**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. 3384 |  | Поздеев В.Д |
| Преподаватель |  | Шестопалов Р.П. |

Санкт-Петербург

2024

## Цель работы.

Научиться основным принципам ООП. Разобраться в определениях ООП и получить практические навыки путем написания программы на языке С++. Создать несколько классов для игры морской бой.

## Задание.

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.

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

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:

неизвестно (изначально вражеское поле полностью неизвестно),

пустая (если на клетке ничего нет)

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

## Теоретические положения

1. Объектно-ориентированное программирование (ООП)

В работе применяются основные принципы ООП: инкапсуляция, наследование и полиморфизм. Классы служат шаблонами для объектов, содержащих данные и методы.

- Инкапсуляция: Скрытие данных класса от внешнего доступа. Например, состояние сегментов корабля скрыто с помощью приватных полей и публичных методов.

- Наследование: Создание новых классов на основе существующих, что позволяет расширять функционал.

- Полиморфизм: Обработка объектов разных типов через общий интерфейс.

2. Использование *enum*

Перечисления (*enum*) задают ограниченные значения для переменных, полезные для статусов клеток игрового поля и состояний сегментов корабля. Это улучшает читаемость кода и снижает вероятность ошибок.

3. Управление памятью

Эффективное управление памятью критично. В работе реализовано глубокое копирование объектов для предотвращения нежелательных изменений и механизм перемещения, который оптимизирует производительность.

- Глубокое копирование: Создание независимой копии объекта.

- Перемещение: Передача владения ресурсами без копирования.

4. Проверка условий и управления состояниями

Необходимо проверять условия перед выполнением операций:

- Размеры и ориентация корабля перед размещением.

- Состояния клеток перед атакой для обновления статусов.

- Обеспечение того, чтобы корабли не соприкасались или не пересекались на поле.

*initializer\_list —* класс из директивы *initializer\_list*, принимает массив заданного типа в формате *{a\_0, .. , a\_n}*, не принимая изначально размеры, который в последствие можно подавать в функции. Данный класс полезен когда пользователь вводит массив неизвестной длинны.  
 *std* — стандартное пространство имён в C++.

*move* — функция в C++, которая используется для явного указания, что объект может быть перемещён, а не скопирован.

## Выполнение работы.

Класс Ship состоит из полей *segments — vector* который состоит из *segmentState*, *coordinates — pair* который принимает два *size\_t*, *is\_vertical —* является *bool, length —* является *size\_t*. Также в нем реализован *enum segmentState* состоящий из *normal*, *damaged*, *destroyed*. У класса есть несколько конструкторов. Конструктор *Ship()*, который ничего не принимает. Он записывает в поле *coordinates* значения *{0, 0}*, в поле *length — 0* , в поле *is\_vertical — true*. Конструктор который принимает *length, coordinates, is\_vertical*, соответствующих типам записанных выше. В поля класса записываются переменные соответствующие названиям. Конструктор копирования, который принимает переменную *ship* типа *const Ship&.* В *length* записывается *ship.length*, в остальные поля соответственно записываются поля *ship* названиям. Конструктор перемещения, который принимает *Ship && ship* — ссылку на *rvalue*. В каждое поле соответственно записывается *std::move(ship.<название\_поля>)*. Созданы два оператора присваивания. Один для копирования, другой для перемещения. Оператор для копирования принимает *const Ship& ship*. Внутри происходит проверка: *this != \*ship*, которая проверяет не происходит ли копирование самого объекта перенос данных в которых происходит. Если все хорошо, то каждому полю класса присваивается значение поле подаваемого значения *ship* в конце возвращается с помощью *\*this* с помощью *return,* так как оператор присваивания должен возвращать ссылку на объект. Оператор присваивания для перемещения работает также как и предыдущий, но принимает он ссылку на *rvalue* — *Ship && ship*. А также внутри присваивание идет не к самим переменным а с помощью *std::move(ship.<название\_поля>)*.

