**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

|  |  |  |
| --- | --- | --- |
| Студентка гр. 3385 |  | Мучник М.К. |
| Преподаватель |  | Первицкий А.Ю. |

Санкт-Петербург

2024

## Цель работы

Изучить методику создания классов на языке C++. Реализовать классы корабля, менеджера кораблей и игрового поля с соответствующим функционалом 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**

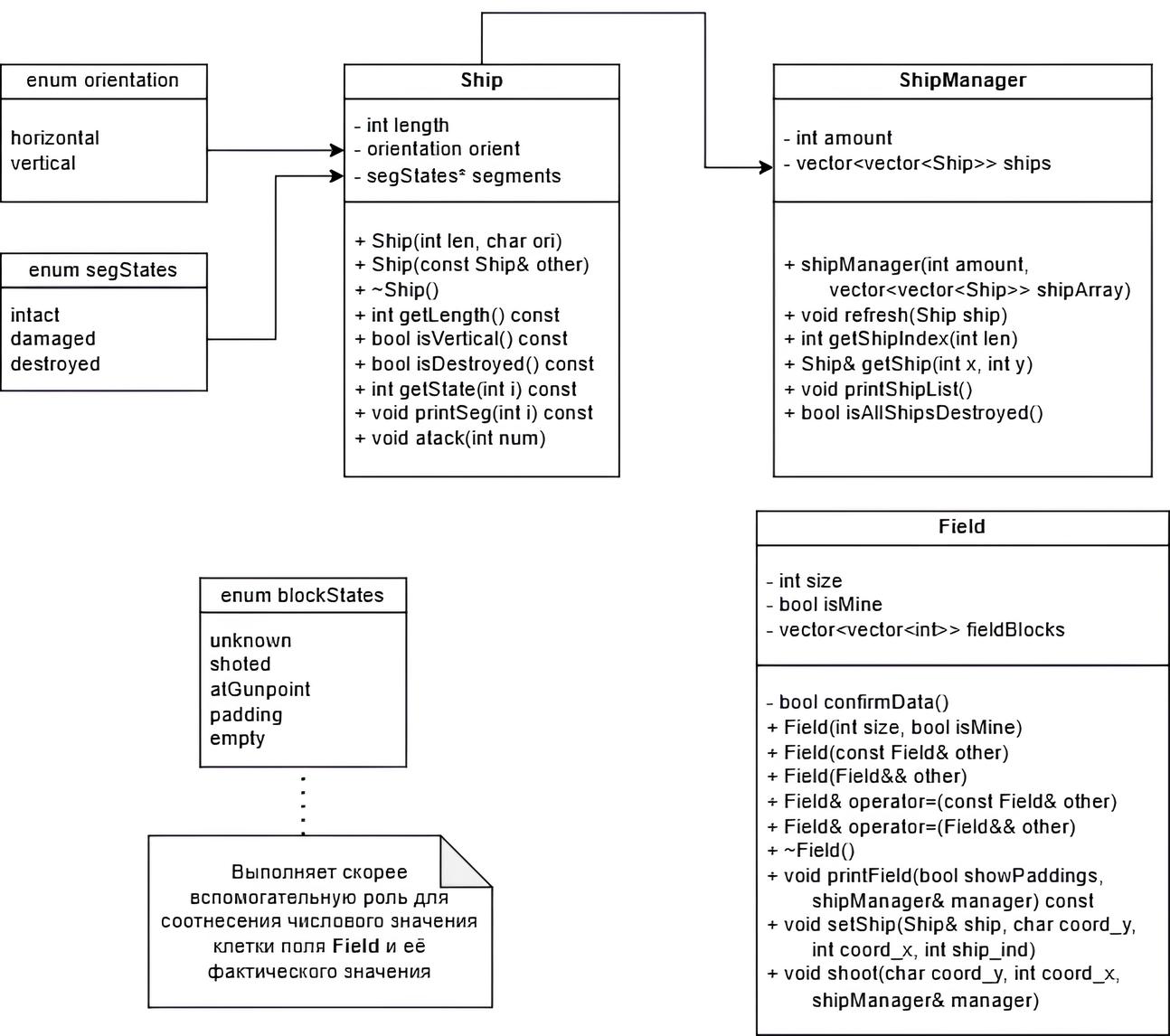


Рисунок 1. UML-диаграмма классов

Реализация корабля.

