**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**.

1. 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.
2. Создать класс менеджера кораблей, хранящий информацию о кораблях. Данный класс в конструкторе принимает количество кораблей и их размеры, которые нужно расставить на поле.
3. 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:

1. неизвестно (изначально вражеское поле полностью неизвестно),
2. пустая (если на клетке ничего нет)
3. 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 одну клетку на игровом поле. Он хранит информацию о позиции клетки, ее статусе и связанности с кораблем, если таковой имеется.

**Поля класса**:

1. *int x:* Координата по оси X на игровом поле, определяющая горизонтальное положение клетки.
2. *int y:* Координата по оси Y, определяющая вертикальное положение клетки.
3. *bool is\_open:* Логическое значение, указывающее, открыта ли клетка (т.е. была ли она атакована).
4. *Status status:* Поле, представляющее статус клетки, который может принимать значения, такие как Empty (пустая), Occupied (занятая).
5. *Ship\* pointer\_to\_ship:* Указатель на объект класса Ship, который находится в данной клетке. Если клетка пустая, указатель равен nullptr.

**Методы класса**:

1. **Конструктор**:
   * *Cell(int x, int y):* Инициализирует координаты клетки и устанавливает ее статус и открытость по умолчанию.
2. **Метод copyCell**:
   * *void copyCell(const Cell &other):* Копирует значения из другого объекта класса Cell, включая глубокое копирование указателя на корабль. Это позволяет выделить память для нового объекта Ship, если он существует в копируемой клетке. Метод используется в конструкторе копирования и соответствующего ему операторе присваивания.
3. **Конструктор копирования**:
   * *Cell(const Cell &other):* Вызывает метод copyCell для копирования значений из другого объекта класса Cell.
4. **Оператор присваивания**:
   * *Cell& operator=(const Cell &other):* Вызывает метод CopyCell для копирования значения из другого объекта Cell.
5. *void display():* Отображает текущий статус клетки в консоль, используя специальные символы для различных состояний (открыта, пустая, поврежденная и т.д.).
6. *void openCell():* Устанавливает значение is\_open в true, что позволяет открыть клетку для дальнейшего взаимодействия.
7. **Геттеры и сеттеры**: Предоставляют доступ к полям класса и позволяют изменять статус, указатель на корабль и координаты клетки.

* **Класс Ship**

**Описание:**

Класс Ship представляет собой корабль. Он отвечает за хранение информации о длине корабля, его сегментах и их состоянии. Этот класс инкапсулирует логику, связанную с повреждением и размещением корабля на игровом поле, обеспечивая взаимодействие между различными компонентами игры, такими как GameField и ShipManager.

**Поля класса:**

1. *unsigned short length:* Длина корабля, представляющая количество его сегментов.
2. *std::vector<Segment> segments:* Вектор, хранящий сегменты корабля. Каждый сегмент имеет координаты и состояние, которые определяют, был ли он поврежден.
3. *bool is\_placed:* Флаг, указывающий, размещен ли корабль на игровом поле.

**Методы класса:**

1. **Конструкторы и оператор присваивания:** Инициализируют корабль заданной длины или копируют его. Оператор присваивания позволяет присваивать один объект Ship другому.
2. **Методы работы с повреждениями:** *takeDamage(int x, int y):* Обрабатывает атаку на сегмент. Если состояние сегмента больше 0, то отнимает единицу.  
   bool isShipDestroyed(): Проверяет, разрушен ли корабль, возвращает true, если состояние хотя бы одного из сегментов корабля больше 0.
3. **Методы доступа к данным:** Геттеры и сеттеры для длины, координат сегментов и флага размещения. Предусмотрены проверки длины корабля и корректности индексов сегментов.
4. **Вспомогательные методы:** *getShipCoords():* Возвращает координаты сегментов корабля, что позволяет легко получать информацию о его размещении на игровом поле. Используется для проверки столкновения и соприкосновения кораблей в методе класса GameField check\_collide().

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, а также за уведомление их о полученном уроне. Это обеспечивает централизованный контроль над всеми кораблями, что упрощает управление их состоянием и взаимодействие с игровым полем.

**Поля класса**:

1. *std::vector<Ship> ships:* Вектор, хранящий объекты класса Ship, представляющие все корабли, используемые в игре.
2. *std::vector<int> ship\_sizes:* Вектор, хранящий размеры кораблей, передаваемые в конструктор.