В классе созданы методы-геттеры, которые возвращают значения полей перечисленных выше. Методы копируют значения полей и возвращают их. Перед каждым из таких методов стоит *const*, который говорит, что этот метод не изменяет поле класса. Соответственно реализованы методы: *getLen(), getCoor(), IsVertical()* и метод *getSegment*, который принимает *size\_t index* и возвращает копию сегмента корабля под индексом *index* из вектора *segments*. Создан метод *void Attack()*, который принимает *size\_t index*. Внутри метода вызывается конструкция *switch-case*, которая принимает *segments[index].* В случае если оно равно *normal*, то *segments[index]* меняется на *damaged*, далее вызывается *break*, чтобы не попасть в другие случаи. Когда *damaged*, то заменяется на *destroyed.* Создан метод *bool isDestroyed*(). Создается флаг *bool is\_destroyed = true*. Создается цикл в котором мы копируем каждый сегмент корабля в переменную*(segmentState segment: segments)* и проверяем является ли он *destroyed(segment != destroyed)*. В случае выполнения условия флаг *is\_destroyed* меняется на *false*. Вызывается *break.* Возвращается *is\_destroyed*.

Класс *shipManager* имеет поле *vector<Ships> ships.* В нем реализованы несколько конструкторов. Пустой конструктор *shipManager()=default*, который ничего не принимает, прописан в *header* файле класса. Конструктор, который принимает *initializer\_list<size\_t> lengths*, *initializer\_list<pair<size\_t, size\_t>> coordinates\_arr, initializer\_list<bool> is\_vertical.* В начале идет проверка чтобы размеры трех массивов совпадали. В случае не выполнения условия вызывается ошибка *invalid\_argument("ARRAYS MUST HAVE SAME SIZE!").* Далее создается переменная *auto length*, которая принимает итератор *lengths.begin()*, далее также создаются переменные *coordinates* и *is\_vertical*. Создается цикл *while,* который работает пока *length* не станет *lengths.enc()*. Внутри вызывается метод *addShip,*который будет описан ниже. К каждой из переменных вызывается префиксный оператор увеличения. Создан конструктор копирования, который копирует с помощью *ships(ship\_manager.ships)*. Также работает конструктор перемещения, но вместо этого происходит *ships(std::move(ship\_manager.ships))*.

Созданы операторы присваивания копирования и перемещения. Внутри происходит проверка на само присваивание. И в первом случае *ships* приравнивается к *ship\_manager.ships*, во втором случае к *std::move(ship\_manager.ships)*.

Создан метод-геттер *Ship getShip,* который принимает индекс и соответственно возвращает копию объекта *ships[index]*. Перед ним также стоит *const*.

Создан метод проверки на близость корабля по координате и расположению к другим кораблям *bool closeShips,* который принимает *size\_t length, pair<size\_t, size\_t> coordinates, bool is\_vertical*. Внутри создается переменная *size\_t len\_subtr\_x = is\_vertical ? length-1 : 0* и переменная *size\_t len\_subtr\_y = is\_vertical ? Length-1 : 0.* Создается флаг *bool close\_to\_ship = false.* Данные переменные нужны для проверки координаты дальнего сегмента. Если корабль вертикальный, то прибавляется по *x.* Если горизонтальный, то по *y*. Далее создается цикл в котором проверяется каждый корабль. В нем создается условие на проверку является ли текущий корабль вертикальным. При выполнении условия идет проверка координаты x и координаты y на попадание в прямоугольник вокруг корабля. Делается это с помощью *(ship.getCoor().second-1 <= coordinates.second &&*

*coordinates.second + len\_subtr\_y <= ship.getCoor().second+ship.getLen()) &&*

*(ship.getCoor().first-1 <= coordinates.first &&*

*coordinates.first+len\_subtr\_x <= ship.getCoor().first+1)*

*).* Получаем координату *y* текущего корабля вычитаем -1 и она должна быть меньше или равна чем координата принимаемого значения *y*, затем прибавляем к координате *len\_subtr\_y*, и оно должно быть меньше или равно чем координата y корабля + его длина. Делаем тоже самое с первой координатой, но теперь первая координата + *len\_subtr\_y* должна быть меньше или равна первой координаты корабля + 1. Если условие выполняется, то меняется флаг *close\_to\_ship* на *true.* Выводится ошибка с помощью *cerr,* что корабль соприкоснулся с другим кораблем по координате.(*std::cerr << "SHIP x:" << coordinates.first << " y:" << coordinates.second << " length:" << length*

*<< " is\_vertical: " << is\_vertical;*

*std::cerr << " TOUCHES OTHER SHIP x:" << ship.getCoor().first << " y:" << ship.getCoor().second << " length:" << ship.getLen()*

*<< " is\_vertical:" << ship.IsVertical() << "\n";*) и вызывается *break.* Точно также происходит если корабль расположен горизонтально, только теперь мы должны прибавлять к первой координате длину текущего корабля. Вызывается такая же ошибка. После этого возвращаем значение *close\_to\_ship*.