Для реализации корабля был создан класс **Ship** и два перечисляемых типа enum для состояния сегмента корабля и ориентации корабля:

* Значения enum **segStates:**
  + *intact* **–** сегмент корабля неповреждён;
  + *damaged*– сегмент корабля повреждён;
  + *destroyed* – сегмент корабля уничтожен.
* Возможные значения enum **orientation**:
  + *vertical* – вертикальное расположение корабля;
  + *horizontal* – горизонтальное расположение корабля.

Класс корабля имеет три приватных поля для хранения длины – **length**, положения корабля на поле – **orient** со значением *vertical* или *horizontal*, и массива сегментов – **segments**, состоящего из соответствующих состояний **segStates** и имеющего длину **length**.

В классе также реализованы следующие публичные методы:

* конструктор Ship(int len, char ori) – принимает на вход длину и расположения и создаёт объект корабля с заданными параметрами, а также массивом сегментов заданной длины, заполненным значением *intact* (все сегменты корабля изначально цельны). В случае некорректного значения длины или положения конструктор выкидывает исключение;
* конструктор копирования Ship(const Ship& other) – присваивает объекту значения иного объекта с глубоким копированием всех значений;
* деструктор ~Ship() – очищает память из-под массива сегментов после использования;
* метод getLength() – возвращает длину корабля;
* метод isVertical() – возвращает истину в случае, если корабль располагается вертикально, в противном случае возвращает ложь;
* метод isDestroyed() – возвращает истину, если все сегменты корабля имеют значение *destroyed*,то есть корабль полностью уничтожен. В противном случае возвращает ложь;
* метод getState(int i) – возвращает состояние i-го сегмента;
* метод printSeg(int i) – выводит на экран i-ый сегмент, в соответствие с его состоянием;
* метод atack(int num) – наносит урон по i-му сегменту, соответственно меняя состояние того.

Реализация менеджера кораблей.

Для реализации менеджера кораблей был создан класс **shipManager**. Класс менеджера кораблей имеет два приватных поля для хранения количества кораблей – **amount** – и двумерного вектора кораблей – **ships**. Корабли хранятся в двумерном векторе для упорядочивания последних по длине (номер строки соответствует длине корабля), данная структура была введена для удобства проверки кораблей на верное количество для каждый из длин.

В классе также реализованы следующие публичные методы:

* конструктор shipManager(int amount, std::vector<std::vector<Ship>> shipArray) – принимает на вход количество кораблей, а также сами корабли. Полученными значениями инициализируются соответствующие поля. По прямому назначению используется только для вражеского менеджера, потому проверки на корректность длин кораблей не имеется;
* метод refresh(Ship ship) – добавляет в уже существующий, но не заполненный окончательно набор кораблей новый, располагая его в соответствие с длиной, значение **amount** увеличивается на 1. В случае, если все корабли данной длины уже были введены, выкидывает исключение;
* метод isAllShipsEntered() – возвращает истину, если были инициализированы все корабли (в классических правилах «Морского боя» – 10 штук), в противном случае – ложь;
* метод getShipIndex(int len) – возвращает каким по счёту был инициализирован последний корабль заданной длины. Нужен для реализации привязки некоторой клетки поля к конкретному кораблю;
* метод Ship& getShip(int x, int y) – по заданным индексам возвращает ссылку на корабль;
* метод printShipList() – выводит на экран список кораблей пользователя, а также их состояние;
* метод isAllShipsDestroyed() – возвращает истину, если все корабли пользователя были уничтожены, в противном случае возвращает ложь.

Реализация игрового поля.

Для реализации игрового поля был создан класс **Field** и перечисляемый тип enum для состояния клетки игрового поля:

* Значения enum **blockStates:**
  + *unknown* **–** клетка поля, состояние которой неизвестно пользователю;
  + *shoted*– клетка поля была атакована и оказалась пуста;
  + *atGunpoint* – указывает на клетку, которую собирается атаковать пользователь. Необходимо для запроса подтверждения атаки;
  + *padding* – рядом с данной клеткой стоит корабль. Необходимо для валидного расположения корабля на поле;
  + *empty* – клетка поля пуста.