**Методы класса**:

1. **Конструктор**:
   * *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. **Геттер**:
   * *std::vector<Ship>& getShips():* Возвращает ссылку на вектор ships, позволяя другим частям программы получить доступ к списку кораблей.
3. **Метод reportDamage**:
   * *void reportDamage(int x, int y):* Обрабатывает урон, полученный по указанным координатам. Метод проходит по всем кораблям в векторе ships и вызывает для каждого из них метод takeDamage, передавая координаты атаки.

Таким образом, класс ShipManager берет на себя ответственность за уведомление всех кораблей о произошедших событиях, и является неким посредником между классом Ship и другими классами

* **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:**

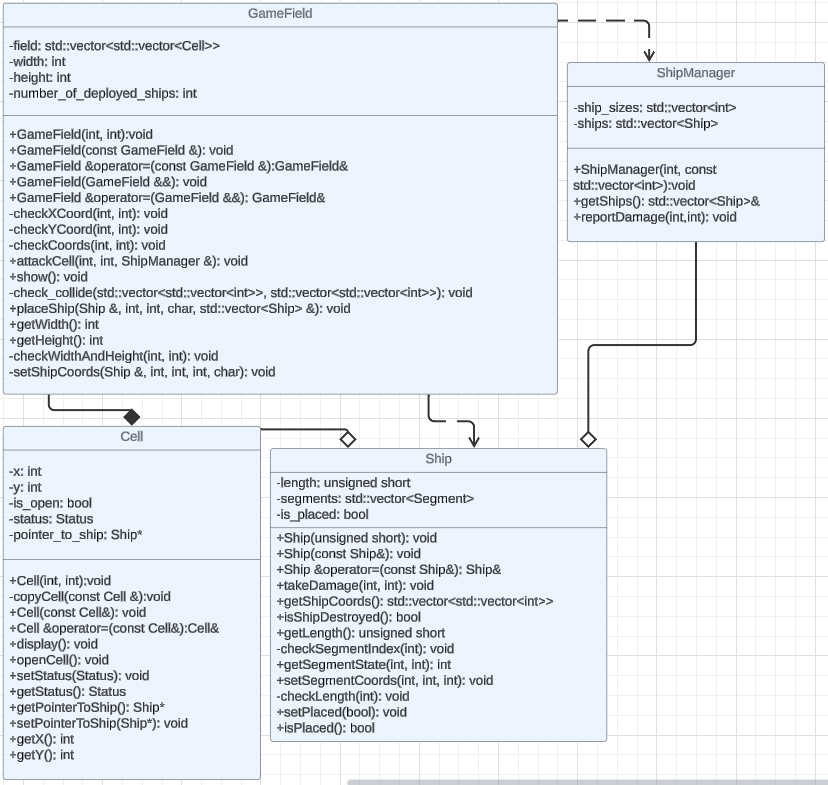
1. *int width:* Ширина игрового поля. (ось Х)
2. *int height:* Высота игрового поля. (ось У)
3. *int number\_of\_deployed\_ships:* Количество размещённых кораблей на поле.
4. *std::vector<std::vector<Cell>> field:* Двумерный вектор клеток, представляющий игровое поле.

#### 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. Глубокое копирование достигается за счёт использования конструктора копирования в классе Cell, что позволяет избежать совместного использования памяти между объектами.
3. **Оператор присваивания копированием:**
   * Обеспечивает глубокое копирование объекта поля, копируя все клетки поля. Глубокое копирование реализуется аналогично конструктору копирования, с использованием конструктора копирования класса Cell.
4. **Конструктор перемещения:**
   * Реализует перемещение объектов, передавая владение ресурсами от одного объекта GameField к другому. Это позволяет избежать лишнего копирования данных.
5. **Оператор присваивания перемещением:**
   * Перемещает ресурсы от одного объекта к другому, аналогично конструктору копирования.
6. **Метод атаки клетки:**
   * *attackCell(int x, int y, ShipManager &ship\_manager):* Обрабатывает атаку на клетку поля по заданным координатам. Если клетка занята кораблем, сообщает об атаке через объект ShipManager.
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, char position, std::vector<Ship> &ships):* Размещает корабль на поле, проверяя на пересечения с другими кораблями. В случае пересечения выбрасывается исключение.
9. **Проверка столкновений кораблей:**
   * *check\_collide(std::vector<std::vector<int>> ship\_1\_coords, std::vector<std::vector<int>> ship\_2\_coords):* 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.

