**МИНОБРНАУКИ РОССИИ**

**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**

**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)  
Кафедра информационных систем**

**ОТЧЕТ**

по практической работе №1

## ТЕМА: Введение в ООП: создание классов, конструкторов классов, методов классов; наследование.

Студент гр. 0324 Косенко А.Р.

Преподаватель Глущенко А.Г.

Санкт-Петербург

2021

**Цель работы.**

Разработать и реализовать набор классов:

●     Класс игрового поля

●     Набор классов юнитов

## Методические сведенья. Описание класса

Класс является абстрактным типом данных, определяемым пользователем, и представляет собой модель реального объекта в виде данных и функций для работы с ними.

Данные класса называются полями (по аналогии с полями структуры), а функции класса — методами. Поля и методы называются элементами класса.

Класс может определять переменные и константы для хранения состояния объекта и функции для определения поведения объекта.

Например, определим простейший класс:

class Person {

};

Для определения класса применяется ключевое слово class, после которого идет собственно название класса. В данном случае класс называется Person и представляет человека. После названия класса идет блок кода, который определяет тело класса.

После определения класса мы можем создавать его переменные:

class Person {

};

int main() {

Person person;

return 0;

}

Описание класса в первом приближении выглядит так:

class <имя> {

[ private: ]

<описание скрытых элементов>

public:

<описание доступных элементов>

}; / / Описание заканчивается точкой с запятой

Спецификаторы доступа private и public управляют видимостью элементов класса. Элементы, описанные после служебного слова private, видимы только внутри класса. Этот вид доступа принят в классе по умолчанию. Интерфейс класса описывается после спецификатора public. Действие любого спецификатора распространяется до следующего спецификатора или до конца класса. Можно задавать несколько секций private и public, порядок их следования значения не имеет.  
  
**Поля класса:**

* могут иметь любой тип, кроме типа этого же класса (но могут быть указателями или ссылками на этот класс);
* могут быть описаны с модификатором const, при этом они инициализируются только один раз (с помощью конструктора) и не могут изменяться;
* могут быть описаны с модификатором static, но не как auto, extern и register.

Классы могут быть глобальными (объявленными вне любого блока) и локальными (объявленными внутри блока, например, функции или другого класса).

Ниже перечислены некоторые особенности локального класса:

* внутри локального класса можно использовать типы, статические (static) и внешние (extern) переменные, внешние функции и элементы перечислений из области, в которой он описан; запрещается использовать автоматические переменные из этой области;
* локальный класс не может иметь статических элементов;
* методы этого класса могут быть описаны только внутри класса;
* если один класс вложен в другой класс, они не имеют каких-либо особых прав доступа к элементам друг друга и могут обращаться к ним только по общим правилам.

В качестве примера создадим класс, моделирующий персонаж компьютерной игры. Для этого требуется задать его свойства (например, количество щупалец, силу или наличие гранатомета) и поведение. Естественно, пример будет схематичен, поскольку приводится лишь для демонстрации синтаксиса.

class monstr {

int health, ammo:

public:

monstr(int he = 100. int am = 10){

health = he;

ammo = am;

}

void draw(int x. int y, int scale, int position);

int getHealth() { return health; }

int getAmmo(){ return ammo; }

};

Все методы класса имеют непосредственный доступ к его скрытым полям, иными словами, тела функций класса входят в область видимости private элементов класса.

void monstr::draw(int х. int у. int scale, int position) {

/\* тело метода \*/

}

Метод можно определить как встроенный и вне класса с помощью директивы inline (как и для обычных функций, она носит рекомендательный характер):

inline int monstr::get\_ammo(){

return ammo;

}

В каждом классе есть хотя бы один метод, имя которого совпадает с именем класса. Он называется конструктором и вызывается автоматически при создании объекта класса. Конструктор предназначен для инициализации объекта. Автоматический вызов конструктора позволяет избежать ошибок, связанных с использованием неинициализированных переменных. Мы подробно рассмотрим конструкторы в разделе «Конструкторы».

Типы данных struct и union являются видами класса, с ними вы уже сталкивались в прошлых курсах

**Описание объектов**

Конкретные переменные типа «класс» называются экземплярами класса, или объектами. Время жизни и видимость объектов зависит от вида и места их описания и подчиняется общим правилам С++:

monstr Vasia; // Объект класса monstr с параметрами по умолчанию

monstr Super(200, 300); // Объект с явной инициализацией

monstr stado[100]; // Массив объектов с параметрами по умолчанию

monstr\* beavis = new monstr (10); // Динамический объект

//(второй параметр задается по умолчанию)

monstr& butthead = Vasia; // Ссылка на объект

При создании каждого объекта выделяется память, достаточная для хранения всех eib полей, и автоматически вызывается конструктор, выполняющий их инициализацию. Методы класса не тиражируются. При выходе объекта из области действия он уничтожается, при этом автоматически вызывается деструктор.

Доступ к элементам объекта аналогичен доступу к полям структуры. Для этого используются операция . (точка) при обращении к элементу через имя объекта и операция -> при обращении через указатель, например:

int ammo = Vasid.getAmmo();

stado[5].draw;

cout << beavis -> getHealth();

Обратиться таким образом можно только к элементам со спецификаторам public. Получить или изменить значения элементов со спецификатором private можно только через обращение к соответствующим методам. Можно создать константный объект, значения полей которого изменять запрещается. К нему должны применяться только константные методы:

class monstr{

int getHealth() const {return health;}

};

const monstr Dead(0,0); // Константный объект

cout << Dead.getHealth();

Константный метод:

* объявляется с ключевым словом const после списка параметров; а не может изменять значения полей класса;
* может вызывать только константные методы;
* может вызываться для любых (не только константных) объектов. Рекомендуется описывать как константные те методы, которые предназначены для получения значений полей.