Данный состояния соответствуют некоторым отрицательным числовым значениям. Для хранения информации о клетках поля, на которых расположен корабль, используются положительные значения – трёхзначное число равное *длина\_корабля*&*номер\_корабля\_в\_менеджере*&*номер\_сегмента\_корабля*. Подобная структура реализована для осуществления быстрого доступа к сегменту корабля из менеджера по числовому значению клетки поля.

Класс игрового поля имеет три приватных поля для хранения размера – **size**, принадлежности поля пользователю (необходимо для режима отрисовки поля) – **isMine**, и двумерного вектора клеток поля – **fieldBlocks**, состоящего из соответствующих состояний **fieldStates** и имеющего размеры **size\*size**. Приватный метод **confirmData** получает у пользователя подтверждение установки корабля на некоторые координаты или совершения атаки.

В классе также реализованы следующие публичные методы:

* конструктор Field(int size, bool isMine) – инициализирует приватные поля размера и принадлежности поля; создаёт двумерный вектор **size**\***size**, изначально заполненный значением *empty* (все клетки пусты);
* деструктор ~Field() – очищает игровое поле;
* метод printField(bool showPaddings, shipManager& manager) – выводит поле на экран. Аргумент showPaddings скрывает или отображает отступы кораблей (для удобства расположения новых кораблей на поле во избежание пересечений), manager необходим для получения доступа к сегментам кораблей и корректного вывода их на экран в соответствие с состоянием;
* метод setShip(Ship& ship, char coord\_y, int coord\_x, int ship\_ind) – устанавливает корабль на заданные координаты начала корабля. В случае выхода за размеры поля, пересечения или соприкосновения с другим кораблём выкидывается ошибка. В соответствие с ориентацией корабля нужным клеткам присваивается расчётное положительное значение, а клеткам вокруг корабля отрицательное значение, соответствующее отступам *padding*;
* метод shoot(char coord\_y, int coord\_x, shipManager& manager) – производит атаку по заданным координатам. В случае выхода за границы поля или нацеливания на уже уничтоженные клетки поля выкидывает ошибку. В случае уничтожения сегмента корабля, корабля целиком или всей флотилии печатает соответствующее сообщение полльзователю.

Для класса игрового поля также реализованы конструкторы копирования и перемещения, а также соответствующие им операторы присваивания: Field(const Field& other), Field(Field&& other), Field& operator=(const Field& other) и Field& operator=(Field&& other).

В работе также используются управляющие последовательности ANSI для управления графическим представлением символов. Реализовано для более понятного пользовательского интерфейса.

**Выводы**

Были реализованы классы корабля, менеджера кораблей и игрового поля с соответствующими методами и конструкторами. Программа также была протестирована и отлажена для проверки на корректность.

# **ПРИЛОЖЕНИЕ А ИСХОДНЫЙ КОД ПРОГРАММЫ**

**main.cpp**

#include <iostream>

#include <climits>

#include <ctime>

#include "ship.h"

#include "shipManager.h"

#include "field.h"

#define TERM\_GREEN "\033[32m"

#define TERM\_RED "\033[31m"

#define TERM\_DEF "\033[0m"

#define FIELD\_SZ 10

// works only on linux

typedef struct {

int x;

char y;

} coords\_t;

shipManager createEnemyField(Field\* enemyField) {

std::vector<std::vector<Ship>> enemyShips(4);

shipManager manager = shipManager(0, enemyShips);

int x, y;

int amount = 0;

for (int len = 0; len < enemyShips.size(); len++) {

for (int count = 0; count < (4 - len); count++) {

char ori = (len%2) ? 'h' : 'v';

Ship tmp(len+1, ori);

enemyShips[len].push\_back(tmp);

//srand(time(0));

while (1) {

y = std::rand()/(RAND\_MAX/FIELD\_SZ);

x = std::rand()/(RAND\_MAX/FIELD\_SZ);

try {

enemyField->setShip(tmp, (char)(y + 97), x, count+1);

} catch (const char\* err) {

continue;

}

break;

}

amount++;

}

}

manager = shipManager(amount, enemyShips);

return manager;

}

int main() {

Field enemyField(FIELD\_SZ, false);

// change to true to see field and skipi confirmation

shipManager enemyManager = createEnemyField(&enemyField);