Создан метод *void addShip*, который принимает параметры длинны, координат и флаг вертикального положения. Внутри проверяется на близость к кораблям с помощью *!closeShips*. Если условие выполняется, то с помощью метода *push\_back(Ship(length, coordinates, is\_vertical))* в вектор *ships* добавляется корабль также происходит возвращение *return*. Иначе вызывается ошибка с помощью *throw std::invalid\_argument("SHIP TOUCHES OTHER SHIP OR SHIPS!").*

Создан метод *Attack*, который принимает *pair<size\_t, size\_t> coordinates*. Внутри создается цикл по ссылкам на корабли, в котором проверяется в какой корабль попал игрок. Создается условие на положение корабля(*ship.IsVertical()*). Если оно выполняется, то создается условие в котором уже происходит проверка по координатам корабля. Если первая координата атаки равна первой координате корабля и вторая координата атаки лежит между второй координатой корабля и второй координатой корабля + длина корабля - 1 включительно, то создается переменная *size\_t index* равная *coordinates.second-ship.getCoor().second.* Вызывается *ship.Attack(index)*. Возвращается *true*. В противном случае происходит точно такая же проверка, но со второй координаты и соответственно, с первой + длинна корабля - 1. Если цикл закончился и программа не попала ни в одно из условий, то возвращается *false*.

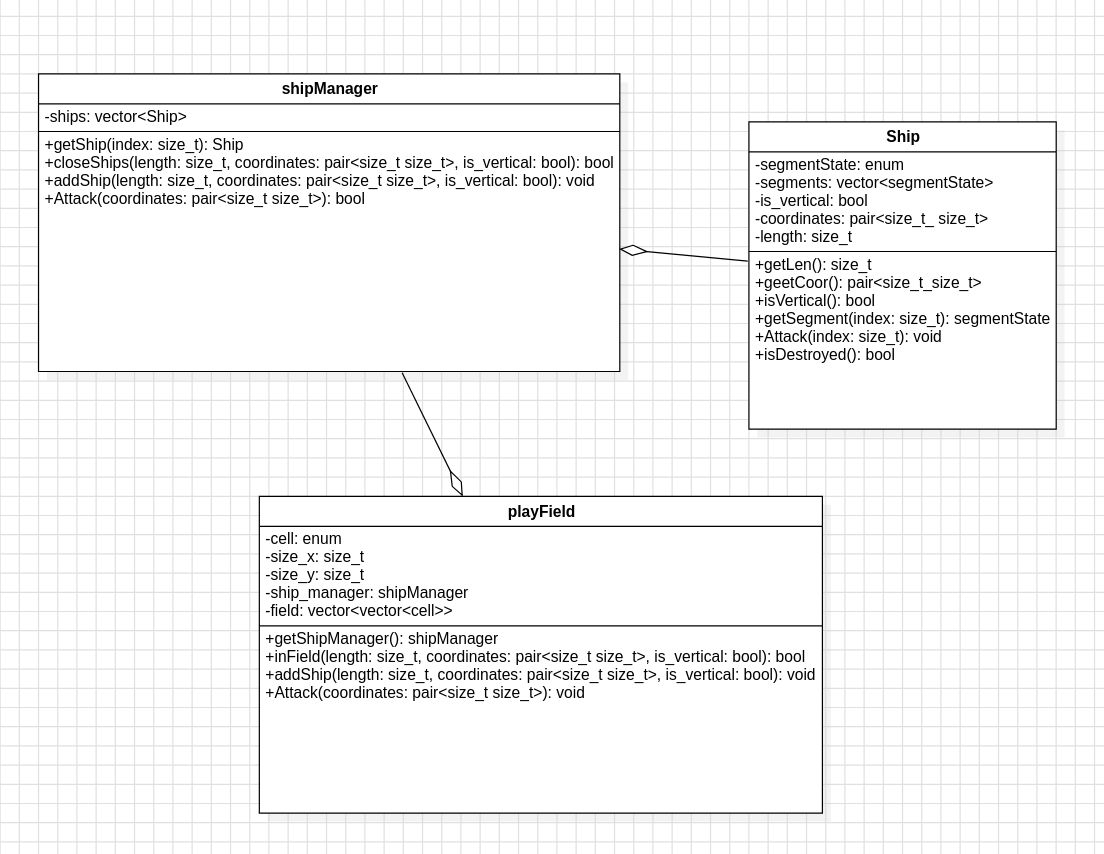
Создан класс игрового поля *playField*. В нем есть *enum cell,* который состоит из *unknown, empty, ship*. Далее созданы поля *size\_x, size\_y, ship\_manager* и двойной вектор *field*. У него есть конструктор который ничего не принимает. Он записывает нули в поля *size\_x, size\_y,* а также вызывает пустой конструктор *ship\_manager()*. Конструктор, который принимает только размеры поля. Он записывает размеры поля в соответствующие поля и вызывает пустой конструктор *ship\_manager()*. Далее у вектора *field* вызывается метод *resize,* который принимает *size\_y* и вторым параметром вектор размера *size\_x*, заполненный *unknown*. Данная функция создаст матрицу. Конструктор, который принимает *size\_t size\_x, size\_t size\_y, std::initializer\_list<size\_t> lengths, std::initializer\_list<std::pair<size\_t, size\_t>> coordinates\_arr, std::initializer\_list<bool> is\_vertical\_arr.* Он запихивает размеры в поля размеров, списки в конструктор *ship\_manager,* а внутри создает двумерный массив по размерам. Создан конструктор копирования, который записывает данные в соответствующие поля, конструктор перемещения, который записывает данные в нужные поля с помощью *std::move*. Точно тоже самое происходит и в операторах присваивания.