**Конструкторы**

Конструктор предназначен для инициализации объекта и вызывается автоматически при его создании. Ниже перечислены основные свойства конструкторов:

* Конструктор не возвращает значение, даже типа void. Нельзя получить указатель на конструктор.
* Класс может иметь несколько конструкторов с разными параметрами для разных видов инициализации (при этом используется механизм перегрузки).
* Конструктор, вызываемый без параметров, называется конструктором по умолчанию.
* Параметры конструктора могут иметь любой тип, кроме этого же класса. Можно задавать значения параметров по умолчанию. Их может содержать только один из конструкторов.
* Если программист не указал ни одного конструктора, компилятор создает его автоматически. Такой конструктор вызывает конструкторы по умолчанию для полей класса и конструкторы по умолчанию базовых классов. В случае, когда класс содержит константы или ссылки, при попытке создания объекта класса будет выдана ошибка, поскольку их необходимо инициализировать конкретными значениями, а конструктор по умолчанию этого делать не умеет.
* Конструкторы не наследуются.
* Конструкторы нельзя описывать с модификаторами const, virtual и static.
* Конструкторы глобальных объектов вызываются до вызова функции main. Локальные объекты создаются, как только становится активной область их действия. Конструктор запускается и при создании временного объекта (например, при передаче объекта из функции).
* Конструктор вызывается, если в программе встретилась какая-либо из синтаксических конструкций:

ИмяКласса имяОбъекта [(список параметров)];

// Список параметров не должен быть пустым

ИмяКласса (список параметров);

// Создается объект без имени (список может быть пустым)

ИмяКласса имяОбъекта = выражение;

// Создается объект без имени и копируется

Примеры:

monstr Super(200, 300), Vas1a(50), Z;

monstr X = monstr(1OOO);

monstr Y = 500;

В первом операторе создаются три объекта. Значения не указанных параметров устанавливаются по умолчанию.

Во втором операторе создается безымянный объект со значением параметра health = 1000 (значение второго параметра устанавливается по умолчанию). Выделяется память под объект X, в которую копируется безымянный объект.

В последнем операторе создается безымянный объект со значением параметра health = 500 (значение второго параметра устанавливается по умолчанию). Выделяется память под объект Y, в которую копируется безымянный объект. Такая форма создания объекта возможна в том случае, если для инициализации объекта допускается задать один параметр.

В качестве примера класса с несколькими конструкторами усовершенствуем описанный ранее класс monstr, добавив в него поля, задающие цвет (skin) и имя (name):

enum color {red, green, blue}; // Возможные значения цвета

class Monstr {

int health, ammo;

color skin;

char \*name;

public:

Monstr(int he = 100. int am =10);

Monstr(color sk);

Monstr(char \* nam);

int get\_health(){return health;}

int get\_ammo(){return ammo;}

...

};

//--------

Monstr::Monstr(int he, int am){

health = he;

ammo = am;

skin = red;

name = 0;

}

//--------

Monstr::Monstr(color sk)(

switch (sk){

case red : health = 100; ammo = 10; skin = red; name = 0; break;

case green : health = 100; ammo = 20; skin = green; name = 0; break;

case blue : health = 100; ammo = 40; skin = blue; name = 0; break;

}

}

//--------

Monstr::Monstr(char\* nam){

name = new char [strlen(nam) + 1];

// К длине строки добавляется 1 для хранения нуль-символа

strcpy(name, nam):

health = 100;

ammo = 10;

sk1n = red;

}

//--------

Monstr\* m = new monstr ("Ork"):

Monstr Green (green):

Первый из приведенных выше конструкторов является конструктором по умолчанию, поскольку его можно вызвать без параметров. Объекты класса monstr теперь можно инициализировать различными способами, требуемый конструктор будет вызван в зависимости от списка значений в скобках. При задании нескольких конструкторов следует соблюдать те же правила, что и при написании перегруженных функций — у компилятора должна быть возможность распознать нужный вариант.

## Простое наследование

Простым называется наследование, при котором производный класс имеет одного родителя. Для различных методов класса существуют разные правила наследования — например, конструкторы и операция присваивания в производном классе не наследуются, а деструкторы наследуются. Рассмотрим наследование классов и проблемы, возникающие при этом, на примере.

Создадим производный от класса monstr класс daemon, добавив полезную в некоторых случаях способность думать. Базовый класс приведен с некоторыми методами, добавленными в предыдущих разделах:

enum Color {red, green, blue};

/ / Класс Monstr

class Monstr {

/ / -- Скрытые поля класса:

int health, ammo;

Color skin;

char \*name;

public:

/ / Конструкторы:

Monstr(int he = 100, int am = 10);

Monstr(color sk);

Monstr(char \* nam);

Monstr(Monstr &M);

/ / Деструктор:

~monstr() {delete [ ] name;}

/ / Операции:

Monstr& operator ++() {

++health;

return \*this;

}

Monstr operator ++(int) {

Monstr M(\*this);

health++;

return M;

}

operator int() {

return health;

}

bool operator >(Monstr &M) {

return health > M.health ? true : false;

}

const Monstr& operator = (Monstr &M){

if (&M == this) return this;

if (name) delete [] name;

if (M.name) {

name = new char [strlen(M.name) + 1];

strcpy(name, M.name);

}

else name = 0;

health = M.health;

ammo = M.ammo;

skin = M.skin;

return \*th1s;

}

// Методы доступа к полям:

int getHealth() const {return health; }

int getAmmo() const {return ammo; }

// Методы, изменяющие значения полей:

void changeHealth(int he) { health << he; }

// Прочие методы:

void draw(int x, int y, int scale, int position);

};

// Реализация класса monstr

Monstr::Monstr(int he, int am) : health (he), ammo (am), skin (red), name (0) { }

Monstr::Monstr(Monstr &M){

if (M.name) {

name = new char [strlen(M.name) + 1];

strcpy(name, M.name);

}

else name = 0;

health - M.health;

ammo - M.ammo;

skin = M.skin;

}

Monstr::Monstr(Color sk) {

switch (sk) {

case red : health = 100; ammo = 10; skin = red; name = 0; break;

case green : health = 100; ammo = 20; skin = green; name = 0; break;

case blue : health = 100; ammo = 40; skin = blue; name = 0; break;