Field playersField(FIELD\_SZ, true);

std::vector<std::vector<Ship>> playersShips;

shipManager playersManager(0, playersShips);

int shipAmount = 0;

int len = 4;

Ship ship1(len, 'v');

playersManager.refresh(ship1);

shipAmount++;

try {

Ship shipErr(5, 'h');

playersManager.refresh(shipErr);

shipAmount++;

} catch (const char\* err) {

std::cerr << TERM\_RED << err << TERM\_DEF << std::endl; // Improper length of ship

}

try {

Ship shipErr(0, 'h');

playersManager.refresh(shipErr);

shipAmount++;

} catch (const char\* err) {

std::cerr << TERM\_RED << err << TERM\_DEF << std::endl; // Improper length of ship

}

try {

Ship shipErr(2, 's');

playersManager.refresh(shipErr); // Improper orientation

shipAmount++;

} catch (const char\* err) {

std::cerr << TERM\_RED << err << TERM\_DEF << std::endl;

}

try {

Ship shipErr(len, 'h');

playersManager.refresh(shipErr); // All ships of this length are already in the list

shipAmount++;

} catch (const char\* err) {

std::cerr << TERM\_RED << err << TERM\_DEF << std::endl;

}

len = 2;

Ship ship2(len, 'h');

playersManager.refresh(ship2);

shipAmount++;

try {

playersField.setShip(ship1, 'c', 8, playersManager.getShipIndex(len));

} catch (const char\* err) {

std::cerr << TERM\_RED << err << TERM\_DEF << std::endl; // Coordinations out of field

}

playersField.setShip(ship1, 'c', 3, playersManager.getShipIndex(len));

try {

playersField.setShip(ship2, 'd', 6, playersManager.getShipIndex(len)); // You can't put a ship right next to another one

} catch (const char\* err) {

std::cerr << TERM\_RED << err << TERM\_DEF << std::endl;

}

try {

playersField.setShip(ship2, 'j', 10, playersManager.getShipIndex(len)); // Coordinates out of field!

} catch (const char\* err) {

std::cerr << TERM\_RED << err << TERM\_DEF << std::endl;

}

try {

playersField.setShip(ship2, 'c', 6, playersManager.getShipIndex(len)); // There's already a ship here!

} catch (const char\* err) {

std::cerr << TERM\_RED << err << TERM\_DEF << std::endl;

}

playersField.setShip(ship2, 'i', 10, playersManager.getShipIndex(len));

enemyField.shoot('g', 5, enemyManager);

enemyField.shoot('e', 5, enemyManager);

enemyField.shoot('i', 3, enemyManager);

enemyField.shoot('i', 3, enemyManager);

try {

enemyField.shoot('k', 5, enemyManager);

} catch(const char\* err) {

std::cerr << TERM\_RED << err << TERM\_DEF << std::endl; // Coordinates out of field

}

try {

enemyField.shoot('a', 11, enemyManager);

} catch(const char\* err) {

std::cerr << TERM\_RED << err << TERM\_DEF << std::endl; // Coordinates out of field

}

try {

enemyField.shoot('i', 3, enemyManager);

} catch(const char\* err) {

std::cerr << TERM\_RED << err << TERM\_DEF << std::endl; // You already destroyed the ship segment at these coordinates!

}

std::cout << "Your ships: " << std::endl;

playersManager.printShipList();

std::cout << "Your field: " << std::endl;

playersField.printField(true, playersManager);

std::cout << std::endl;

std::cout << "Enemy ships: " << std::endl;

enemyManager.printShipList();

std::cout << "Enemy field: " << std::endl;

enemyField.printField(false, enemyManager);

return 0;

}

**ship.h**

#ifndef SHIP\_H

#define SHIP\_H

enum segStates {intact, damaged, destroyed};

enum orientation {horizontal, vertical};

class Ship

{

private:

int length;

orientation orient;

segStates\* segments = nullptr;

public:

Ship(int len, char ori);

Ship(const Ship& other);

~Ship();

int getLength() const;

bool isVertical() const;

bool isDestroyed() const;

int getState(int i) const;

void printSeg(int i) const;

void atack(int i);

};

#endif

**ship.cpp**

#include <iostream>

#include "ship.h"

#define TERM\_DEF "\033[0m"

#define TERM\_CROSSED "\033[9m"

#define TERM\_GREEN "\033[32m"

