**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. 3341 |  | Шаповаленко Е.В. |
| Преподаватель |  | Жангиров Т.Р. |

Санкт-Петербург

2024

## Цель работы

Цель работы – создание классов корабля, менеджера кораблей и поля и связей между ними.

Для достижения поставленной цели требуется:

1. Ознакомиться с понятием класс и его структурой;
2. Реализовать указанные классы согласно требованиям из условия;
3. Написать программу, в которой бы проводилась проверка работоспособности классов.

## 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.

В) 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

## Выполнение работы

Реализован класс *Ship*, моделирующий корабль в игре, представляя его размеры, состояние самого корабля и его сегментов:

1) Конструктор *Ship(int size)* позволяет создать корабль с определенным размером. Согласно условию размер корабля может варьироваться от 1 до 4.

2) Метод *getSize()* позволяет получить размер корабля, что может быть необходимо для итерации по его сегментам.

3) Метод *getSegmentStatus(int index)* возвращает состояние сегмента по индексу (*intact*, *damaged*, *destroyed*). Состояние сегмента зависит от его здоровья. Это необходимо для вывода информации пользователю.

4) Метод *getShipStatus()* возвращает состояние корабля (*dead*, *alive*). Это необходимо для вывода информации пользователю.

5) Метод *damageSegment(int index, int damage)* симулирует нанесение определенного урона выбранному сегменту.

6) Метод *healSegment(int index, int heal)* симулирует лечение на определенную величину выбранного сегмента. Данный метод противоположен по действию предыдущему.

7) Главным полем *Ship* является *vector<ShipSegment> segments\_*. Оно хранит сегменты корабля. Для простоты работы реализовано поле *size\_* для хранения размера корабля.

8) Переменные *kMinSize* и *kMaxSize* задают минимально и максимально допустимые размеры корабля соответственно.

Таким образом, класс *Ship* моделирует корабль в целом. Для моделирования его сегментов в классе *Ship* реализован подкласс *ShipSegment*:

1) Поле *health*\_ хранит здоровье сегмента. При создании объекта класса полю присваивается значение максимального здоровья *kMaxHealth*.

2) Метод *takeDamage(int damage)* симулирует нанесение определенного урона выбранному сегменту. Урон не может быть отрицательным.

3) Метод *takeHeal(int heal)* симулирует лечение на определенную величину выбранного сегмента. Величина лечения не может быть отрицательной. Данный метод противоположен по действию предыдущему.

4) Метод *getStatus()* возвращает состояние сегмента (*intact*, *damaged*, *destroyed*). Состояние сегмента зависит от значения *health\_*. Это необходимо для вывода информации пользователю.

Реализован класс ShipManager. Этот класс создает корабли и расставляет их на указанном поле. Также через него можно получить информацию о состоянии кораблей. Реализованы следующие поля и методы:

1) Конструктор *ShipManager(std::initializer\_list<int> ship\_sizes)* создает корабли в соответствии с поданными на вход размерами и сохраняет их в поле *list<pair<Ship, bool>> ships\_*. В этом поле хранятся пары: корабль и был ли он размещен на поле.

2) Метод *addShip(int ship\_size)* создает корабль в соответствии с поданным на вход размером и сохраняет его в поле *list<pair<Ship, bool>> ships\_*.

3) Метод *placeShip(Field& field, int index, int x, int y, ShipOrientation orientation)* размещает корабль по индексу *index* в *ships\_* на игровом поле *field* в соответствии с координатами *x* и *y* и ориентацией корабля *orientation*. Корабль отмечается размещенным на игровом поле в *ships\_*.

4) Методы *getUsedShips()* и *getUnusedShips()* позволяют получить векторы, содержащие копии использованных и неиспользованных кораблей соответственно. Это может понадобиться для отрисовки кораблей в GUI/CLI.

5) Методы *getUsedShipsSize()* и *getUnusedShipsSize()* позволяют получить количество использованных и неиспользованных кораблей соответственно, что может быть необходимо для итерации по ним.

6) Отладочный метод *printShips()* выводит на экран *ships\_*. С помощью него можно отслеживать какие корабли были размещены на поле, а какие нет, и состояние их сегментов.

Таким образом, при помощи класса *ShipManager* реализовано взаимодействие между кораблями и полем, а также в будущем между пользователем и кораблями.