}

}

Monstr::Monstr(char \* nam) {

name = new char [strlen(nam) + 1];

strcpy(name, nam);

health = 100; ammo = 10; skin << red;

}

void Monstr::draw(int x, int y, int scale, int position)

{ /\* ... Отрисовка monstr \*/ }

// Класс Daemon

class Daemon : public Monstr {

int brain;

public:

// Конструкторы:

Daemon(int br << 10) { brain = br; }

Daemon(Color sk) : Monstr (sk) { brain = 10; }

Daemon(char \* nam) : Monstr (nam) { brain = 10; }

Daemon(Daemon &M) : Monstr (M) { brain = M.brain; }

// Операции:

const Daemon& operator = (Daemon &M) {

if (&M -= this) return this;

brain = M.brain;

Monstr::operator << (M);

return \*this;

}

// Методы, изменяющие значения полей:

void think();

// Прочие методы:

void draw(int х, int у, int scale, int position);

};

// Реализация класса Daemon

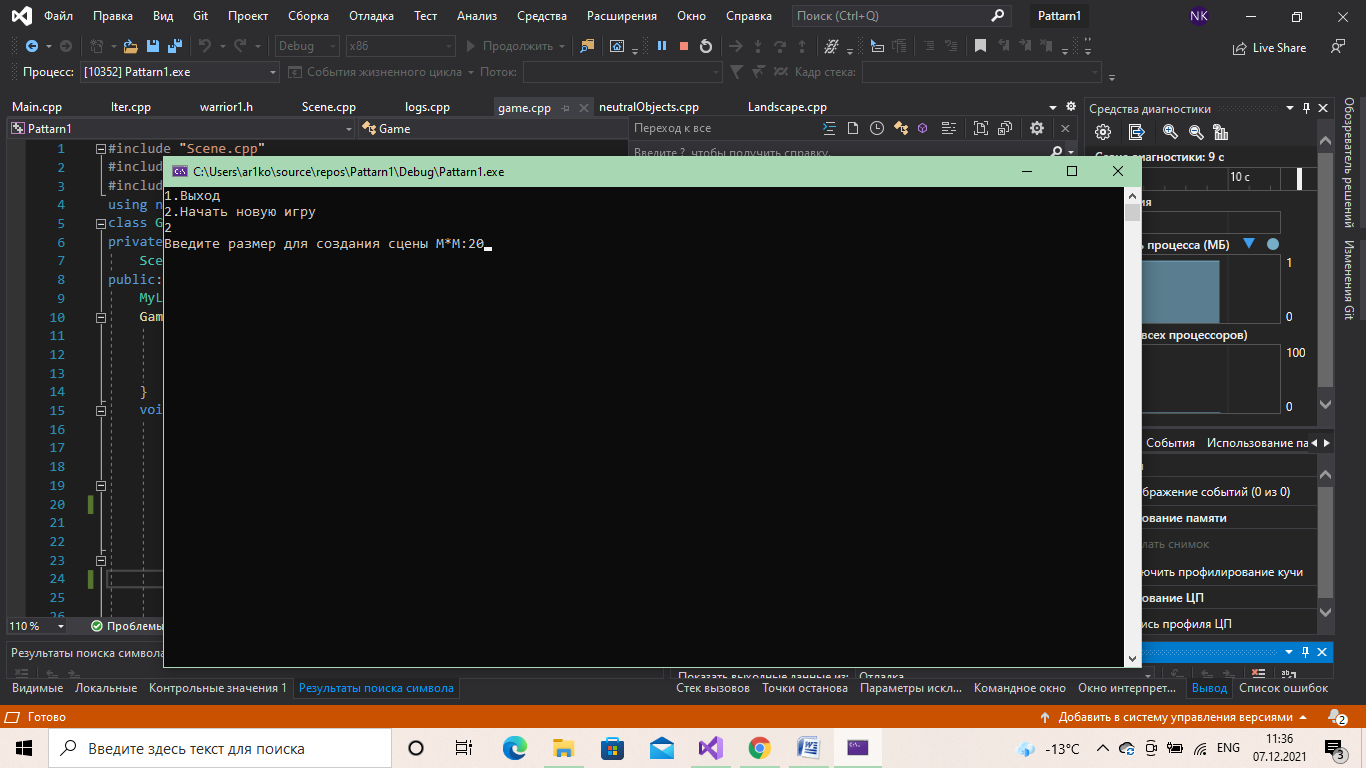
void Daemon::think() { /\* ... \*/ }

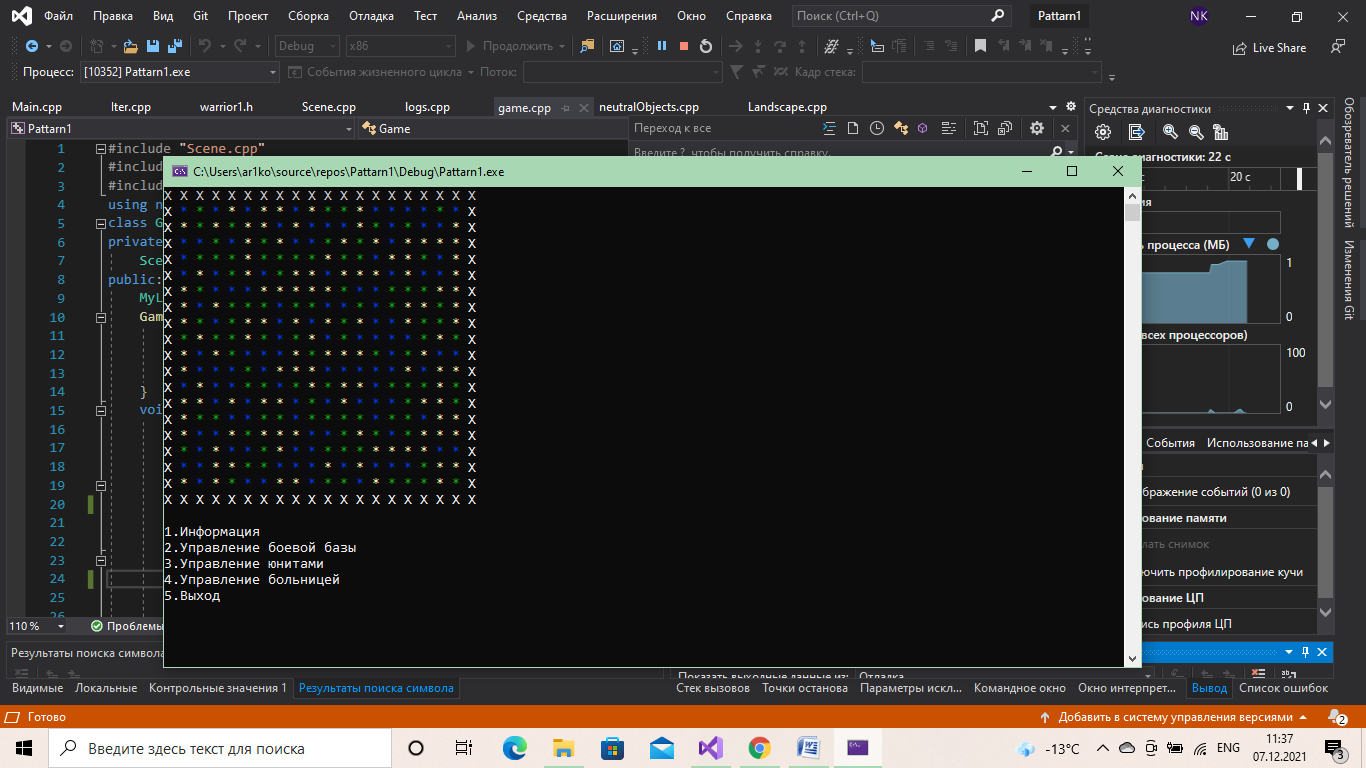
void Daemon::draw(int x, int y, int scale, int position)