#define TERM\_RED "\033[31m"

Ship::Ship(int len, char ori) {

if ((len < 1) || (len > 4))

throw "Improper length of ship!";

length = len;

segments = new segStates[length];

for (int i = 0; i < length; i++)

segments[i] = segStates::intact;

if ((ori != 'h') && (ori != 'v'))

throw "Improper orientation!";

orient = (ori == 'h') ? orientation::horizontal : orientation::vertical;

}

Ship::~Ship() {

if (segments != nullptr)

delete[] segments;

}

Ship::Ship(const Ship& other) : length(other.length), orient(other.orient) {

delete[] segments;

if (other.segments != nullptr) {

segments = new segStates[length];

for(int i = 0; i < length; i++)

segments[i] = other.segments[i];

}

}

int Ship::getLength() const {

return length;

}

bool Ship::isVertical() const {

return orient == orientation::vertical;

}

bool Ship::isDestroyed() const {

for (int i = 0; i < length; i++) {

if (segments[i] != segStates::destroyed)

return false;

}

return true;

}

int Ship::getState(int i) const {

return segments[i-1];

}

void Ship::printSeg(int i) const {

if (segments[i-1] != segStates::intact) {

std::cout << TERM\_RED;

if (segments[i-1] == segStates::destroyed)

std::cout << TERM\_CROSSED;

}

std::cout << 'X' << TERM\_DEF;

}

void Ship::atack(int i) {

i--;

switch(segments[i]) {

case segStates::intact:

segments[i] = segStates::damaged;

std::cout << TERM\_GREEN << "Congratulations! You're damaged the ship!" << TERM\_DEF << std::endl;

break;

case segStates::damaged:

segments[i] = segStates::destroyed;

std::cout << TERM\_GREEN << "Congratulations! You destroyed the ship segment!" << TERM\_DEF << std::endl;

break;

default:

throw "An error occured...";

}

}

**shipManager.h**

#ifndef SHIP\_MANAGER\_H

#define SHIP\_MANAGER\_H

#include <vector>

#include "ship.h"

class shipManager

{

private:

int amount;

std::vector<std::vector<Ship>> ships;

public:

shipManager(int amount, std::vector<std::vector<Ship>> shipArray);

void refresh(Ship ship);

bool isAllShipsEntered() const;

int getShipIndex(int len) const;

Ship& getShip(int x, int y);

void printShipList() const;

bool isAllShipsDestroyed() const;

};

#endif

**shipManager.cpp**

#include <iostream>

#include <iomanip>

#include "shipManager.h"

#define MAX\_SHIP\_COUNT 10

#define TERM\_UNDERLINE "\033[4m"

#define TERM\_DEF "\033[0m"

shipManager::shipManager(int am, std::vector<std::vector<Ship>> shipArray) : amount{am} {

ships.resize(shipArray.size());

for (int x = 0; x < shipArray.size(); x++) {

for (int y = 0; y < shipArray[x].size(); y++) {

ships[x].push\_back(shipArray[x][y]);

}

}

}

void shipManager::refresh(Ship ship) {

int len = ship.getLength();

if (ships.size() < len)

ships.resize(len);

if (ships[len-1].size() >= (5 - len))

throw "All ships of this length are already in the list! ";

ships[len-1].push\_back(ship);

amount++;

}

bool shipManager::isAllShipsEntered() const {

return (amount == MAX\_SHIP\_COUNT);

}

int shipManager::getShipIndex(int len) const {

return ships[len-1].size();

}

Ship& shipManager::getShip(int x, int y) {

return ships[x-1][y-1];

}

void shipManager::printShipList() const {

std::cout << TERM\_UNDERLINE << "\t № |";

for (int i = 1; i < amount+1; i++)

std::cout << " " << i << " |";

std::cout << TERM\_DEF << std::endl;

std::cout << "\tShip |";

for (int x = 0; x < ships.size(); x++) {

for (int y = 0; y < ships[x].size(); y++) {

std::cout << std::setw(2-x/2) << ' ';

for (int i = 1; i <= x+1; i++)

ships[x][y].printSeg(i);

std::cout << std::setw(2-(x+1)/2) << ' ';

std::cout << "|";

}

}

std::cout << std::endl;