Создан метод-геттер, который возвращает копию *shipManager*. Перед ним *const*.   
 Создан метод проверки на нахождения корабля в поле *bool inField*. Внутри создается переменная *size\_t len\_subtr\_x = is\_vertical ? length-1 : 0* и переменная *size\_t len\_subtr\_y = is\_vertical ? Length-1 : 0.* Далее создается условие в котором проверяются координаты дальнего и первого сегмента. Если первая координата первого сегмента больше равна 0 и первая координата первого сегмента + *len\_subtr\_x* меньше или равна *size\_x-1.* Точно такая же проверка второй координаты. Есливсе хорошо, то возвращается *true*, иначе *false.* Метод *addShip* принимает поля для корабля, далее идет проверка *inField().* Если все хорошо, то возвращается значение *ship\_manager.addShip()*. Иначе вызывается ошибка *throw std::invalid\_argument("OBJECT IS OUT OF BORDER!").*

Метод *Attack* принимает координаты для атаки. Внутри происходит на валидность координаты с помощью *inField*, далее вызывается метод *Attack* у *ship\_manager* и если произошло попадание то состояние клетки меняется с *unknown* на *ship,* иначе на *empty*. Вызывается *return*. Если все плохо то вызывается ошибка *throw std::invalid\_argument("COORDINATES ARE OUT OF BORDER!").*

Метод *printField* печатает игровое поле.

## Диаграммы классов.

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 |
|  | playField skibidi(5, 5);  skibidi.addShip(1, {4, 4}, true);  skibidi.Attack({4, 4});  std::cout<<skibidi.getShipManager().getShip(0).isDestroyed() <<std::endl;  skibidi.Attack({4,4});  std::cout<<skibidi.getShipManager().getShip(0).isDestroyed() << std::endl; | 0  1 | Correct |
|  | playField skibidi(1, 1);  skibidi.printField();  skibidi.Attack({0,0 });  skibidi.printField(); | 0  1 | Correct |
|  | playField skibidi(1, 1);  skibidi.addShip(1, {0, 0}, true);  skibidi.printField();  skibidi.Attack({0,0 });  skibidi.printField(); | 0  2 | Correct |

## Выводы.

Были изучены основные принципы ООП. Разработан программный код с поставленными в задании классами и соответствующими методами для них.