Реализован класс *Field*, моделирующий игровое поле, представляя его размеры и состояние его клеток:

1) Конструктор *Field(int size\_x, int size\_y)* создает игровое поле в соответствии с размерами *size\_x* и *size\_y* и сохраняет его в поле *vector<vector<FieldCell>>* *field\_*. Также сохраняются размеры в *size\_x\_* и *size\_y\_* соответственно.

2) Реализованы конструкторы копирования и перемещения и соответствующие им операторы.

3) Метод *placeShip(Ship\* ship, int x, int y, ShipOrientation orientation)* размещает на игровом поле переданный корабль ship в соответствии с координатами *x* и *y* и ориентацией корабля *orientation*. Корабли не могут выходить за рамки игрового поля и пересекаться или касаться друг друга.

4) С помощью метода *attackCell(int x, int y, int damage)* атакуется выбранная клетка игрового поля.

5) Отладочный метод *printField()* выводит на экран *field\_*. С помощью него можно отслеживать состояние клеток игрового поля.

Таким образом, класс *Field* моделирует игровое полке в целом. Для моделирования его клеток в классе *Field* реализован подкласс *FieldCell*:

1) Конструктор *FieldCell()* инициализирует поля клетки игрового поля *status\_*, *ship\_* и *ship\_segment\_index\_* значениями *unknown*, *nullptr* и -1 соответственно. Поле *status\_* хранит состояние клетки (*unknown*, *empty*, *ship*). Поля *ship\_* и *ship\_segment\_index\_* хранят указатель на корабль и индекс сегмента в этом корабле соответственно. Они нужны для реализации атаки корабля с игрового поля.

2) Метод *getStatus()* возвращает состояние клетки (*unknown*, *empty*, *ship*), в зависимости от того, была ли атакована клетка или нет и есть ли на ней корабль или нет.

3) Метод *setStatus(FieldCellStatus status)* задает статус клетке игрового поля.

4) С помощью метода *attackCell(int damage)* атакуется сегмент корабля, расположенный на данной клетке игрового поля.

5) Метод *isShip()* возвращает информацию о том, есть ли на данной клетке игрового поля корабль или нет.

6) Метод *setShipSegment(Ship\* ship, int index)* задает поля *ship\_* и *ship\_segment\_index\_*.

7) Метод *getShipSegmentStatus()* возвращает состояние сегмента корабля, который расположен в данной клетке игрового поля.

Таким образом, при помощи класса *Field* реализовано взаимодействие между кораблями и полем.

В местах, где может возникнуть ошибка (выход за допустимые границы, недопустимое значение переменной, нарушение логики программы), программа завершает работу с сообщением о соответствующей ошибке.

Ниже представлена UML-диаграмма реализованных в данной работе классов:

|  |
| --- |
|  |
| Рисунок 1: UML-диаграмма классов |

Разработанный программный код см. в приложении А.

Тестирование программы см. в приложении Б.

## Выводы

Цель работы – создание классов корабля, менеджера кораблей и поля и связей между ними – была достигнута.

В ходе выполнения работы были выполнены следующие задачи:

1. Было проведено ознакомление с понятием класса и его структурой;
2. Были реализованы указанные классы согласно требованиям из условия;
3. Написана программа, в которой проводится проверка работоспособности классов.

.

# Приложение А Исходный код программы

Название файла: ship.h

#ifndef SHIP

#define SHIP

#include <vector>

#include <stdexcept>

enum class ShipStatus : int

{

dead,

alive

};

enum class ShipSegmentStatus : int

{

destroyed,

damaged,

intact

};

class Ship

{

public:

Ship();

explicit Ship(int size);

~Ship();

int getSize() const noexcept;

ShipSegmentStatus getSegmentStatus(int index) const;

ShipStatus getShipStatus() const noexcept;

void damageSegment(int index, int damage);

void healSegment(int index, int heal);

private:

class ShipSegment

{

public:

ShipSegment();

~ShipSegment();

void takeDamege(int damage);

void takeHeal(int heal);

ShipSegmentStatus getStatus() const noexcept;

private:

int kMaxHealth = 2;

int health\_;

};

int kMinSize = 1;

int kMaxSize = 4;

int size\_;

std::vector<ShipSegment> segments\_;

};

#endif

Название файла: ship.cpp

#include "ship.h"

#include <vector>

#include <stdexcept>

Ship::Ship() = default;

Ship::Ship(int size)

{

if (size < kMinSize || size > kMaxSize) {

throw std::logic\_error("Ship size can be from 1 to 4");