}

bool shipManager::isAllShipsDestroyed() const {

for (int x = 0; x < ships.size(); x++) {

for (int y = 0; y < ships[x].size(); y++) {

if (!ships[x][y].isDestroyed())

return false;

}

}

return true;

}

field.h

#ifndef FIELD\_H

#define FIELD\_H

#include "ship.h"

#include "shipManager.h"

enum blockStates {unknown=-4, shoted=-3, atGunpoint=-2, padding=-1, empty=0};

class Field

{

private:

int size;

bool isMine;

std::vector<std::vector<int>> fieldBlocks;

bool confirmData() const;

public:

Field(int size, bool isMine);

Field(const Field& other);

Field(Field&& other);

Field& operator=(const Field& other);

Field& operator=(Field&& other);

~Field();

void printField(bool showPaddings, shipManager& manager) const;

void setShip(Ship& ship, char coord\_y, int coord\_x, int ship\_ind);

void shoot(char coord\_y, int coord\_x, shipManager& manager);

};

#endif

field.cpp

#include <iostream>

#include <climits>

#include "field.h"

#define TERM\_DEF "\033[0m"

#define TERM\_UNDERLINE "\033[4m"

#define TERM\_RED "\033[31m"

#define TERM\_YELLOW "\033[33m"

#define TERM\_RED\_BG "\033[101m"

Field::Field(int sz, bool isMine) : size{sz}, isMine{isMine} {

fieldBlocks.resize(size);

for (int x = 0; x < size; x++) {

for (int y = 0; y < size; y++)

fieldBlocks[x].push\_back(blockStates::empty);

}

}

Field::~Field() {}

Field::Field(const Field& other) : size(other.size), isMine(other.isMine), fieldBlocks(other.fieldBlocks) { }

Field::Field(Field&& other) : size(other.size), isMine(other.isMine), fieldBlocks(std::move(other.fieldBlocks)) {

other.size = 0;

other.isMine = false;

}

Field& Field::operator = (const Field& other) {

if (this != &other) {

size = other.size;

isMine = other.isMine;

fieldBlocks = other.fieldBlocks;

}

return \*this;

}

Field& Field::operator = (Field&& other) {

if (this != &other) {

size = other.size;

isMine = other.isMine;

fieldBlocks = std::move(other.fieldBlocks);

other.size = 0;

other.isMine = 0;

}

return \*this;

}

bool Field::confirmData() const{

std::cout << "Do you agree? (enter N to replace ship, Y or ENTER to continue): ";

char ans = getchar();

while ((tolower(ans) != 'n') && (ans != '\n') && (tolower(ans) != 'y')) {

std::cerr << TERM\_RED << "Incorrect value. Try again: " << TERM\_DEF;

std::cin.clear();

std::cin.ignore(LONG\_MAX, '\n');

ans = getchar();

}

return (tolower(ans) != 'n');

}

void Field::setShip(Ship& ship, char coord\_y, int coord\_x, int ship\_ind) {

std::vector<std::vector<int>> copyField = fieldBlocks;

if (!isalpha(coord\_y))

throw "Incorrect first coordinate! Must be a letter! ";

int x = coord\_x - 1;

int y = (int)coord\_y - 96 - 1;

int len = ship.getLength();

bool ori = ship.isVertical();

int max\_x = x+len\*ori;

int max\_y = y+len\*!ori;

if ((x < 0) || (max\_x > size) || (y < 0) || (max\_y > size))

throw "Coordinates out of field! ";

int x1, y1;

// coords for loop

for (int i = -1; i < len+1; i++) {

x1 = x + i\*ori;

y1 = y + i\*!ori;

if ((x1-!ori > -1) && (x1-!ori < size) && (y1-ori > -1) && (y1-ori < size))

copyField[x1-!ori][y1-ori] = blockStates::padding;

if ((x1+!ori > -1) && (x1+!ori < size) && (y1+ori > -1) && (y1+ori < size))

copyField[x1+!ori][y1+ori] = blockStates::padding;

if ((i != -1) && (i != len)) {

if (fieldBlocks[x1][y1] == blockStates::padding)

throw "You can't put a ship right next to another one! ";

if (fieldBlocks[x1][y1] > 0)

throw "There's already a ship here! ";

copyField[x1][y1] = len\*100 + ship\_ind\*10 + (i+1);