# Приложение А Исходный код программы

Сначала указываем имя файла, в котором код лежит в репозитории:

Название файла: Ship.h

#include <vector>

#include <iostream>

#include <utility>

#include <initializer\_list>

#include <list>

#include <cmath>

class Ship{

enum segmentState{

normal,

damaged,

destroyed

};

std::vector<segmentState> segments;

bool is\_vertical = true;

std::pair<size\_t, size\_t> coordinates;

size\_t length;

public:

Ship() = default;

Ship(size\_t length, std::pair<size\_t, size\_t> coordinates, bool is\_vertical);

Ship(const Ship &ship);

Ship& operator = (const Ship& ship);

Ship(Ship && ship) noexcept;

Ship& operator = (Ship && ship) noexcept;

size\_t getLen() const;

std::pair<size\_t, size\_t> getCoor() const;

bool IsVertical() const;

segmentState getSegment(size\_t index) const;

void Attack(size\_t index);

bool isDestroyed();

};

Название файла: shipManager.h

#include "Ship.h"

class shipManager{

std::vector<Ship> ships;

public:

shipManager() = default;

shipManager(std::initializer\_list<size\_t> lengths,

std::initializer\_list<std::pair<size\_t, size\_t>> coordinates\_arr,

std::initializer\_list<bool> is\_vertical\_arr);

shipManager(const shipManager &ship\_manager);

shipManager& operator = (const shipManager& ship\_manager);

shipManager(shipManager && ship\_manager) noexcept;

shipManager& operator = (shipManager && ship\_manager) noexcept;

Ship getShip(size\_t index) const;

bool closeShips(size\_t length, std::pair<size\_t, size\_t>coordinates, bool is\_vertical);

void addShip(size\_t length, std::pair<size\_t, size\_t>coordinates, bool is\_vertical);

bool Attack(std::pair<size\_t, size\_t> coordinates);

};

Название файла: playField.h

#include "shipManager.h"

class playField{

enum cell{

unknown,

empty,

ship

};

size\_t size\_x;

size\_t size\_y;

shipManager ship\_manager;

std::vector<std::vector<cell>> field;

public:

playField();

playField(size\_t size\_x, size\_t size\_y);

playField(size\_t size\_x, size\_t size\_y,

std::initializer\_list<size\_t> lengths,

std::initializer\_list<std::pair<size\_t, size\_t>> coordinates\_arr,

std::initializer\_list<bool> is\_vertical\_arr);

playField(const playField &play\_field);

playField& operator = (const playField& play\_field);

playField(playField && play\_field) noexcept;

playField& operator = (playField && play\_field) noexcept;

shipManager getShipManager() const;

bool inField(size\_t length, std::pair<size\_t, size\_t>coordinates, bool is\_vertical);

void addShip(size\_t length, std::pair<size\_t, size\_t>coordinates, bool is\_vertical);

void Attack(std::pair<size\_t, size\_t> coordinates);

void printField();

};

Название файла: Ship.cpp

#include "Ship.h"

Ship::Ship(size\_t length, std::pair<size\_t, size\_t> coordinates, bool is\_vertical)

: length(length), coordinates(coordinates), is\_vertical(is\_vertical), segments(length, normal){

if(length < 1 || length > 4 ){

throw std::invalid\_argument("SHIP LENGTH IS BETWEEN 1-4!");

}

}

Ship::Ship(const Ship &ship) : length(ship.length), coordinates(ship.coordinates), segments(ship.segments){} // copy construct

Ship& Ship::operator = (const Ship& ship){

if(this != &ship){

length = ship.length;

coordinates = ship.coordinates;

segments = ship.segments;

}

return \*this;

}

Ship::Ship(Ship && ship) noexcept : length(std::move(ship.length)), is\_vertical(std::move(ship.is\_vertical)) {

coordinates = std::move(ship.coordinates);

segments = std::move(ship.segments);

}

Ship& Ship::operator = (Ship && ship) noexcept{

if(this != &ship){

length = std::move(ship.length);

is\_vertical = std::move(ship.is\_vertical);

coordinates = std::move(ship.coordinates);

segments = std::move(ship.segments);

}

return \*this;

}

size\_t Ship::getLen() const{

return length;