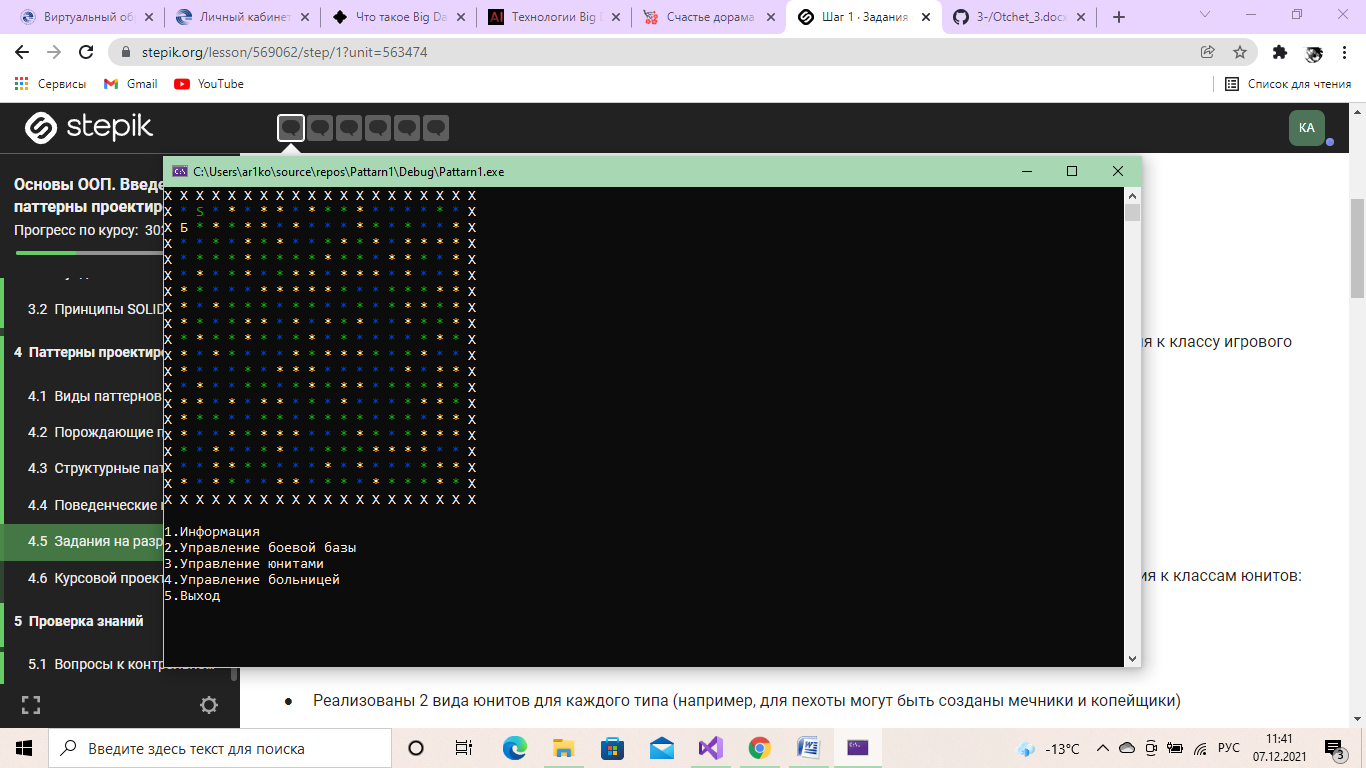
{ /\* ... Отрисовка Daemon \*/ }

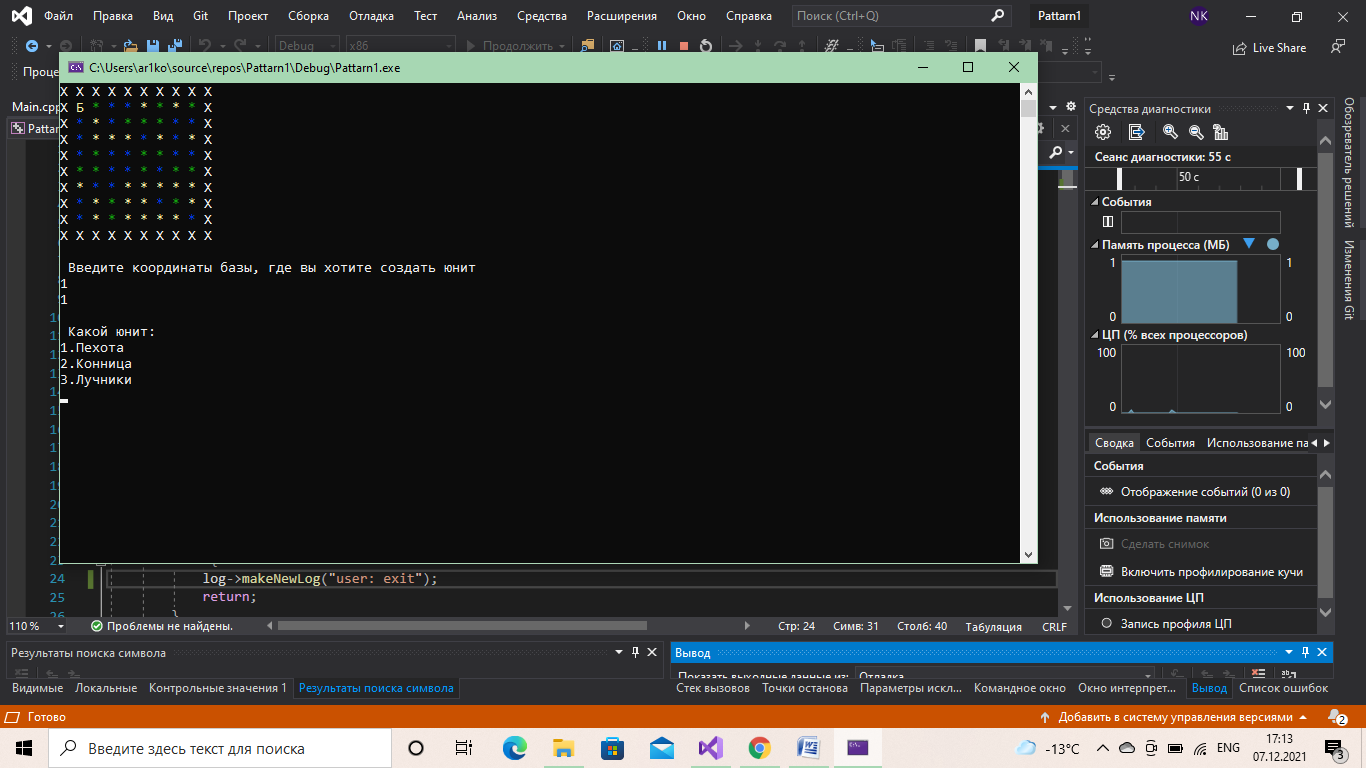
В классе Daemon введено поле brain и метод think, определены собственные конструкторы и операция присваивания, а также переопределен метод отрисовки draw. Все поля класса Monstr, операции (кроме присваивания) и методы getHealth, getAmmo и changeHealth наследуются в классе daemon, а деструктор формируется по умолчанию.