} else if ((x1 > -1) && (x1 < size) && (y1 > -1) && (y1 < size)) {

copyField[x1][y1] = blockStates::padding;

}

}

// if (isMine && !getAgreement())

// throw "Please, re-enter your coordinates: ";

fieldBlocks = copyField;

}

void Field::printField(bool showPaddings, shipManager& manager) const {

std::cout << TERM\_UNDERLINE << "\t | ";

for (int i = 0; i < size; i++)

std::cout << (char)(i+97) << ' ';

std::cout << TERM\_DEF;

std::cout << std::endl;

for (int x = 0; x < size; x++) {

std::cout << '\t';

if (x+1 < 10)

std::cout << ' ';

std::cout << x+1 << "| ";

for (int y = 0; y < size; y++) {

switch(fieldBlocks[x][y]) {

case -3: // shoted

std::cout << TERM\_RED << '\*' << TERM\_DEF;

break;

case -2: // at gunpoint

std::cout << TERM\_RED\_BG << '+' << TERM\_DEF;

break;

case -1: // padding

if (showPaddings && isMine) {

std::cout << '\\';

break;

}

case 0: // empty

std::cout << (isMine ? '~' : '?');

break;

default:

Ship ship = manager.getShip(fieldBlocks[x][y]/100, (fieldBlocks[x][y]%100)/10);

int segState = ship.getState(fieldBlocks[x][y]%10);

if (isMine || segState != segStates::intact) {

ship.printSeg(fieldBlocks[x][y]%10);

} else {

std::cout << (isMine ? '~' : '?');

}

}

std::cout << ' ';

}

std::cout << std::endl;

}

}

void Field::shoot(char coord\_y, int coord\_x, shipManager& manager) {

int x = coord\_x - 1;

int y = (int)coord\_y - 96 - 1;

if ((x < 0) || (x > size-1) || (y < 0) || (y > size-1))

throw "Coordinates out of field! ";

if (fieldBlocks[x][y] > 0) {

Ship ship = manager.getShip(fieldBlocks[x][y]/100, (fieldBlocks[x][y]%100)/10);

if (ship.getState(fieldBlocks[x][y]%10) == segStates::destroyed)

throw "You already destroyed ship segment at these coordinates! ";

} else {

if (fieldBlocks[x][y] == blockStates::shoted)

throw "You can't shoot at these coordinates! ";

}

if (!isMine) {

int tmp = fieldBlocks[x][y];

fieldBlocks[x][y] = blockStates::atGunpoint;

printField(false, manager);

if (!confirmData()) {

fieldBlocks[x][y] = tmp;

throw "Please, re-enter your coordinates: ";

}

fieldBlocks[x][y] = tmp;

}

if (fieldBlocks[x][y] > 0) {

Ship& ship = manager.getShip(fieldBlocks[x][y]/100, (fieldBlocks[x][y]%100)/10);

ship.atack(fieldBlocks[x][y]%10);

if (ship.isDestroyed()) {

bool ori = ship.isVertical();

int x1, y1;

for (int i = -1; i < ship.getLength()+1; i++) {

x1 = x + i\*ori;

y1 = y + i\*!ori;

if ((x1-!ori > -1) && (x1-!ori < size) && (y1-ori > -1) && (y1-ori < size))

fieldBlocks[x1-!ori][y1-ori] = blockStates::shoted;

if ((x1+!ori > -1) && (x1+!ori < size) && (y1+ori > -1) && (y1+ori < size))

fieldBlocks[x1+!ori][y1+ori] = blockStates::shoted;

if ((i == -1) || (i == ship.getLength()) && (x1 > -1) && (x1 < size) && (y1 > -1) && (y1 < size))

fieldBlocks[x1][y1] = blockStates::shoted;

}

if (manager.isAllShipsDestroyed()) {

if (isMine) {

std::cout << TERM\_RED << "\tYOU'RE LOSE..." << TERM\_DEF << std::endl;

exit(EXIT\_SUCCESS);

} else {

std::cout << TERM\_YELLOW << "\tYOU'RE WON!!!" << TERM\_DEF << std::endl;

exit(EXIT\_SUCCESS);

}

}

}

} else {

fieldBlocks[x][y] = blockStates::shoted;