}

std::pair<size\_t, size\_t> Ship::getCoor() const{

return coordinates;

}

bool Ship::IsVertical() const{

return is\_vertical;

}

Ship::segmentState Ship::getSegment(size\_t index) const{

return segments[index];

}

void Ship::Attack(size\_t index){

switch(segments[index]){

case normal:

segments[index] = damaged;

break;

case damaged:

segments[index] = destroyed;

break;

}

}

bool Ship::isDestroyed(){

bool is\_destroyed = true;

for(segmentState segment: segments){

if(segment != destroyed){

is\_destroyed = false;

break;

}

}

return is\_destroyed;

}

Название файла: shipManager.cpp

#include "shipManager.h"

shipManager::shipManager(std::initializer\_list<size\_t> lengths,

std::initializer\_list<std::pair<size\_t, size\_t>> coordinates\_arr,

std::initializer\_list<bool> is\_vertical\_arr){

if(lengths.size() != coordinates\_arr.size() ||

is\_vertical\_arr.size() != coordinates\_arr.size() ||

lengths.size() != is\_vertical\_arr.size()){

throw std::invalid\_argument("ARRAYS MUST HAVE SAME SIZE!");

}

auto length = lengths.begin();

auto coordinates = coordinates\_arr.begin();

auto is\_vertical = is\_vertical\_arr.begin();

while (length != lengths.end()) {

this->addShip(\*length, \*coordinates, \*is\_vertical);

++length;

++coordinates;

++is\_vertical;

}

};

shipManager::shipManager(const shipManager &ship\_manager):ships(ship\_manager.ships){} // конструктор копирования

shipManager& shipManager::operator = (const shipManager& ship\_manager){

if(this != &ship\_manager){

ships = ship\_manager.ships;

}

return \*this;

}

shipManager::shipManager(shipManager && ship\_manager)noexcept:ships(std::move(ship\_manager.ships)){}

shipManager& shipManager::operator = (shipManager && ship\_manager)noexcept{

if(this != &ship\_manager){

ships = std::move(ship\_manager.ships);

}

return \*this;

}

Ship shipManager::getShip(size\_t index) const{

return ships[index];

}

bool shipManager::closeShips(size\_t length, std::pair<size\_t, size\_t>coordinates, bool is\_vertical){

size\_t len\_subtr\_x = is\_vertical ? length-1 : 0;

size\_t len\_subtr\_y = is\_vertical ? 0 : length-1;

bool close\_to\_ship = false;

for(Ship ship: ships){

if(ship.IsVertical()){

if((ship.getCoor().second-1 <= coordinates.second &&

coordinates.second + len\_subtr\_y <= ship.getCoor().second+ship.getLen()) &&

(ship.getCoor().first-1 <= coordinates.first &&

coordinates.first+len\_subtr\_x <= ship.getCoor().first+1)

) {

std::cerr << "SHIP x:" << coordinates.first << " y:" << coordinates.second << " length:" << length

<< " is\_vertical: " << is\_vertical;

std::cerr << " TOUCHES OTHER SHIP x:" << ship.getCoor().first << " y:" << ship.getCoor().second << " length:" << ship.getLen()

<< " is\_vertical:" << ship.IsVertical() << "\n";

close\_to\_ship = true;

break;

}

}

else{

if((ship.getCoor().second-1 <= coordinates.second &&

coordinates.second + len\_subtr\_y <= ship.getCoor().second+1) &&

(ship.getCoor().first-1 <= coordinates.first &&

coordinates.first+len\_subtr\_x <= ship.getCoor().first+ship.getLen())

) {

std::cerr << "SHIP x:" << coordinates.first << " y: " << coordinates.second << " length: " << length

<< " positioned: " << is\_vertical ? "vertical\n" : "horizontal\n";

std::cerr << "TOUCHES OTHER SHIP x: " << ship.getCoor().first << " y: " << ship.getCoor().second << " length: " << ship.getLen()

<< " positioned: " << ship.IsVertical() ? "vertical\n" : "horizontal\n";

close\_to\_ship = true;

break;

}

}

}

return close\_to\_ship;

}

void shipManager::addShip(size\_t length, std::pair<size\_t, size\_t>coordinates, bool is\_vertical){

if(!closeShips(length, coordinates, is\_vertical)){

return ships.push\_back(Ship(length, coordinates, is\_vertical));