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 |
|  | GameField field{10, 10}; auto ship\_manager = ShipManager(3, **{**3, 1, 2**}**); int ships\_coords[3][2] = {{5, 4},  {9, 4},  {7, 1}}; auto &ships = ship\_manager.getShips(); char positions[] = {'h', 'h', 'v'}; for (int i = 0; i < 3; i++) {  field.placeShip(ships[i], ships\_coords[i][0], ships\_coords[i][1], positions[i], ships); } field.attackCell(5, 4, ship\_manager); field.show();   return 0; | ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️  ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️  ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️  ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️  ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ 🟥 ⬜️ ⬜️ ⬜️ ⬜️  ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️  ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️  ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️  ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️  ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ | Верно |
|  | auto ship\_manager = ShipManager(3, **{**3, 1, 2**}**); auto &ships = ship\_manager.getShips(); std::cout<<ships[0].getLength()<<'\n'; | 3 | Верно |
|  | GameField field{10, 10}; auto ship\_manager = ShipManager(3, **{**3, 1, 2**}**); int ships\_coords[3][2] = {{5, 4},  {9, 4},  {7, 1}}; auto &ships = ship\_manager.getShips(); char positions[] = {'h', 'h', 'v'}; for (int i = 0; i < 3; i++) {  field.placeShip(ships[i], ships\_coords[i][0], ships\_coords[i][1], positions[i], ships); } auto copy\_field = GameField(field); field.attackCell(5, 4, ship\_manager); field.show(); copy\_field.show(); | ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️  ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️  ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️  ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️  ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ 🟥 ⬜️ ⬜️ ⬜️ ⬜️  ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️  ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️  ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️  ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️  ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️  ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️  ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️  ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️  ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️  ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️  ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️  ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️  ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️  ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️  ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ ⬜️ | 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 A.**

**ИСХОДНЫЙ КОД ПРОГРАММЫ.**

Cell.cpp:

*#include "Cell.h"  
Cell::Cell(int x, int y) {  
 this->x = x;  
 this->y = y;  
 is\_open = false;  
 status = Status::Empty;  
 pointer\_to\_ship = nullptr;  
}  
  
void Cell::copyCell(const Cell &other) {  
 x = other.x;  
 y = other.y;  
 is\_open = other.is\_open;  
 status = other.status;  
 if (other.pointer\_to\_ship != nullptr) {  
 pointer\_to\_ship = new Ship(\*other.pointer\_to\_ship);  
 } else {  
 pointer\_to\_ship = nullptr;  
 }  
}  
  
Cell::Cell(const Cell &other) {  
 copyCell(other);  
}  
  
Cell &Cell::operator=(const Cell &other) {  
 if (this != &other) {  
 delete pointer\_to\_ship;  
 copyCell(other);  
 }  
 return \*this;  
}  
  
void Cell::display() {  
 if (is\_open) {  
 if (status == Status::Empty)  
 std::cout << "⚪";  
 else {  
 if (pointer\_to\_ship->isShipDestroyed()) {  
 std::cout << "🔥";  
 return;  
 }  
 int segment\_state = pointer\_to\_ship->getSegmentState(x, y);  
 if (segment\_state == 1) {  
 std::cout << u8"\U0001F7E5";  
 } else {  
 std::cout << "❌";  
 }  
  
 }  
 } else  
 std::cout << "⬜\uFE0F";  
}  
  
void Cell::openCell() {  
 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;  
}  
  
int Cell::getX() {  
 return x;  
}  
  
int Cell::getY() {  
 return y;  
}*

Cell.h:

*#ifndef* ***OOP\_LAB1\_CELL\_H****#define* ***OOP\_LAB1\_CELL\_H****#include "Ship.h"  
  
enum class Status {  
 Empty, Occupied  
};  
  
class Cell {  
private:  
 int x;  
 int y;  
 bool is\_open;  
 Status status;  
 Ship \*pointer\_to\_ship;  
  
 void copyCell(const Cell &other);  
  
public:  
 Cell(int x = 0, int y = 0);  
  
