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. 3383 _____ Боривец С. Ю.

Преподаватель _____ Жангиров Т. Р.

Санкт-Петербург

2024

Цель работы

Создать классы корабля, менеджера кораблей и игрового поля, реализовать методы для взаимодействия с ними. Продумать архитектуру проекта.

Задание

Лабораторная работа №1 - Создание классов

a. 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.

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

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

Каждая клетка игрового поля имеет три статуса:

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

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

Примечания:

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

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

Выполнение работы

Первое, что требуется реализовать в задании – корабль. Реализация расположена в файлах ship.cpp и ship.h. В приватных полях класса Ship у нас хранится размер, а также вектор из указателей на сегменты корабля: ShipPart*. В публичном доступе реализованы методы:

- Конструктор, который принимает длину и создает корабль;
- ship_status() – метод, который показывает состояние корабля в виде квадратов(белый – целый, желтый – поврежденный, красный – разрушенный);
- get_ship_parts() – метод, который возвращает вектор с сегментами корабля. В дальнейшем будет использоваться для выставления корабля на игровое поле;
- get_ship_length() – метод, который возвращает длину корабля;

Как было сказано ранее, каждый корабль состоит из сегментов, для которых был реализован отдельный класс ShipPart. Такие сегменты будет удобно расставлять на поле и взаимодействовать с ними, например, наносить урон, обращаясь к клетке, в которой сохранен сегмент, вызывая метод получения урона. Он имеет одно приватное поле – целостность сегмента. В публичном доступе реализованы методы:

- Конструктор, создающий целый сегмент;
- get_hp() – метод, который возвращает значение целостности сегмента;
- part_status() – метод, который показывает целостность сегмента в виде квадрата(белый – целый, желтый – поврежденный, красный – разрушенный);
- part_damage() – метод, который наносит урон сегменту, отнимает единицу от значения целостности сегмента;

Для хранения информации о кораблях нужен новый класс – класс менеджера кораблей. В приватном доступе у него несколько полей: количество кораблей, вектор с информацией о размерах кораблей, вектор с указателями на корабли(объектами класса Ship), вектор с координатами

кораблей, вектор с ориентациями кораблей. В приватном доступе также расположены два метода, использующиеся для записи данных в векторы координат и ориентации по индексу. Это сделано для того, чтобы лишь методы менеджера могли записывать/обновлять данные кораблей. В публичном доступе реализованы методы:

- Конструктор – принимает на вход количество кораблей, а также вектор с их размерами.
- Place_ships_on_field(Gamefield* gamefield) – метод, запрашивающий корректные данные для каждого корабля. Пока не будут введены корректные координаты и ориентация, позволяющие расположить корабль на поле, указатель на которое передается в аргументах.
- Print_ship_data(int ship_ind) – метод, который выведет длину, координаты, ориентацию корабля по индексу ship_ind в векторе кораблей менеджера.
- Print_ships_status() – метод, который для каждого корабля покажет квадраты с состоянием корабля, а так же выведет всю информацию.

Цели менеджера – хранить, собирать информацию о кораблях, обращаться к полю для выставления и удаления кораблей. Следующий реализованный класс – класс игрового поля. Это самостоятельный класс, который будет получать обращения от менеджера, например, если нужно будет поставить корабль, менеджер вызовет нужный метод у игрового поля, передаст ему корабль, координаты, а также ориентацию. В приватных полях класс Gamefield содержит двумерный массив указателей на объекты класса Cell – клетки, в который будут располагаться сегменты кораблей. Из методов публичного доступа в игровом поле есть – размещение корабля, как и было ранее сказано, менеджер передаст нужные данные, а поле займется выставлением каждого сегмента в нужные ячейки. Два метода для проверки координат, первый метод проверяет,

принадлежат ли координаты полю, а второй проверяет, можно ли в ячейку выставить сегмент корабля, не мешают ли ему другие сегменты. Метод remove_ship() отвечает за удаление корабля с поля, менеджер кораблей будет обращаться к игровому полю, чтобы убрать переданный корабль по переданным координатам и ориентации. Метод field_take_hit() принимает координаты места, в который должен будет нанестись урон, поле вызовет нужный метод у клетки, клетка, если в ней есть сегмент, вызовет метод получения урона у сегмента, сегмент потеряет единицу целостности. У поля существуют операторы и конструкторы копирования и перемещения. Также существует метод вывода игрового поля, клетки в которых ничего нет подсвечиваются синим, клетки, которым был нанесен урон - фиолетовым

Последний реализованный класс – класс клетки игрового поля. У клетки два приватных поля – статус попадания в нее и указатель на лежащий в ней сегмент корабля. Указатель нужен для того, чтобы в случае указания выстрела от игрока, поле вызывало метод получения урона у нужной по координатам клетки. Метод получения урона у клетки – cell_hit() – вызывает метод получения урона у сегмента, либо меняет статус попадания по клетке. В публичном доступе существует методы размещения, удаления сегмента корабля, проверки, размещен ли сегмент на клетке, а также вывод самой клетки.

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
1.		Выведено поле, с расставленными кораблями и нанесенными по ним ударами. Менеджер кораблей вывел состояние и данные о существующих кораблях	OK

Выводы

Созданы классы корабля, менеджера кораблей и игрового поля, реализованы методы для взаимодействия с ними. Архитектура проекта продумана.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.c

```
class CustomStack {
public:
    CustomStack()
    {
        mHead = new ListNode;
        mHead->mNext = nullptr;
        st_size = 0;
    }

    void push(int val)
    {
        ListNode *newElem = new ListNode;
        newElem->mData = val;
        newElem->mNext = mHead;
        mHead = newElem;
        st_size++;
    }
    void pop()
    {
        if (!(*this).empty()) {
            ListNode *newHead = mHead->mNext;
            delete mHead;
            mHead = newHead;
            st_size--;
        }
    }
    int top()
    {
        return mHead->mData;
    }

    size_t size()
    {
        return st_size;
    }

    bool empty()
    {
        return st_size == 0;
    }

    ~CustomStack(){
        delete mHead;
    }
private:
    size_t st_size;

protected:
    ListNode* mHead;
};

int main() {
```

```
    CustomStack stack = CustomStack();
    string cmd;
    while (cin >> cmd) {
        if (cmd == "cmd_push") {
            int val;
            cin >> val;
            stack.push(val);
            cout << "ok" << endl;

        } else if (cmd == "cmd_pop") {
            if(stack.empty()){
                cout << "error";
                break;
            } else {
                cout << stack.top() << endl;
                stack.pop();
            }
        } else if (cmd == "cmd_top") {
            if(stack.empty()){
                cout << "error";
                break;
            } else {
                cout << stack.top() << endl;
            }
        } else if (cmd == "cmd_size") {
            cout << stack.size() << endl;
        } else if (cmd == "cmd_exit") {
            cout << "bye";
            break;
        }
    }
    return 0;
}
```

ПРИЛОЖЕНИЕ Б

UML-ДИАГРАММА КЛАССОВ