***ПРИМЕР РАБОТЫ ПРОГРАММЫ.***

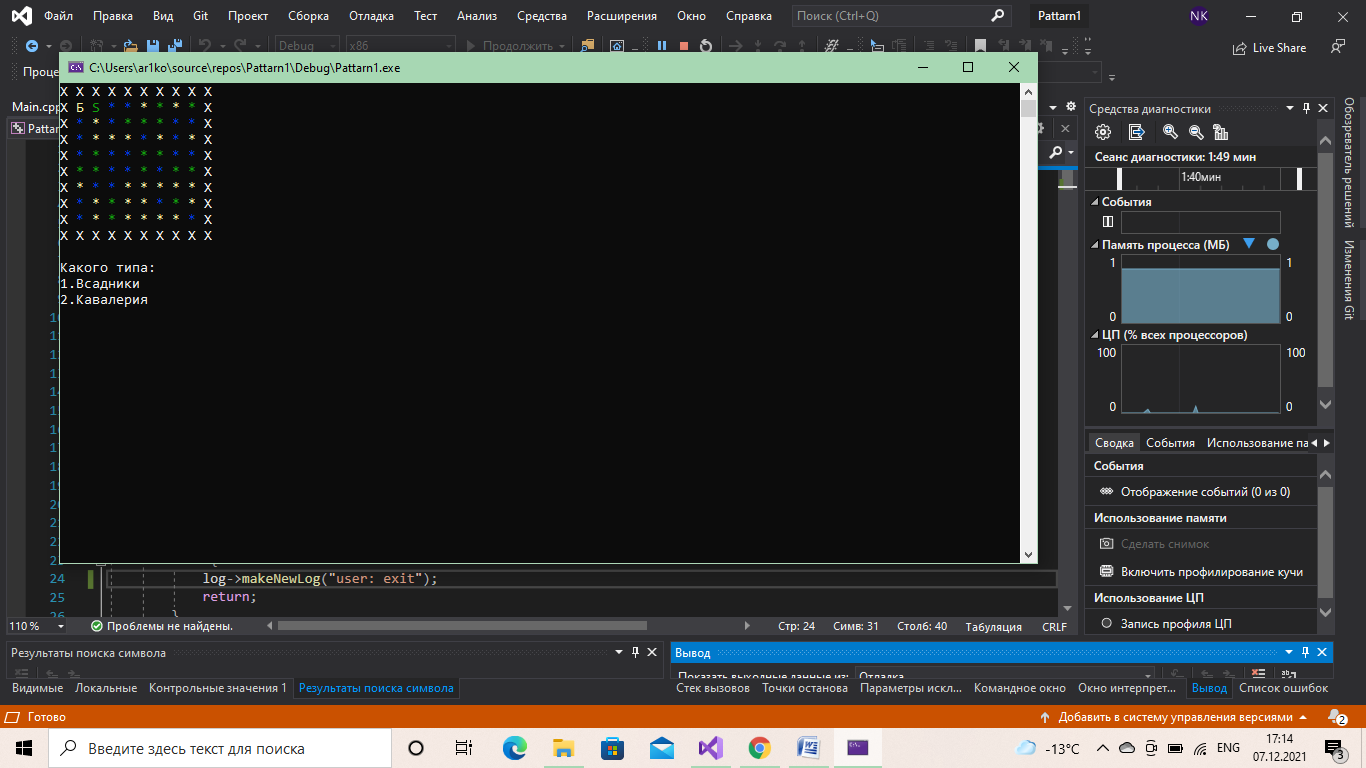
1.Создание игрового поля  


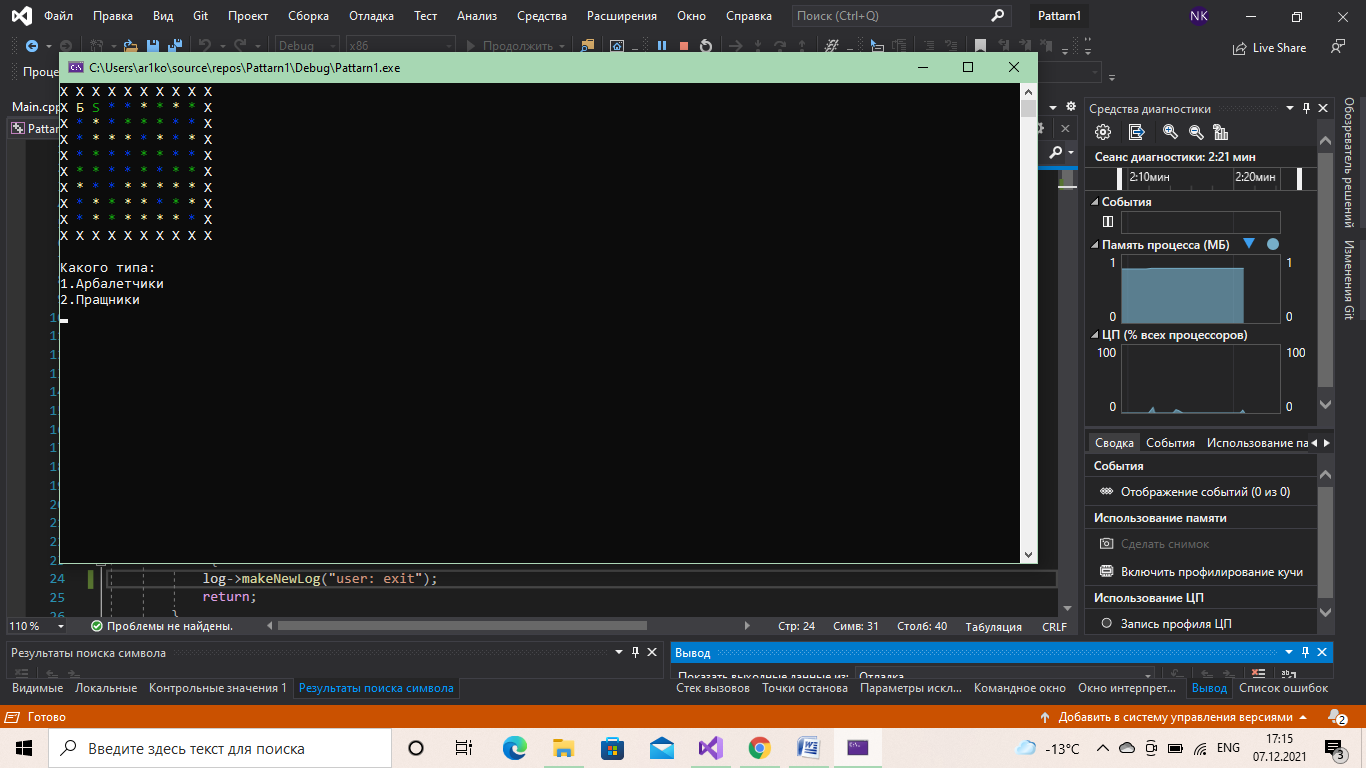


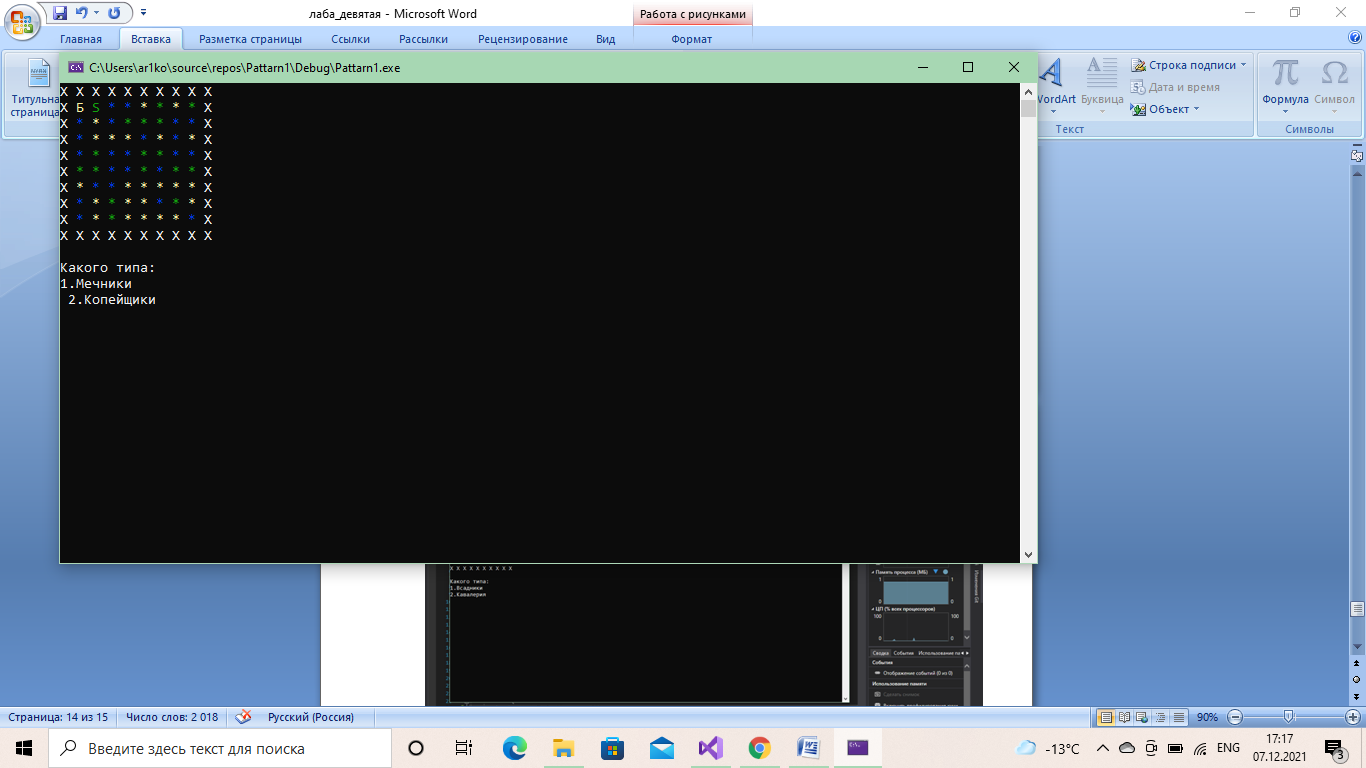
