB1_GAMEFIELD_H
#define OOP_LAB1_GAMEFIELD_H

#include <iostream>
#include "Cell.h"
#include "ShipManager.h"

class GameField {
private:
    std::vector<std::vector<Cell>> field;
    int width;
    int height;
    int number_of_deployed_ships;

    bool checkCollide(int x, int y, Ship* pointer_to_ship);

    bool checkCoords(int x, int y);

    bool checkWidthAndHeight(int width, int height);

    void setShipCoords(Ship &ship, int x, int y, bool is_vertical);

public:
    GameField(int width, int height);

    GameField(const GameField &other);

    GameField &operator=(const GameField &other);

    GameField(GameField &&other);

    GameField &operator=(GameField &&other);

    void attackCell(int x, int y);

    void show();

    void placeShip(Ship &ship, int x, int y, bool is_vertical);

    int getWidth();

    int getHeight();

};
```

```
#endif //OOP_LAB1_GAMEFIELD_H
```

Ship.cpp:

```
#include "Ship.h"

Ship::Ship(unsigned short length) {
    checkLength(length);
    this->length = length;
```

```

        segments.resize(length);
        is_placed = false;
        for (auto &segment: segments) {
            segment.state = 2;
        }
    }

void Ship::checkLength(int length) {
    if (length < 1 || length > 4) {
        throw std::invalid_argument("Invalid ship length");
    }
}

void Ship::takeDamage(int x, int y) {
    for (auto &segment: segments) {
        if (segment.x == x && segment.y == y) {
            if (segment.state > 0)
                segment.state--;
            return;
        }
    }
}

std::vector<std::vector<int>> Ship::getShipCoords() {
    std::vector<std::vector<int>> ship_coords;
    for (auto &segment: segments) {
        ship_coords.push_back({segment.x, segment.y});
    }
    return ship_coords;
}

bool Ship::isShipDestroyed() {
    for (auto &segment: segments) {
        if (segment.state > 0)
            return false;
    }
    return true;
}

unsigned short Ship::getLength() {
    return length;
}

void Ship::checkSegmentIndex(int index) {
    if (index >= segments.size()) {
        throw std::out_of_range("Invalid segment index");
    }
}

int Ship::getSegmentState(int x, int y) {
    for (auto &segment: segments) {
        if (segment.x == x && segment.y == y) {
            return segment.state;
        }
    }
    throw std::invalid_argument("Segment not found");
}

```

```

void Ship::setSegmentCoords(int x, int y, int index) {
    checkSegmentIndex(index);
    segments[index].x = x;
    segments[index].y = y;
}

bool Ship::isPlaced() {
    return is_placed;
}

void Ship::setPlaced(bool is_placed) {
    this->is_placed = is_placed;
}

```

Ship.h:

```

#ifndef OOP_LAB1_SHIP_H
#define OOP_LAB1_SHIP_H

```



```

#include <iostream>

enum class SegmentState{
    FULL, Damaged, Destroyed
};

class Ship {
private:
    unsigned short length;
    std::vector<SegmentState> segments;

    bool checkLength(int length);

public:
    Ship(unsigned short length);

    void takeDamage(int index);

    bool isShipDestroyed();

    unsigned short getLength();

    SegmentState getSegmentState(int index);
};

#endif //OOP_LAB1_SHIP_H

```

ShipManager.cpp:

```

#include "ShipManager.h"

```

```

ShipManager::ShipManager(int number_of_ships, const std::vector<int>
&ship_sizes) {

```

```

        for (int i = 0; i < number_of_ships; i++) {
            ships.emplace_back(Ship(ship_sizes[i]));
        }

std::vector<Ship> &ShipManager::getShips() {
    return ships;
}

Ship& ShipManager::getShipByIndex(int index){
    if (index<ships.size() && index>=0){
        return ships[index];
    }
    return ships[ships.size()-1];
}

```

ShipManager.h:

```

#ifndef OOP_LAB1_SHIPMANAGER_H
#define OOP_LAB1_SHIPMANAGER_H

#include <iostream>
#include "Ship.h"

```

```

class ShipManager {
private:
    std::vector<Ship> ships;
public:
    ShipManager(int number_of_ships, const std::vector<int>
&ship_sizes);

    std::vector<Ship> &getShips();
    Ship& getShipByIndex(int index);

};

#endif //OOP_LAB1_SHIPMANAGER_H

```

main.cpp:

```

#include "GameField.h"
#include "ShipManager.h"
int main() {
    GameField field{10, 10};
    auto ship_manager = ShipManager(3, {3, 1, 2});
    int ships_coords[3][2] = {{5, 4},

```

```

        {1, 4},
        {9, 1}};
    auto &ships = ship_manager.getShips();
    auto& s = ship_manager.getShipByIndex(9);

    bool positions[] = {false, false, true};
    for (int i = 0; i < 3; i++) {
        field.placeShip(ships[i], ships_coords[i][0],
ships_coords[i][1], positions[i]);
    }
    field.attackCell(1, 4);
    field.show();
    field.attackCell(1, 4);
    field.show();

    return 0;
}

```