}

size\_ = size;

for (int i = 0; i < size\_; i++) {

segments\_.push\_back(ShipSegment());

}

}

Ship::~Ship()

{

segments\_.clear();

}

int Ship::getSize() const noexcept

{

return size\_;

}

ShipSegmentStatus Ship::getSegmentStatus(int index) const

{

if (index < 0 || index >= size\_) {

throw std::out\_of\_range("Ship segment index out of range");

}

return segments\_[index].getStatus();

}

ShipStatus Ship::getShipStatus() const noexcept

{

int dead\_flag = 1;

for (int i = 0; i < size\_; i++) {

if (segments\_[i].getStatus() != ShipSegmentStatus::destroyed) {

dead\_flag = 0;

break;

}

}

if (dead\_flag) {

return ShipStatus::dead;

} else {

return ShipStatus::alive;

}

}

void Ship::damageSegment(int index, int damage)

{

if (index < 0 || index >= size\_) {

throw std::out\_of\_range("Ship segment index out of range");

}

segments\_[index].takeDamege(damage);

}

void Ship::healSegment(int index, int heal)

{

if (index < 0 || index >= size\_) {

throw std::out\_of\_range("Ship segment index out of range");

}

segments\_[index].takeHeal(heal);

}

Ship::ShipSegment::ShipSegment()

{

health\_ = kMaxHealth;

}

Ship::ShipSegment::~ShipSegment() = default;

void Ship::ShipSegment::takeDamege(int damage)

{

if (damage < 0) {

throw std::invalid\_argument("Damage can't be negative");

}

health\_ = std::max(0, health\_ - damage);

}

void Ship::ShipSegment::takeHeal(int heal)

{

if (heal < 0) {

throw std::invalid\_argument("Heal can't be negative");

}

health\_ = std::min(kMaxHealth, health\_ + heal);

}

ShipSegmentStatus Ship::ShipSegment::getStatus() const noexcept

{

if (health\_ == kMaxHealth) {

return ShipSegmentStatus::intact;

} else if (health\_ == 0) {

return ShipSegmentStatus::destroyed;

} else {

return ShipSegmentStatus::damaged;

}

}

Название файла: shipManager.h

#ifndef SHIP\_MANAGER

#define SHIP\_MANAGER

#include <iostream>

#include <initializer\_list>

#include <vector>

#include <list>

#include <utility>

#include "ship.h"

#include "field.h"

class ShipManager

{

public:

ShipManager();

explicit ShipManager(std::initializer\_list<int> ship\_sizes);

~ShipManager();

void printShips() const noexcept;

void placeShip(Field& field, int index, int x, int y, ShipOrientation orientation);

void addShip(int ship\_size);

std::vector<Ship> getUnusedShips() const noexcept;

std::vector<Ship> getUsedShips() const noexcept;

int getUnusedShipsSize() const noexcept;

int getUsedShipsSize() const noexcept;

private:

std::list<std::pair<Ship, bool>> ships\_;

};

#endif

Название файла: shipManager.cpp

#include "shipManager.h"

#include "ship.h"

#include "field.h"

#include <iostream>

#include <initializer\_list>

#include <vector>

#include <utility>

ShipManager::ShipManager() = default;

ShipManager::ShipManager(std::initializer\_list<int> ship\_sizes)

{

for (auto ship : ship\_sizes) {

ships\_.push\_back({Ship(ship), false});

}

}

ShipManager::~ShipManager()

{

ships\_.clear();

}

void ShipManager::printShips() const noexcept

{

int counter = 0;

for (auto ship: ships\_) {

std::cout << "Ship " << counter++ << ": ";

for (int i = 0; i < ship.first.getSize(); i++) {

if (ship.first.getSegmentStatus(i) == ShipSegmentStatus::intact) {

std::cout << "[2]";

} else if (ship.first.getSegmentStatus(i) == ShipSegmentStatus::damaged) {

std::cout << "[1]";

} else {

std::cout << "[0]";

}

}

if (ship.second) {

std::cout << " Used" << "\n";

} else {

std::cout << " Unused" << "\n";

}

}

if (ships\_.size() == 0) {

std::cout << "None" << "\n";

}

std::cout << "\n";

return;

}

void ShipManager::placeShip(Field& field, int index, int x, int y, ShipOrientation orientation)

{

if (index < 0 || index >= ships\_.size()) {

throw std::out\_of\_range("Index out of range");