2.Добавление юнитов   
****

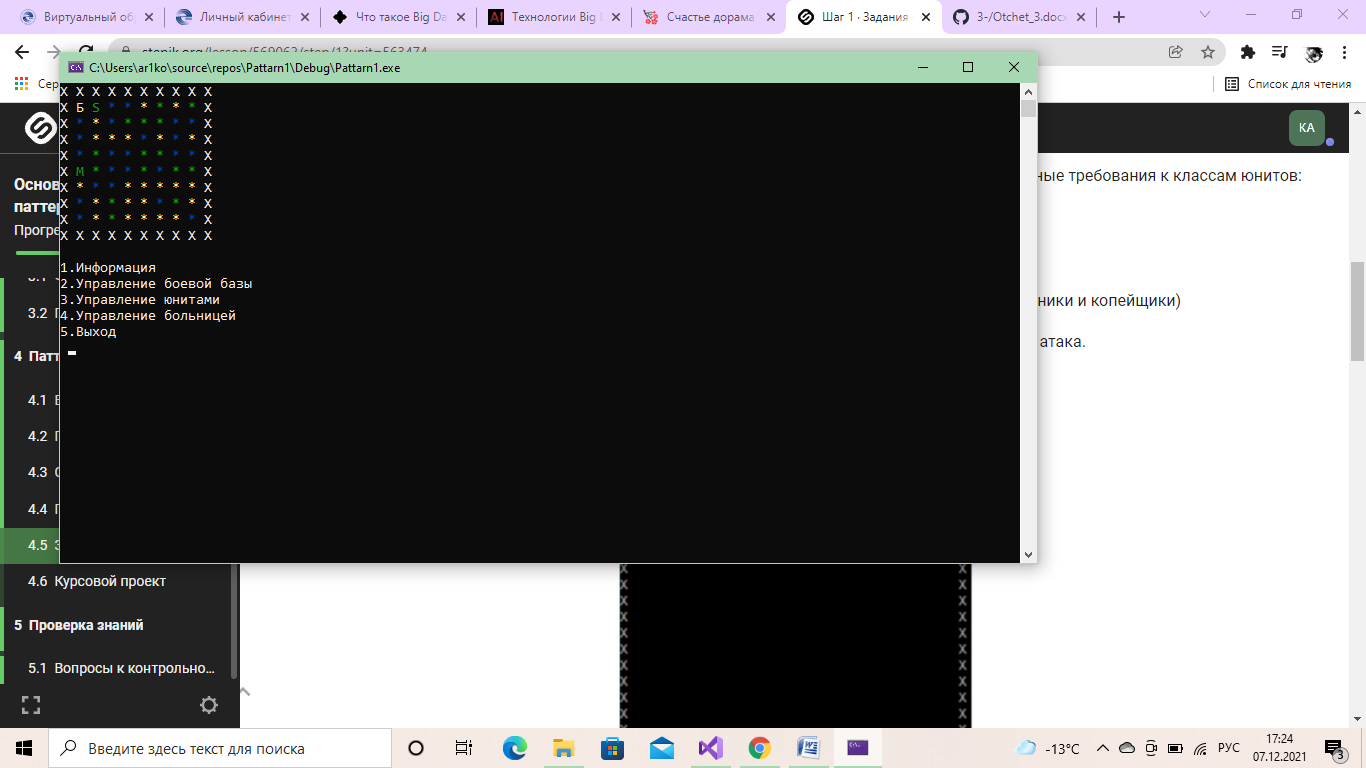
Удаление происходит, если убить юнит.  
3.У меня есть 3 типа юнитов:  
****