}

throw std::invalid\_argument("SHIP TOUCHES OTHER SHIP OR SHIPS!");

}

bool shipManager::Attack(std::pair <size\_t, size\_t> coordinates){

for(Ship &ship: ships){

if(ship.IsVertical()){

if(coordinates.first == ship.getCoor().first &&

ship.getCoor().second <= coordinates.second && coordinates.second <= ship.getCoor().second+ship.getLen()-1){

size\_t index = coordinates.second - ship.getCoor().second;

ship.Attack(index);

return true;

}

}

else{

if(coordinates.second == ship.getCoor().second &&

ship.getCoor().first <= coordinates.first && coordinates.first <= ship.getCoor().first+ship.getLen()-1){

size\_t index = coordinates.first - ship.getCoor().first;

ship.Attack(index);

return true;

}

}

}

return false;

}

Название файла: playField.cpp

#include "playField.h"

playField::playField() : size\_x(0), size\_y(0), ship\_manager() {

}

playField::playField(size\_t size\_x, size\_t size\_y)

: size\_x(size\_x), size\_y(size\_y), ship\_manager() {

field.resize(size\_y, std::vector<cell>(size\_x, unknown));

}

playField::playField(size\_t size\_x, size\_t size\_y,

std::initializer\_list<size\_t> lengths,

std::initializer\_list<std::pair<size\_t, size\_t>> coordinates\_arr,

std::initializer\_list<bool> is\_vertical\_arr)

: size\_x(size\_x), size\_y(size\_y), ship\_manager(lengths, coordinates\_arr, is\_vertical\_arr){

field.resize(size\_y, std::vector<cell>(size\_x, unknown));

}

playField::playField(const playField &play\_field):size\_x(play\_field.size\_x),

size\_y(play\_field.size\_y), ship\_manager(play\_field.ship\_manager), field(play\_field.field){

}

playField& playField::operator = (const playField& play\_field){

if(this != &play\_field){

size\_x = play\_field.size\_x;

size\_y = play\_field.size\_y;

ship\_manager = play\_field.ship\_manager;

field = play\_field.field;

}

return \*this;

}

playField::playField(playField && play\_field) noexcept :size\_x(std::move(play\_field.size\_x)),

size\_y(std::move(play\_field.size\_y)),

ship\_manager(std::move(play\_field.ship\_manager)){

field = std::move(play\_field.field);

play\_field.field.clear();

}

playField& playField::operator = (playField && play\_field) noexcept {

if(this != &play\_field){

size\_x = std::move(play\_field.size\_x);

size\_y = std::move(play\_field.size\_y);

ship\_manager = std::move(play\_field.ship\_manager);

field = std::move(play\_field.field);

}

return \*this;

}

shipManager playField::getShipManager() const{

return ship\_manager;

}

bool playField::inField(size\_t length, std::pair<size\_t, size\_t>coordinates, bool is\_vertical){

size\_t len\_subtr\_y = is\_vertical ? length-1 : 0;

size\_t len\_subtr\_x = is\_vertical ? 0 : length-1;

if(0 <= coordinates.first && coordinates.first + len\_subtr\_x <= size\_x-1 &&

0 <= coordinates.second && coordinates.second + len\_subtr\_y <= size\_y-1

){

return true;

}

return false;

}

void playField::addShip(size\_t length, std::pair<size\_t, size\_t>coordinates, bool is\_vertical){

if(inField(length, coordinates, is\_vertical)){

return ship\_manager.addShip(length, coordinates, is\_vertical);

}

throw std::invalid\_argument("OBJECT IS OUT OF BORDER!");

}

void playField::Attack(std::pair<size\_t, size\_t> coordinates){

if(inField(1, coordinates, true)){

if(ship\_manager.Attack(coordinates)){

field[coordinates.second][coordinates.first] = ship;

}

else{

field[coordinates.second][coordinates.first] = empty;

}

return;

}

throw std::invalid\_argument("COORDINATES ARE OUT OF BORDER!");

}

void playField::printField(){

for(int y = size\_y-1; y != -1; y--){

for(int x = 0; x != size\_x; x++){

std::cout << field[y][x] << " ";

}

std::cout << "\n";

}

}