}

}

Makefile

all: main.o ship.o ship-manager.o field.o

g++ main.o ship.o ship-manager.o field.o -o lb

main.o: main.cpp field.h shipManager.h ship.h

g++ -lstdc++ -c main.cpp

ship.o: ship.cpp ship.h

g++ -c ship.cpp

ship-manager.o: shipManager.cpp shipManager.h ship.h

g++ -c shipManager.cpp

field.o: field.cpp field.h shipManager.h ship.h

g++ -c field.cpp

clean:

rm -f ./\*.o lb

# **ПРИЛОЖЕНИЕ Б ТЕСТИРОВАНИЕ ПРОГРАММЫ**

Таблица 1 – Тестирование программы

|  |  |  |  |
| --- | --- | --- | --- |
| № п/п | Входные данные | Выходные данные | Комментарии |
| 1. | Ship ship1(4, 'v');  playersManager.refresh(ship1);  Ship shipErr(5, 'h');  playersManager.refresh(shipErr);  Ship shipErr(0, 'h');  playersManager.refresh(shipErr);  Ship shipErr(2, 's');  playersManager.refresh(shipErr);  Ship shipErr(4, 'h');  playersManager.refresh(shipErr); | Improper length of ship!  Improper length of ship!  Improper orientation!  All ships of this length are already in the list! | Корабли корректно создаются и добавляются в менеджер.  Валидация входных данных также работает корректно.  При попытке пользователя добавить кораблей больше возможного возникает ошибка. |
| 2. | Ship ship1(4, 'v');  playersManager.refresh(ship1);  Ship ship2(2, 'h');  playersManager.refresh(ship2);  playersField.setShip(ship1, 'c', 8, playersManager.getShipIndex(len));  playersField.setShip(ship1, 'c', 3, playersManager.getShipIndex(len));  playersField.setShip(ship2, 'd', 6, playersManager.getShipIndex(len));  playersField.setShip(ship2, 'j', 10, playersManager.getShipIndex(len));  playersField.setShip(ship2, 'c', 6, playersManager.getShipIndex(len)); | Coordinates out of field!  You can't put a ship right next to another one!  Coordinates out of field!  There's already a ship here! | Программа корректно располагает корабли на игровом поле.  В случае выхода корабля за пределы игрового поля или попытки установить корабль поверх другого или рядом с другим вызываются соответствующие ошибки. |
| 3. | enemyField.shoot('g', 5, enemyManager);  enemyField.shoot('e', 5, enemyManager);  enemyField.shoot('i', 3, enemyManager);  enemyField.shoot('i', 3, enemyManager);  enemyField.shoot('k', 5, enemyManager);  enemyField.shoot('a', 11, enemyManager);  enemyField.shoot('i', 3, enemyManager); | Congratulations! You're damaged the ship!  Congratulations! You're damaged the ship!  Congratulations! You destroyed the ship segment!  Coordinates out of field!  Coordinates out of field!  You already destroyed ship segment at these coordinates! | Атака работает корректно.  При повреждение корабля пользователю об этом сообщается.  При попытке нанести урон по клетке поля, находящейся за пределами поля или по уже атакованной клетке, вызывается соответствующая ошибка. |

Пример пользовательского интерфейса см. на рисунке 2:

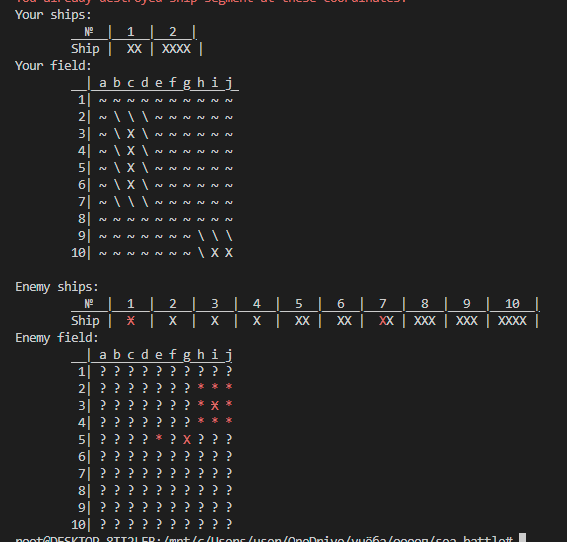


Рисунок 2. Пример пользовательского интерфейса