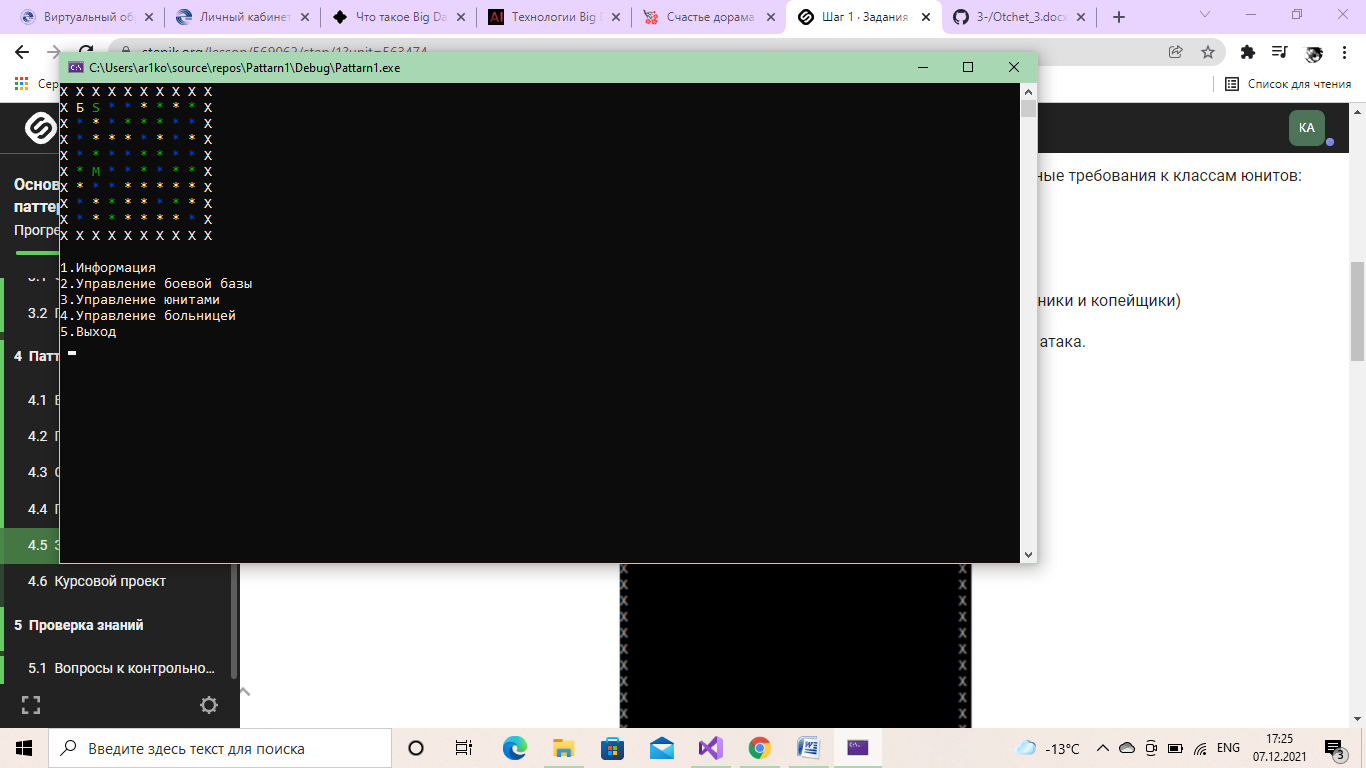
4.У каждого юнита есть разделение на 2 типа

****

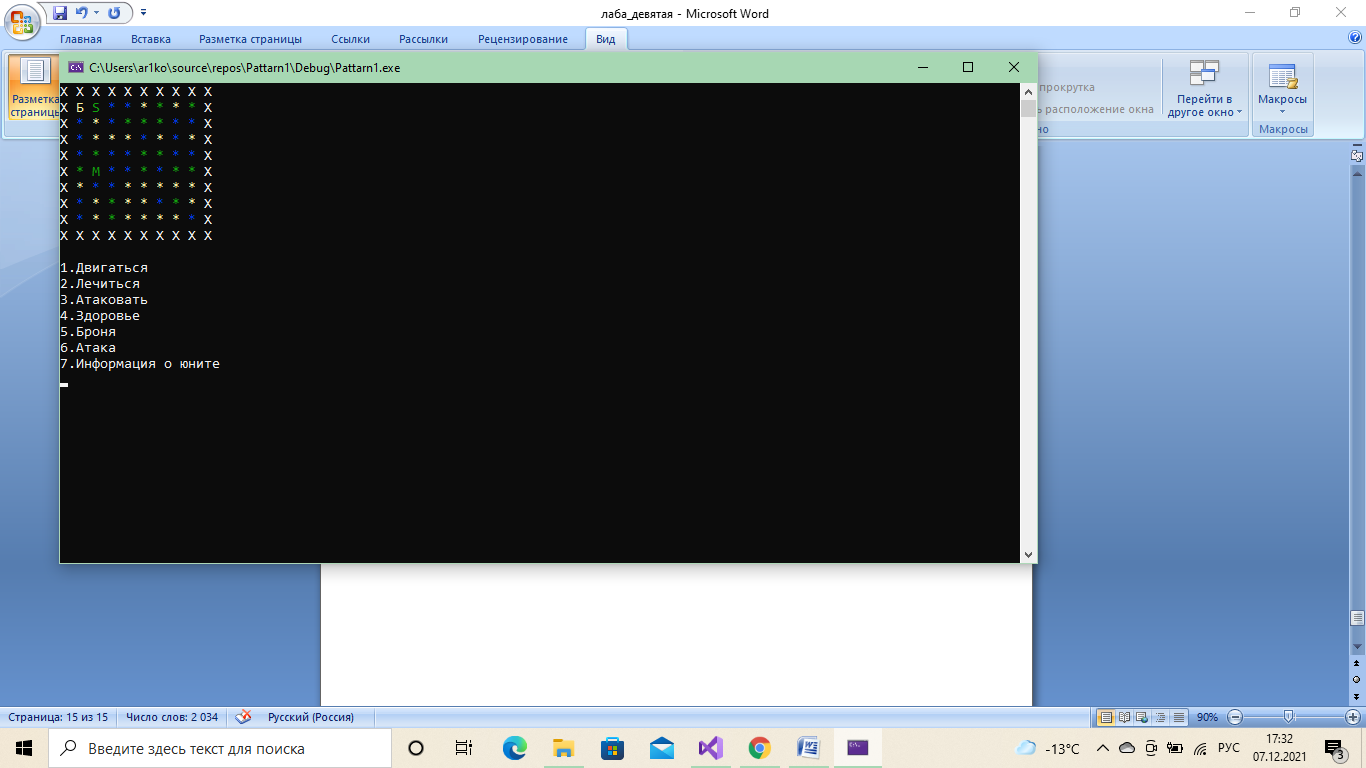
****

****

5.Юнит умеет перемещаться  




6.Юнит имеет характеристики: броня, здоровье, атака.

  
Мы можем узнать характеристики юнитов, указав их координаты, вот к примеру здоровье юнита в координате (1,2)