}

std::list<std::pair<Ship, bool>>::iterator it = ships\_.begin();

std::advance(it, index);

if (it->second) {

throw std::logic\_error("Ship was already placed to field");

}

it->second = true;

field.placeShip(&(it->first), x, y, orientation);

}

void ShipManager::addShip(int ship\_size)

{

ships\_.push\_back({Ship(ship\_size), false});

}

std::vector<Ship> ShipManager::getUnusedShips() const noexcept

{

std::vector<Ship> unused\_ships;

for(auto ship: ships\_)

if(ship.second == false)

unused\_ships.push\_back(ship.first);

return unused\_ships;

}

std::vector<Ship> ShipManager::getUsedShips() const noexcept

{

std::vector<Ship> used\_ships;

for(auto ship: ships\_)

if(ship.second == false)

used\_ships.push\_back(ship.first);

return used\_ships;

}

int ShipManager::getUnusedShipsSize() const noexcept

{

int unused\_ships\_num;

for(auto ship: ships\_)

if(ship.second == false)

unused\_ships\_num++;

return unused\_ships\_num;

}

int ShipManager::getUsedShipsSize() const noexcept

{

int used\_ships\_num;

for(auto ship: ships\_)

if(ship.second == false)

used\_ships\_num++;

return used\_ships\_num;

}

Название файла: field.h

#ifndef FIELD

#define FIELD

#include <iostream>

#include <vector>

#include <algorithm>

#include <stdexcept>

#include "ship.h"

enum class ShipOrientation : int

{

horizontal,

vertical

};

enum class FieldCellStatus : int

{

unknown,

empty,

ship

};

class Field

{

public:

Field();

explicit Field(int size\_x, int size\_y);

Field(const Field& other);

Field& operator=(const Field& other);

Field(Field&& other);

Field& operator=(Field&& other);

~Field();

void placeShip(Ship\* ship, int x, int y, ShipOrientation orientation);

void attackCell(int x, int y, int damage);

void printField() const noexcept;

private:

class FieldCell

{

public:

FieldCell();

~FieldCell() = default;

FieldCellStatus getStatus() const noexcept;

void setStatus(FieldCellStatus status) noexcept;

void attackCell(int damage);

bool isShip() const noexcept;

void setShipSegment(Ship\* ship, int index) noexcept;

ShipSegmentStatus getShipSegmentStatus() const noexcept;

private:

FieldCellStatus status\_;

Ship\* ship\_;

int ship\_segment\_index\_;

};

int size\_x\_;

int size\_y\_;

std::vector<std::vector<FieldCell>> field\_;

};

#endif

Название файла: field.cpp

#include "field.h"

#include "ship.h"

#include <iostream>

#include <vector>

#include <algorithm>

#include <stdexcept>

Field::Field() = default;

Field::Field(int size\_x, int size\_y)

{

if (size\_x <= 0 || size\_y <= 0) {

throw std::invalid\_argument("Field size must be grater than 0");

}

size\_x\_ = size\_x;

size\_y\_ = size\_y;

field\_.resize(size\_x\_);

for (int i = 0; i < size\_x\_; i++) {

for (int j = 0; j < size\_y\_; j++) {

field\_[i].push\_back(FieldCell());

}

}

}

Field::Field(const Field& other) :

Field(other.size\_x\_, other.size\_x\_)

{}

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

{

if (this == &other) {

return \*this;

}

size\_x\_ = other.size\_x\_;

size\_y\_ = other.size\_y\_;

field\_.resize(size\_x\_);

for (int i = 0; i < size\_x\_; i++) {

for (int j = 0; j < size\_y\_; j++) {

field\_[i].push\_back(FieldCell());

}

}

return \*this;

}

Field::Field(Field&& other) :

field\_(std::move(other.field\_)),

size\_x\_(std::move(other.size\_x\_)),

size\_y\_(std::move(other.size\_y\_))

{}

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

{

if (this == &other) {

return \*this;

}

field\_ = std::move(other.field\_);

size\_x\_ = std::move(other.size\_x\_);

size\_y\_ = std::move(other.size\_y\_);

return \*this;

}

Field::~Field()

{

for (int i = 0; i < size\_x\_; i++) {

field\_[i].clear();

}

field\_.clear();

}

void Field::placeShip(Ship\* ship, int x, int y, ShipOrientation orientation)

{

if (ship == nullptr) {

throw std::invalid\_argument("Ship pointer is nullptr");