 Cell(const Cell &other);  
  
 Cell &operator=(const Cell &other);  
  
 void display();  
  
 void openCell();  
  
 void setStatus(Status status);  
  
 Status getStatus();  
  
 Ship \*getPointerToShip();  
  
 void setPointerToShip(Ship \*pointer\_to\_ship);  
  
 int getX();  
  
 int getY();  
  
};  
  
  
#endif //OOP\_LAB1\_CELL\_H*

GameField.cpp:

*#include "GameField.h"  
  
GameField::GameField(int width, int height) {  
 checkWidthAndHeight(width, height);  
 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(j, i);  
 }  
 }  
}  
  
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, ShipManager &ship\_manager) {  
 checkCoords(x, y);  
 field[y][x].openCell();  
 if (field[y][x].getStatus() == Status::Occupied) {  
 ship\_manager.reportDamage(x, y);  
 }  
}  
  
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';  
}  
  
void  
GameField::check\_collide(std::vector<std::vector<int>> ship\_1\_coords, std::vector<std::vector<int>> ship\_2\_coords) {  
 for (auto &ship\_1\_coord: ship\_1\_coords) {  
 int x = ship\_1\_coord[0];  
 int y = ship\_1\_coord[1];  
 int block\_coords[9][2] = {{x - 1, y - 1},  
 {x - 1, y},  
 {x - 1, y + 1},  
 {x, y - 1},  
 {x, y},  
 {x, y + 1},  
 {x + 1, y - 1},  
 {x + 1, y},  
 {x + 1, y + 1}};  
 for (auto &block\_coord: block\_coords) {  
 for (auto &ship\_2\_coord: ship\_2\_coords) {  
 if (block\_coord[0] == ship\_2\_coord[0] && block\_coord[1] == ship\_2\_coord[1]) {  
 throw std::invalid\_argument("Ships collide");  
 }  
 }  
 }  
 }  
}  
  
void GameField::setShipCoords(Ship &ship, int x, int y, int ship\_length, char position) {  
 for (int i = 0; i < ship\_length; i++) {  
 if (position == 'v') {  
 checkYCoord(y, ship\_length);  
 field[y + i][x].setStatus(Status::Occupied);  
 field[y + i][x].setPointerToShip(&ship);  
 ship.setSegmentCoords(x, y + i, i);  
 if (!(ship.isPlaced())) {  
 ship.setPlaced(true);  
 }  
 } else if (position == 'h') {  
 checkXCoord(x, ship\_length);  
 field[y][x + i].setStatus(Status::Occupied);  
 field[y][x + i].setPointerToShip(&ship);  
 ship.setSegmentCoords(x + i, y, i);  
 if (!(ship.isPlaced())) {  
 ship.setPlaced(true);  
 }  
 } else {  
 throw std::invalid\_argument("Invalid position");  
 }  
 }  
}  
  
void GameField::placeShip(Ship &ship, int x, int y, char position, std::vector<Ship> &ships) {  
 auto ship\_length = ship.getLength();  
 if (number\_of\_deployed\_ships != 0) {  
 for (auto &other\_ship: ships) {  
 if (&other\_ship == &ship || !(other\_ship.isPlaced()))  
 continue;  
 auto other\_ship\_coords = other\_ship.getShipCoords();  
 setShipCoords(ship, x, y, ship\_length, position);  
 auto ship\_coords = ship.getShipCoords();  
 check\_collide(ship\_coords, other\_ship\_coords);  
 }  
 }  
 setShipCoords(ship, x, y, ship\_length, position);  
 number\_of\_deployed\_ships++;  
}  
  
int GameField::getWidth() {  
 return width;  
}  
  
int GameField::getHeight() {  
 return height;  
}  
  
  
void GameField::checkWidthAndHeight(int width, int height) {  
 if (width < 0 || height < 0)  
 throw std::invalid\_argument("Width and height must be positive");  
}  
  
void GameField::checkXCoord(int x, int ship\_length) {  
 if (x < 0 || x >= width)  
 throw std::out\_of\_range("X coordinate is out of bounds");  
 if (x + ship\_length > width)  
 throw std::out\_of\_range("Ship goes out of bounds");  
}  
  
void GameField::checkYCoord(int y, int ship\_length) {  
 if (y < 0 || y >= height)  
 throw std::out\_of\_range("Y coordinate is out of bounds");  
 if (y + ship\_length > height)  
 throw std::out\_of\_range("Ship goes out of bounds");  
  
}  
  
void GameField::checkCoords(int x, int y) {  
 if (x < 0 || x >= width || y < 0 || y >= height)  
 throw std::out\_of\_range("Coordinates are out of bounds");  
}*

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;  
  
 void check\_collide(std::vector<std::vector<int>> ship\_1\_coords, std::vector<std::vector<int>> ship\_2\_coords);  
  
 void checkXCoord(int x, int ship\_length);  
  
 void checkYCoord(int y, int ship\_length);  
  
 void checkCoords(int x, int y);  
  
 void checkWidthAndHeight(int width, int height);  
  
 void setShipCoords(Ship &ship, int x, int y, int ship\_length, char position);  
  
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, ShipManager &ship\_manager);  
  
 void show();  
  
 void placeShip(Ship &ship, int x, int y, char position, std::vector<Ship> &ships);  
  
 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;  
 }  
}  
  
Ship::Ship(const Ship &other) {  
 length = other.length;  
 segments.resize(length);  
 is\_placed = other.is\_placed;  
 for (int i = 0;i<length;i++){  
 segments[i].state = other.segments[i].state;  
 segments[i].x = other.segments[i].x;  
 segments[i].y = other.segments[i].y;  
 }  
}  
  
  
Ship &Ship::operator=(const Ship &other) {  
 if (this == &other) {  
 return \*this;  
 }  
 length = other.length;  
 segments.resize(length);  
 is\_placed = other.is\_placed;  
 for (int i = 0;i<length;i++){  
 segments[i].state = other.segments[i].state;  
 segments[i].x = other.segments[i].x;  
 segments[i].y = other.segments[i].y;  
 }  
 return \*this;  
}  
  
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>  
  
struct Segment {  
 int state;  
 int x;  
 int y;  
};  
  
class Ship {  
private:  
 unsigned short length;  
 std::vector<Segment> segments;  
 bool is\_placed;  
  
 void checkSegmentIndex(int index);  
  
 void checkLength(int length);  
  
public:  
 Ship(unsigned short length);  
  
 Ship(const Ship &other);  
  
 Ship &operator=(const Ship &other);  
  
 void takeDamage(int x, int y);  
  
 std::vector<std::vector<int>> getShipCoords();  
  
 bool isShipDestroyed();  
  
 unsigned short getLength();  
  
 int getSegmentState(int x, int y);  
  
 void setSegmentCoords(int x, int y, int index);  
  
 void setPlaced(bool is\_placed);  
  
 bool isPlaced();  
  
};  
  
  
#endif //OOP\_LAB1\_SHIP\_H*

ShipManager.cpp:

#include "ShipManager.h"  
  
ShipManager::ShipManager(*int* number\_of\_ships, *const* std::vector<*int*> &ship\_sizes) {  
 *this*->ship\_sizes = 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;  
}  
  
*void* ShipManager::reportDamage(*int* x, *int* y) {  
 *for* (*auto* &ship: ships) {  
 ship.takeDamage(x, y);  
 }  
}

ShipManager.h:

*#ifndef* ***OOP\_LAB1\_SHIPMANAGER\_H****#define* ***OOP\_LAB1\_SHIPMANAGER\_H****#include <iostream>  
#include "Ship.h"  
  
class ShipManager {  
private:  
 std::vector<int> ship\_sizes;  
 std::vector<Ship> ships;  
public:  
 ShipManager(int number\_of\_ships, const std::vector<int> &ship\_sizes);  
  
 std::vector<Ship> &getShips();  
  
 void reportDamage(int x, int y);  
  
};  
  
  
#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},  
 {9, 4},  
 {7, 1}};  
 auto &ships = ship\_manager.getShips();  
 char positions[] = {'h', 'h', 'v'};  
 for (int i = 0; i < 3; i++) {  
 field.placeShip(ships[i], ships\_coords[i][0], ships\_coords[i][1], positions[i], ships);  
 }  
 field.attackCell(5, 4, ship\_manager);  
 field.show();  
 field.attackCell(5, 4, ship\_manager);  
 field.show();  
  
 return 0;  
}*