}

int offset\_x, offset\_y;

if (orientation == ShipOrientation::horizontal) {

offset\_x = ship->getSize() - 1;

offset\_y = 0;

} else {

offset\_x = 0;

offset\_y = ship->getSize() - 1;

}

if (x < 0 || x >= size\_x\_ - offset\_x || y < 0 || y >= size\_y\_ - offset\_y) {

throw std::out\_of\_range("Ship coordinates out of range");

}

for (int i = x - 1; i < x + offset\_x + 1; i++) {

if (i >= 0 && i < size\_x\_) {

for (int j = y - 1; j < y + offset\_y + 1; j++) {

if (j >= 0 && j < size\_y\_) {

if (field\_[i][j].isShip()) {

throw std::logic\_error("Ships may not contact each other");

}

}

}

}

}

int segment\_index = 0;

for (int i = x; i < x + offset\_x + 1; i++) {

for (int j = y; j < y + offset\_y + 1; j++) {

field\_[i][j].setShipSegment(ship, segment\_index++);

}

}

}

void Field::attackCell(int x, int y, int damage)

{

if (x < 0 || x >= size\_x\_ || y < 0 || y >= size\_y\_) {

throw std::out\_of\_range("Coordinates out of range");

}

field\_[x][y].attackCell(damage);

if (field\_[x][y].isShip()) {

field\_[x][y].setStatus(FieldCellStatus::ship);

} else {

field\_[x][y].setStatus(FieldCellStatus::empty);

}

}

void Field::printField() const noexcept

{

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

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

if (field\_[x][y].getStatus() == FieldCellStatus::unknown) {

std::cout << "[ ]";

} else if (field\_[x][y].getStatus() == FieldCellStatus::empty) {

std::cout << "[.]";

} else {

if (field\_[x][y].getShipSegmentStatus() == ShipSegmentStatus::intact) {

std::cout << "[2]";

} else if (field\_[x][y].getShipSegmentStatus() == ShipSegmentStatus::damaged) {

std::cout << "[1]";

} else {

std::cout << "[0]";

}

}

}

std::cout << "\n";

}

std::cout << "\n";

}

Field::FieldCell::FieldCell()

{

status\_ = FieldCellStatus::unknown;

ship\_ = nullptr;

ship\_segment\_index\_ = -1;

}

FieldCellStatus Field::FieldCell::getStatus() const noexcept

{

return status\_;

}

void Field::FieldCell::setStatus(FieldCellStatus status) noexcept

{

status\_ = status;

}

void Field::FieldCell::attackCell(int damage)

{

if (ship\_ != nullptr) {

ship\_->damageSegment(ship\_segment\_index\_, damage);

}

}

bool Field::FieldCell::isShip() const noexcept

{

return ship\_ != nullptr;

}

void Field::FieldCell::setShipSegment(Ship\* ship, int index) noexcept

{

ship\_ = ship;

ship\_segment\_index\_ = index;

}

ShipSegmentStatus Field::FieldCell::getShipSegmentStatus() const noexcept

{

return ship\_->getSegmentStatus(ship\_segment\_index\_);

}

.

# Приложение Б Тестирование программы

Название файла: main.cpp

#include <iostream>

#include "ship.h"

#include "shipManager.h"

#include "field.h"

int main()

{

Ship test\_ship\_1(2); // OK

Ship test\_ship\_2(0); // Error

Ship test\_ship\_3(5); // Error

test\_ship\_1.damageSegment(0, 1); // OK

test\_ship\_1.damageSegment(0, -1); // Error

test\_ship\_1.damageSegment(3, 1); // Error

ShipManager manager({1, 3, 4});

manager.printShips();

Field field(5, 5);

manager.placeShip(field, 1, 0, 0, ShipOrientation::horizontal); // OK

manager.placeShip(field, 1, 0, 3, ShipOrientation::horizontal); // Error

manager.placeShip(field, 2, 0, 0, ShipOrientation::vertical); // Error

manager.placeShip(field, 2, 3, 0, ShipOrientation::vertical); // Error

manager.addShip(2);

manager.printShips();

field.attackCell(0, 0, 1); // OK

field.attackCell(6, 10, 1); // Error

field.attackCell(1, 0, 0);

field.attackCell(2, 0, 1);

field.attackCell(3, 0, 0);

field.attackCell(0, 1, 0);

field.attackCell(1, 1, 0);

field.attackCell(2, 1, 0);

field.attackCell(3, 1, 0);

manager.printShips();

field.printField();

}

Результат работы программы:

