**МИНОБРНАУКИ РОССИИ**

**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**

**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

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

**ОТЧЕТ**

по практической работе №3

## ТЕМА: Логическое разделение классов

Студент гр. 0324 Косенко А.Р.

Преподаватель Глущенко А.Г.

Санкт-Петербург

2021

**Цель работы.**

Разработать и реализовать набора классов для взаимодействия пользователя с юнитами и базой. Основные требования:

●     Должен быть реализован функционал управления юнитами

●     Должен быть реализован функционал управления базой

## 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.

**Перегрузка операций**

C++ позволяет переопределить действие большинства операций так, чтобы при использовании с объектами конкретного класса они выполняли заданные функции. Эта дает возможность использовать собственные типы данных точно так же, как стандартные. Обозначения собственных операций вводить нельзя. Можно перегружать любые операции, существующие в C++, за исключением:

. , \* ?: # ## sizeof

Перегрузка операций осуществляется с помощью методов специального вида {функций-операций) и подчиняется следующим правилам:

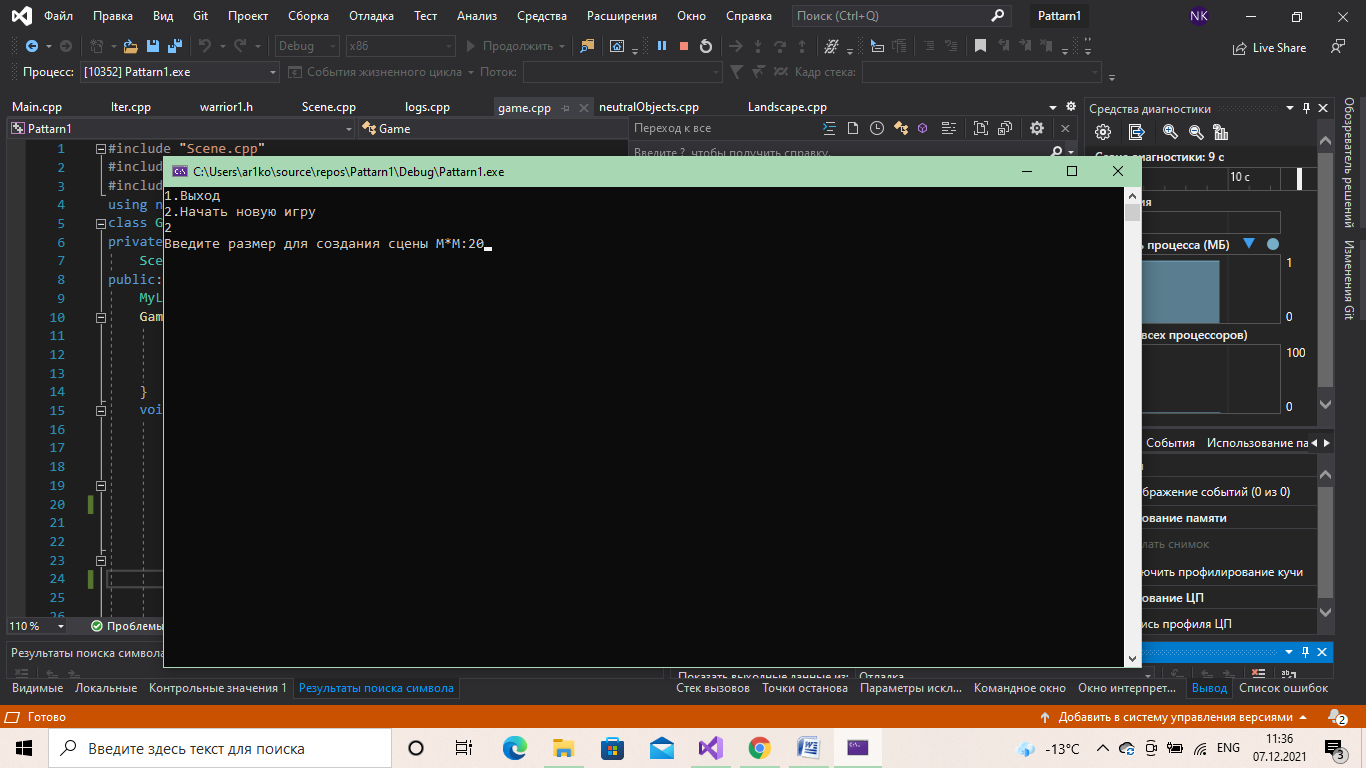
* при перегрузке операций сохраняются количество аргументов, приоритеты операций и правила ассоциации (справа налево или слева направо), используемые в стандартных типах данных;
* для стандартных типов данных переопределять операции нельзя;
* функции-операции не могут иметь аргументов по умолчанию;
* функции-операции наследуются (за исключением =);
* функции-операции не могут определяться как static.

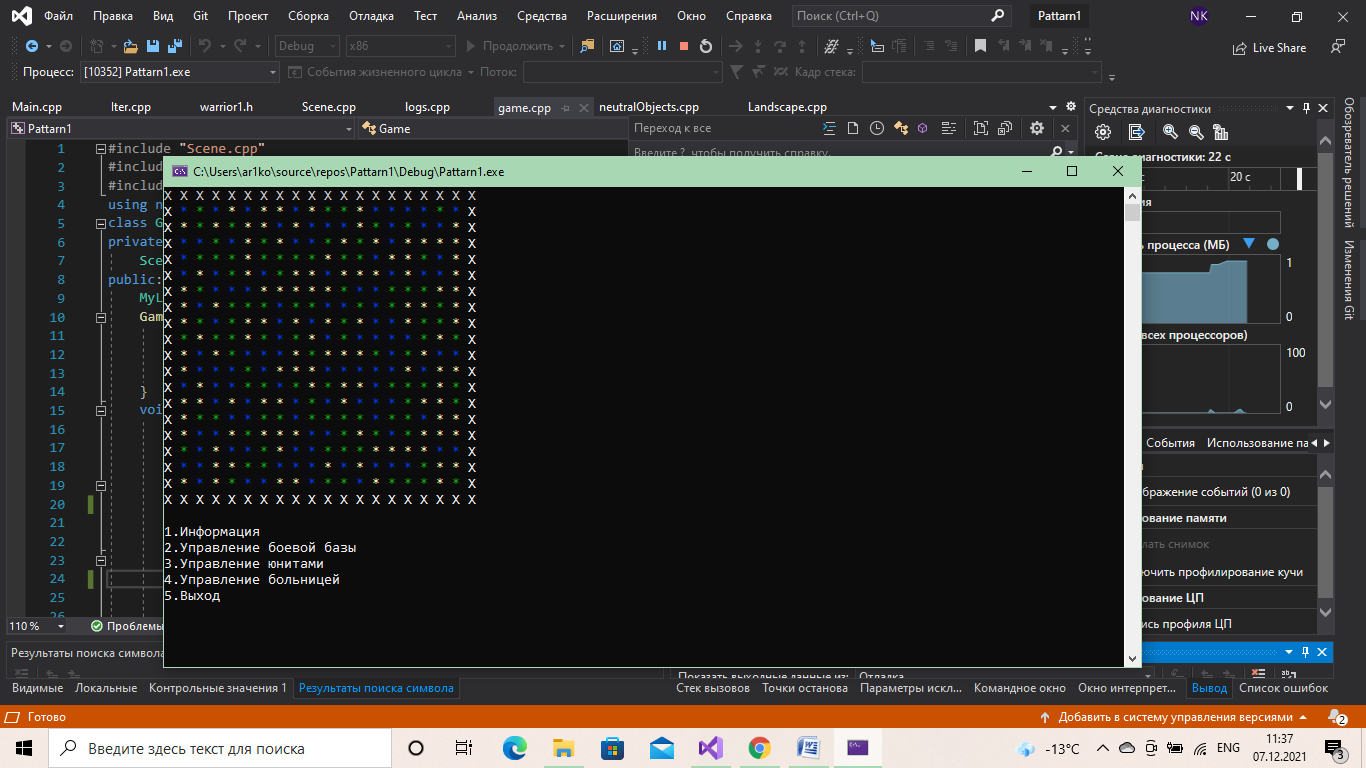
Функцию-операцию можно определить тремя способами: она должна быть либо методом класса, либо дружественной функцией класса, либо обычной функцией. В двух последних случаях функция должна принимать хотя бы один аргумент, имеющий тип класса, указателя или ссылки на класс.

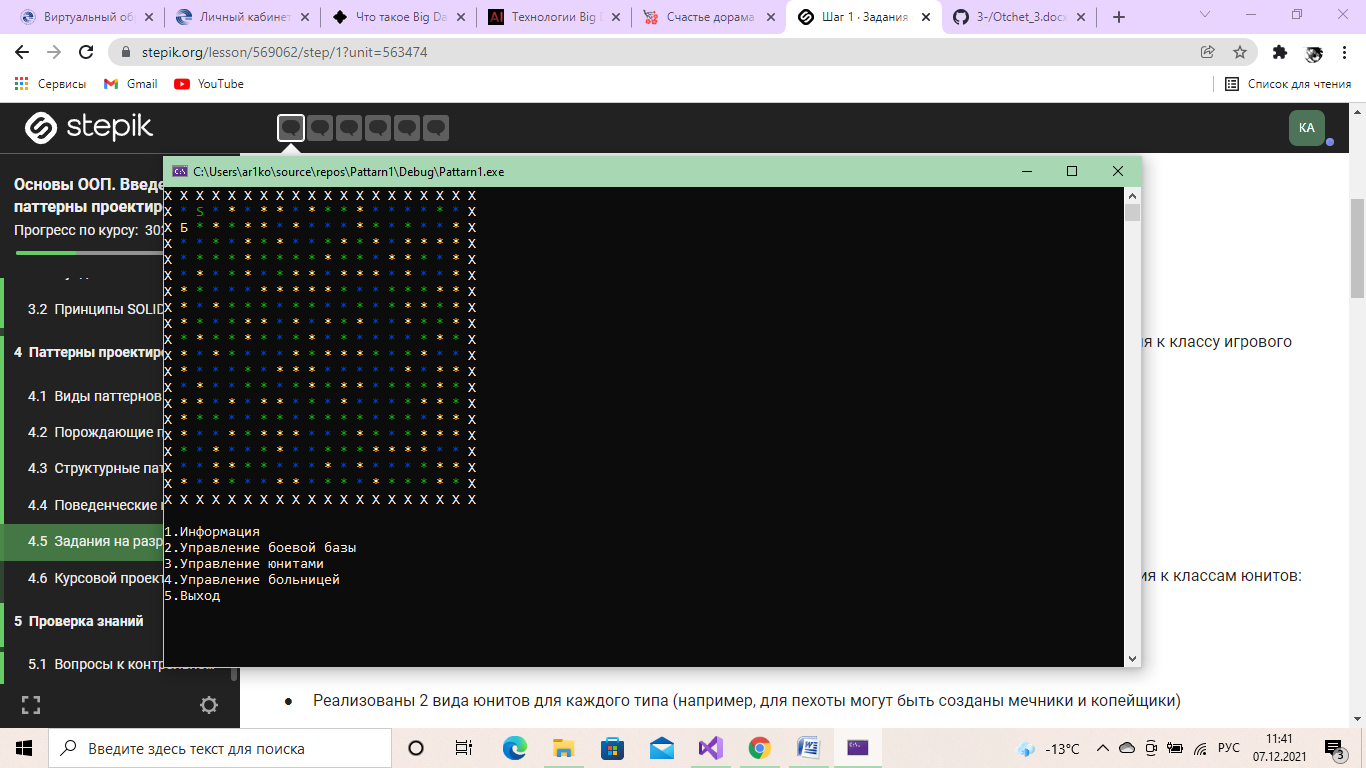
Функция-операция содержит ключевое слово operator, за которым следует знак переопределяемой операции:

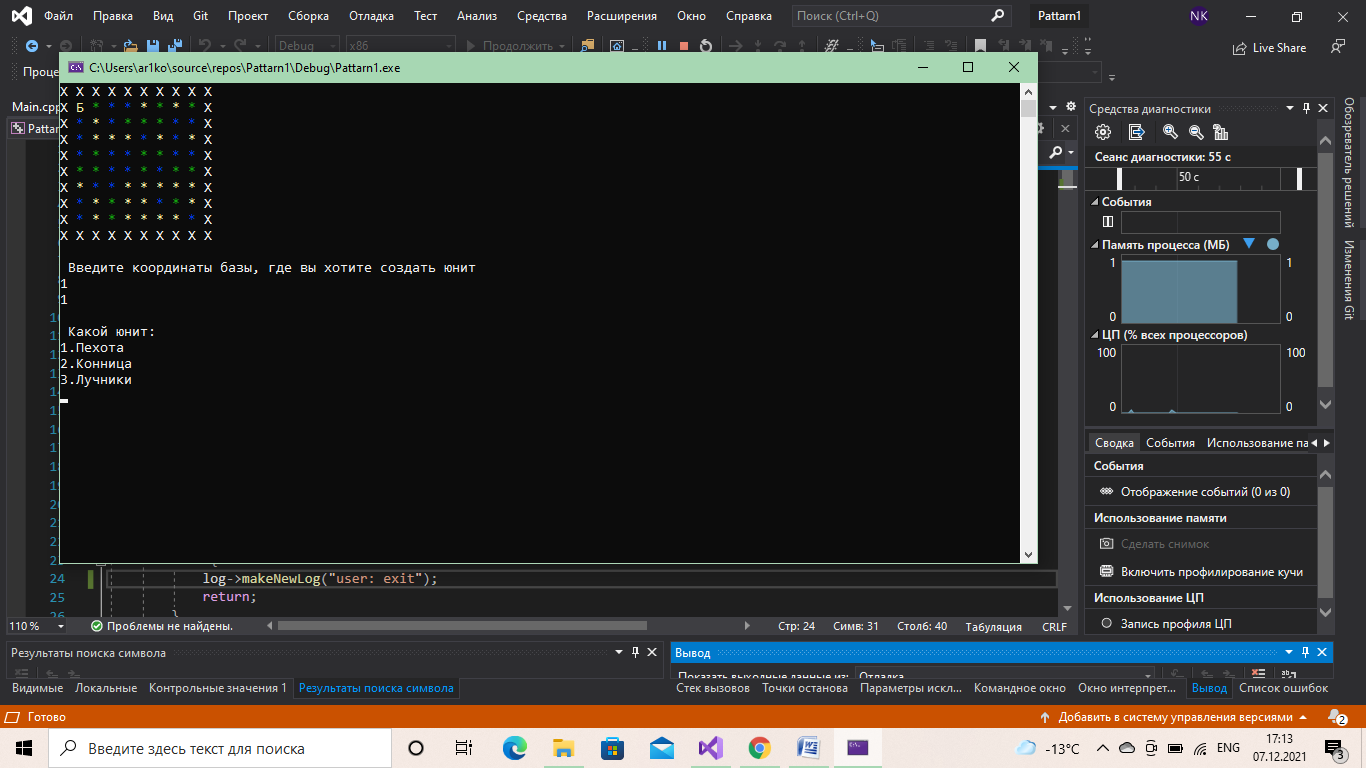
тип operator операция ( список параметров) { тело функции }

***ПРИМЕР РАБОТЫ ПРОГРАММЫ.***

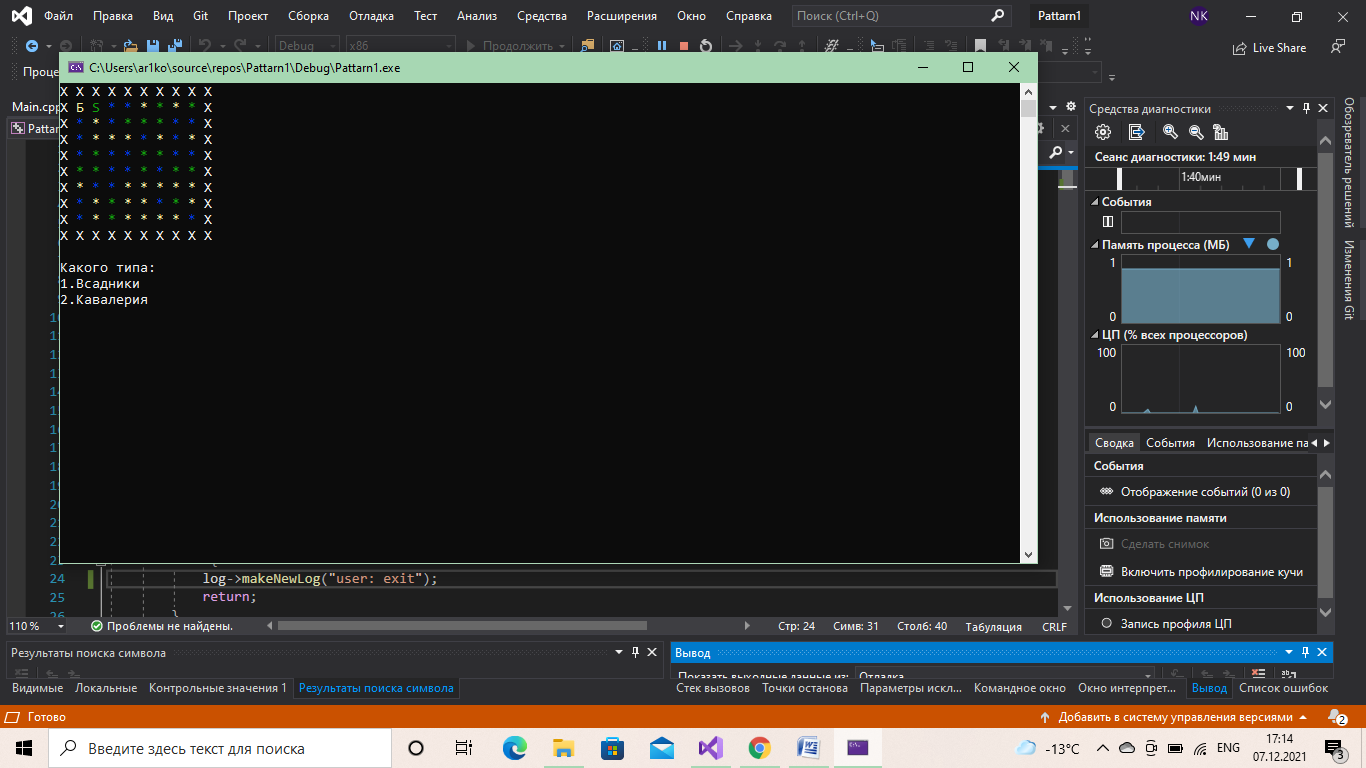
1.Создание игрового поля  


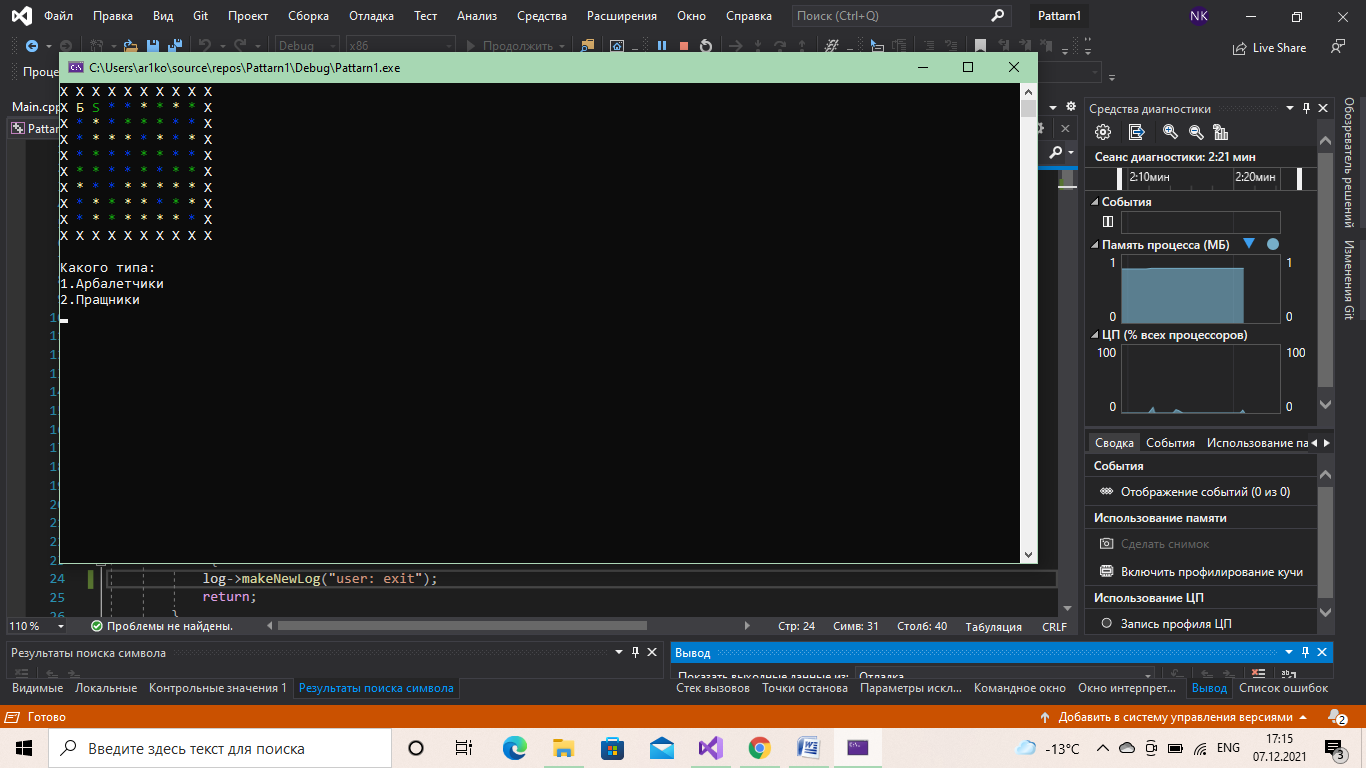


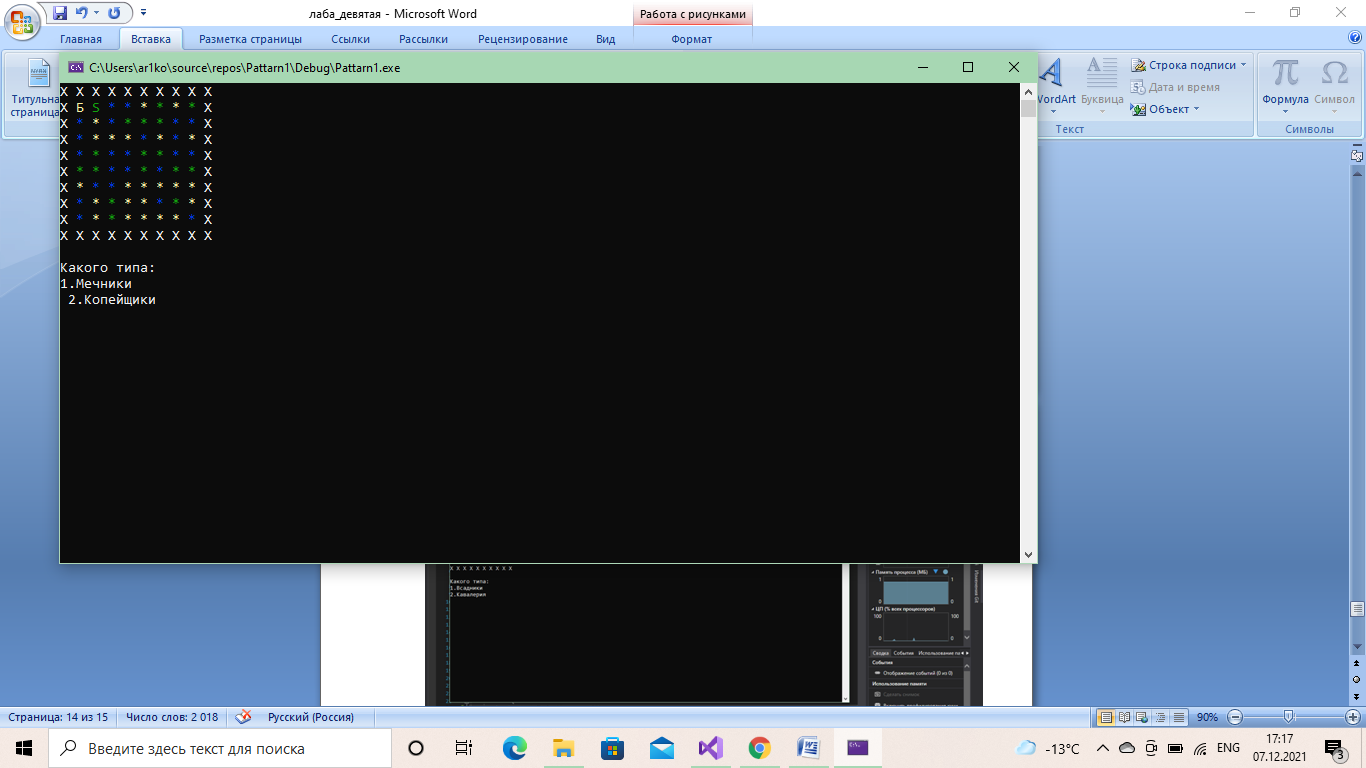
2.Добавление юнитов и базы  
****

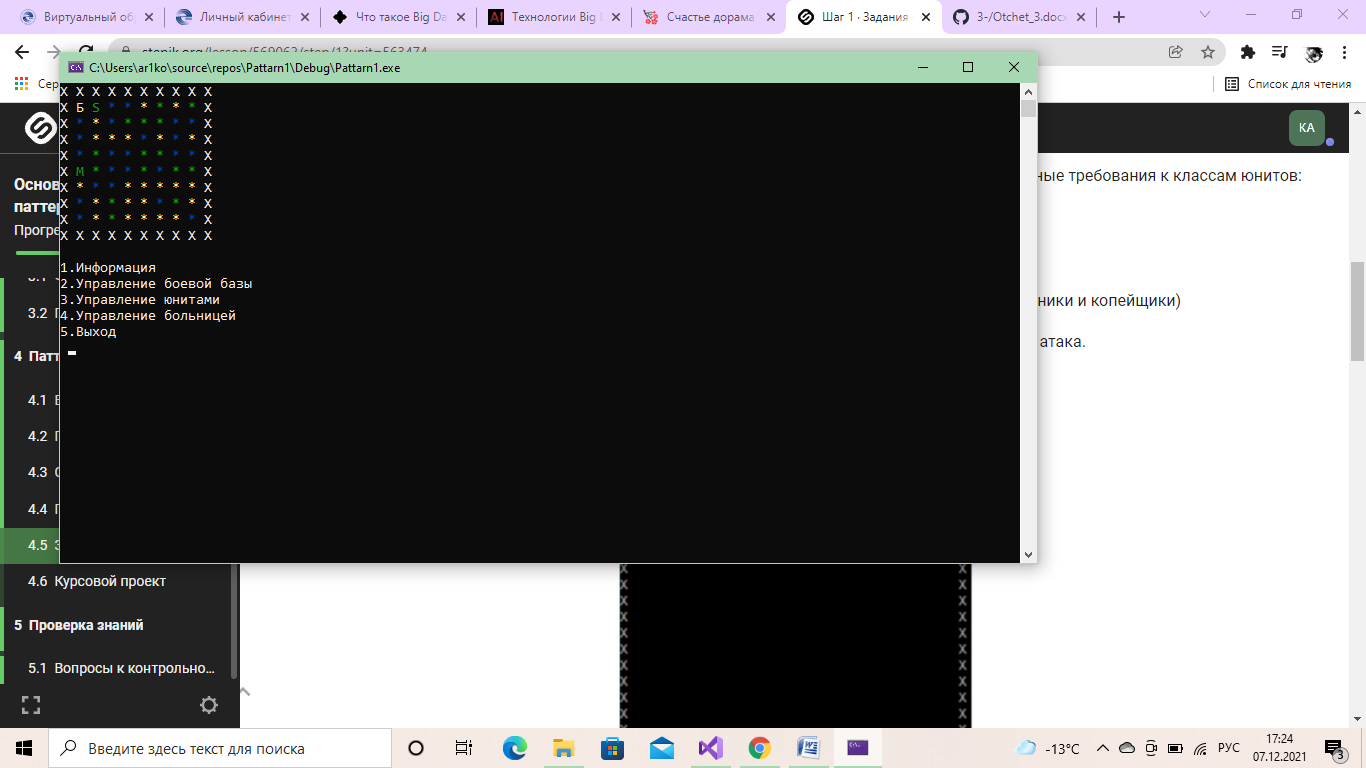
Удаление происходит, если убить юнит.  
3.У меня есть 3 типа юнитов:  
****

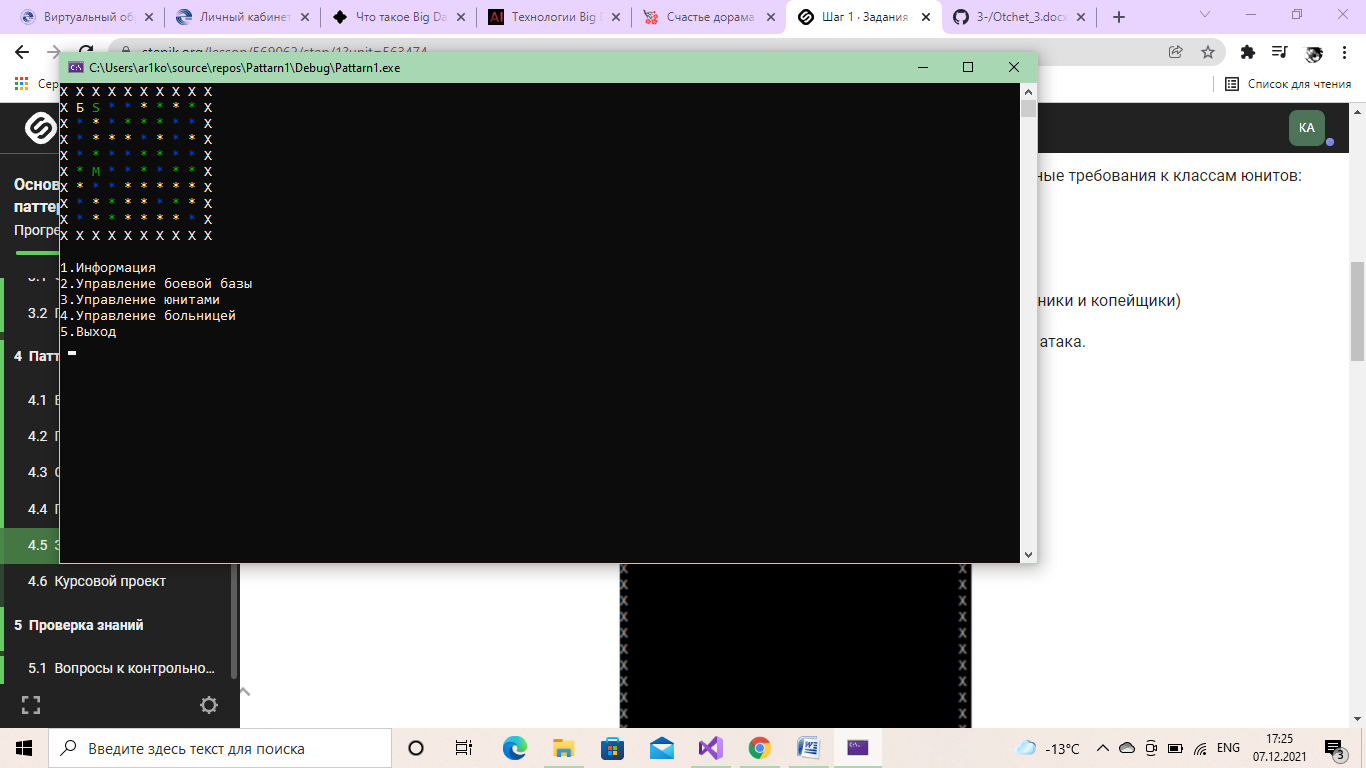
4.У каждого юнита есть разделение на 2 типа

****

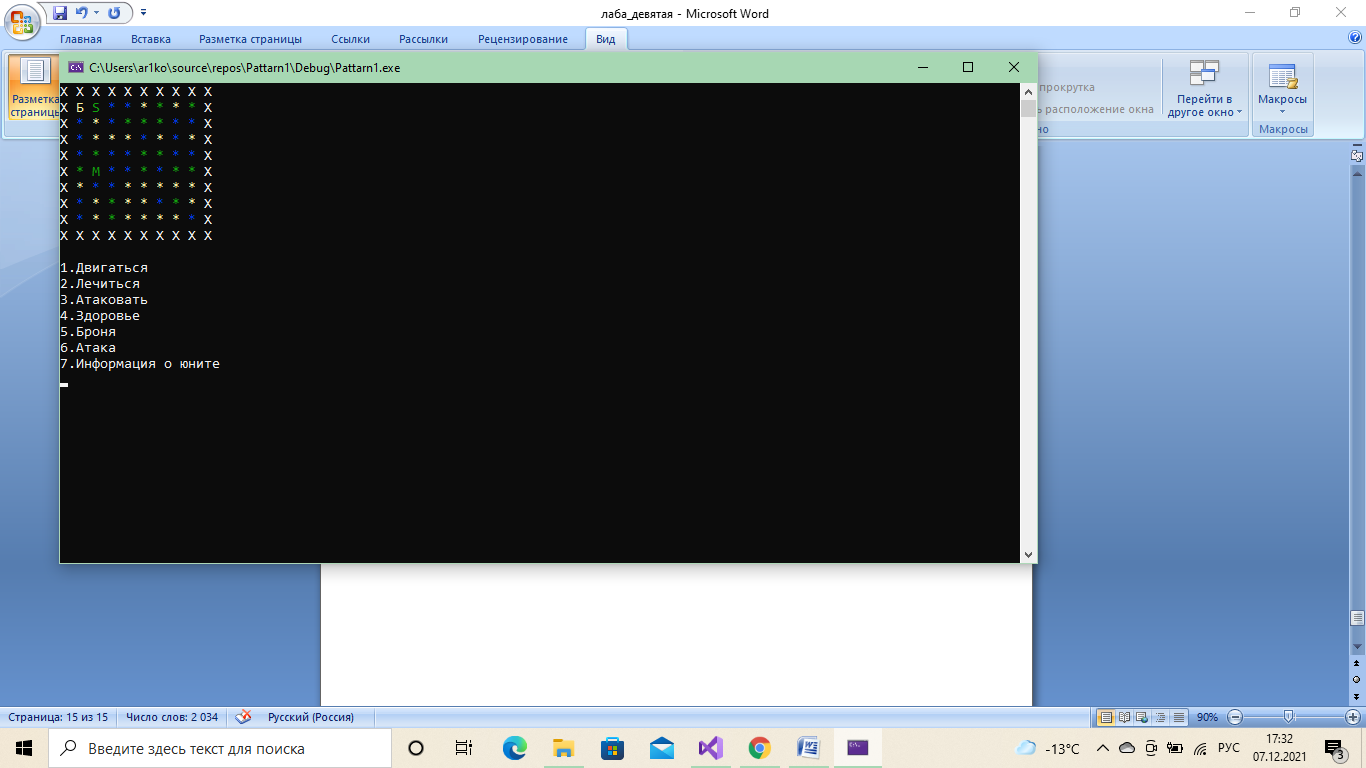
****

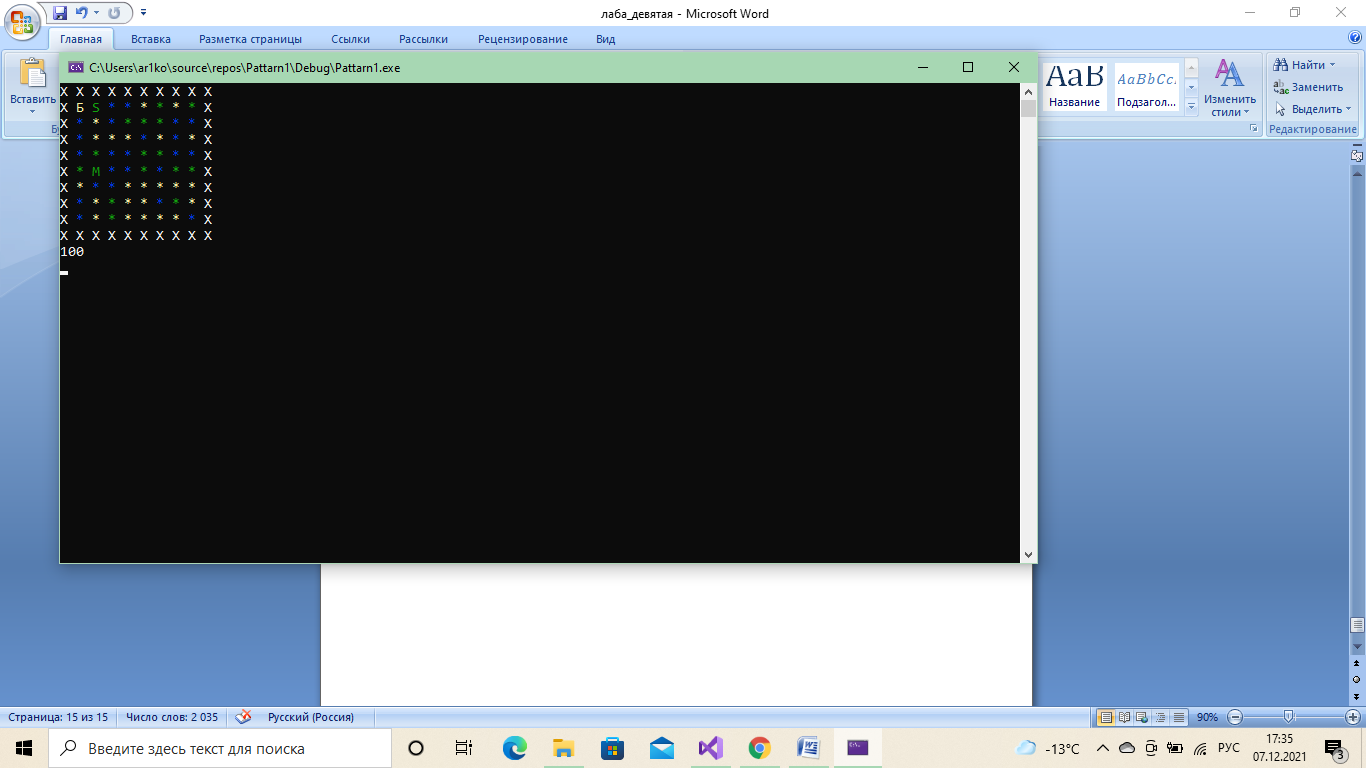
****

5.Юнит умеет перемещаться  


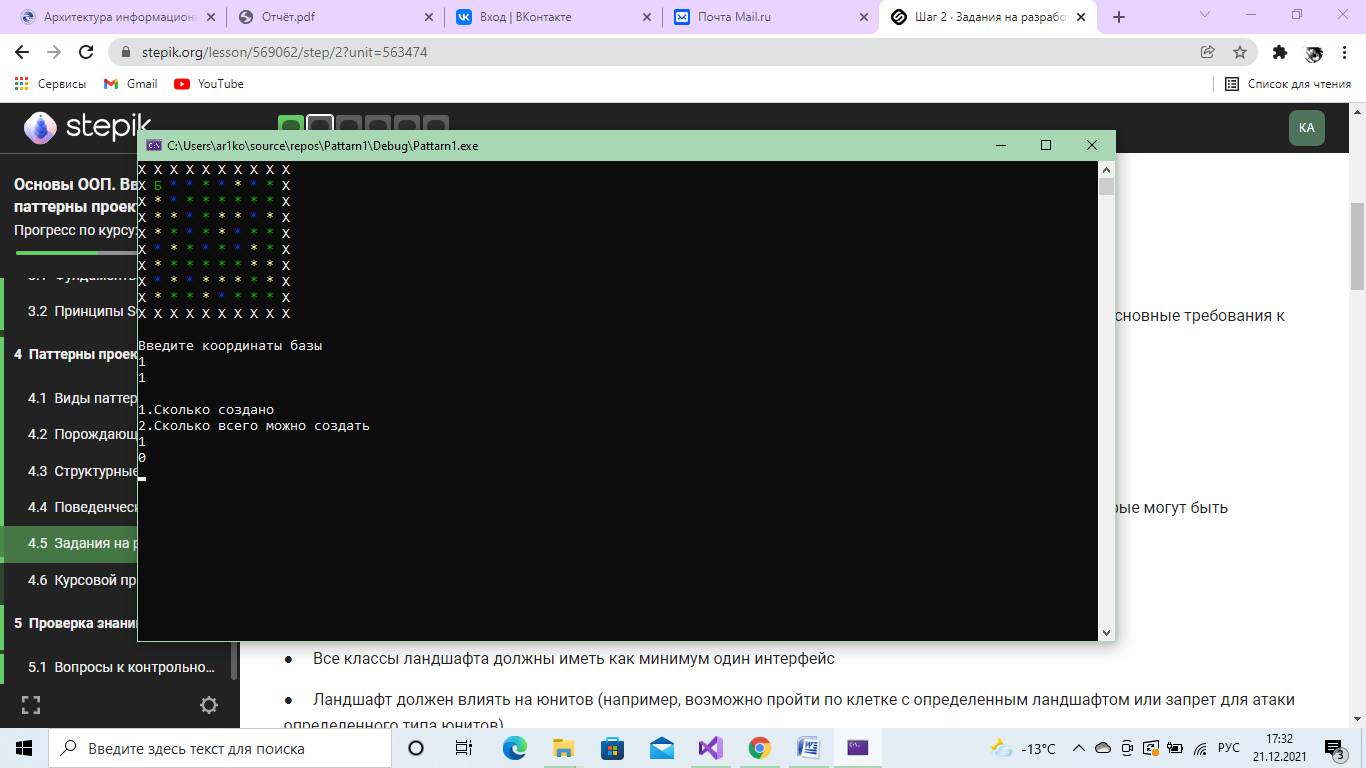


6.Юнит имеет характеристики: броня, здоровье, атака.

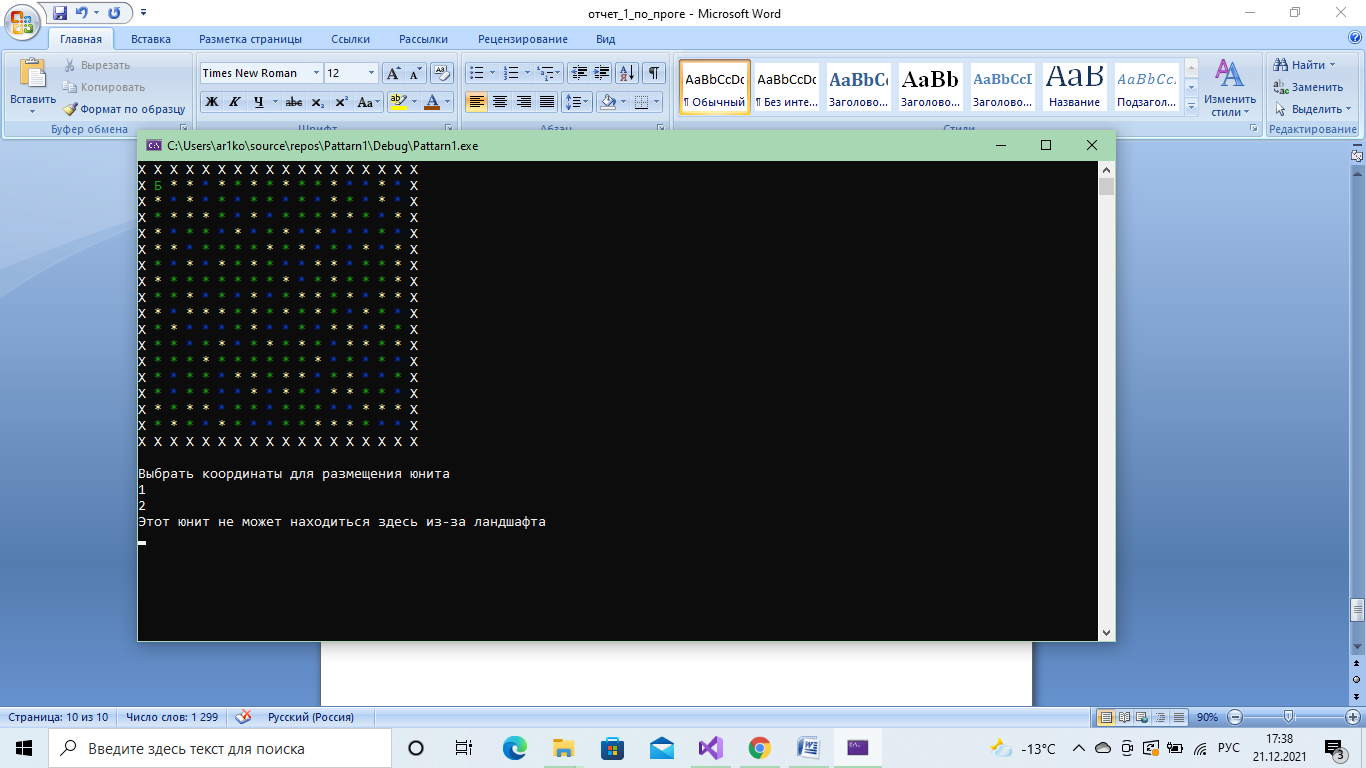
  
Мы можем узнать характеристики юнитов, указав их координаты, вот к примеру здоровье юнита в координате (1,2)



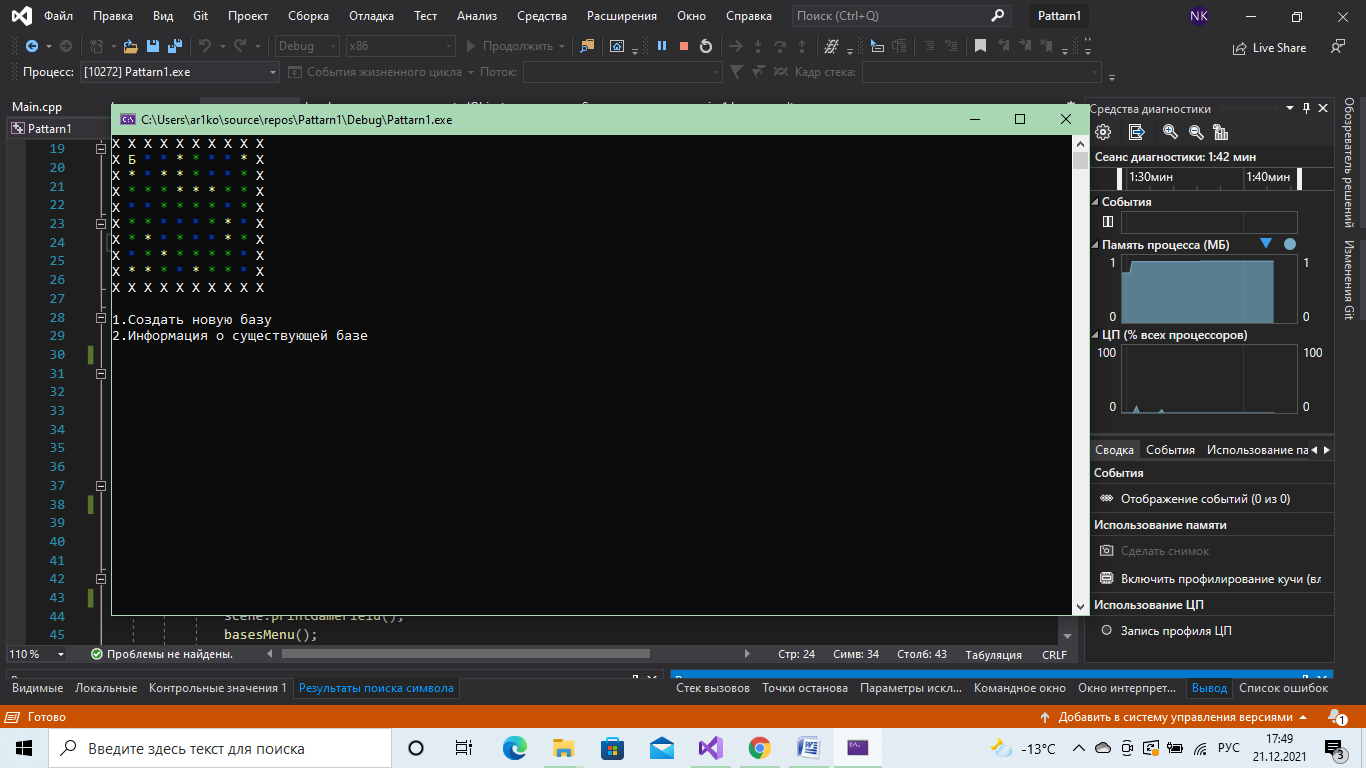
7.Получение информации о юнитах от базы(Сколько можно создать и сколько создали)

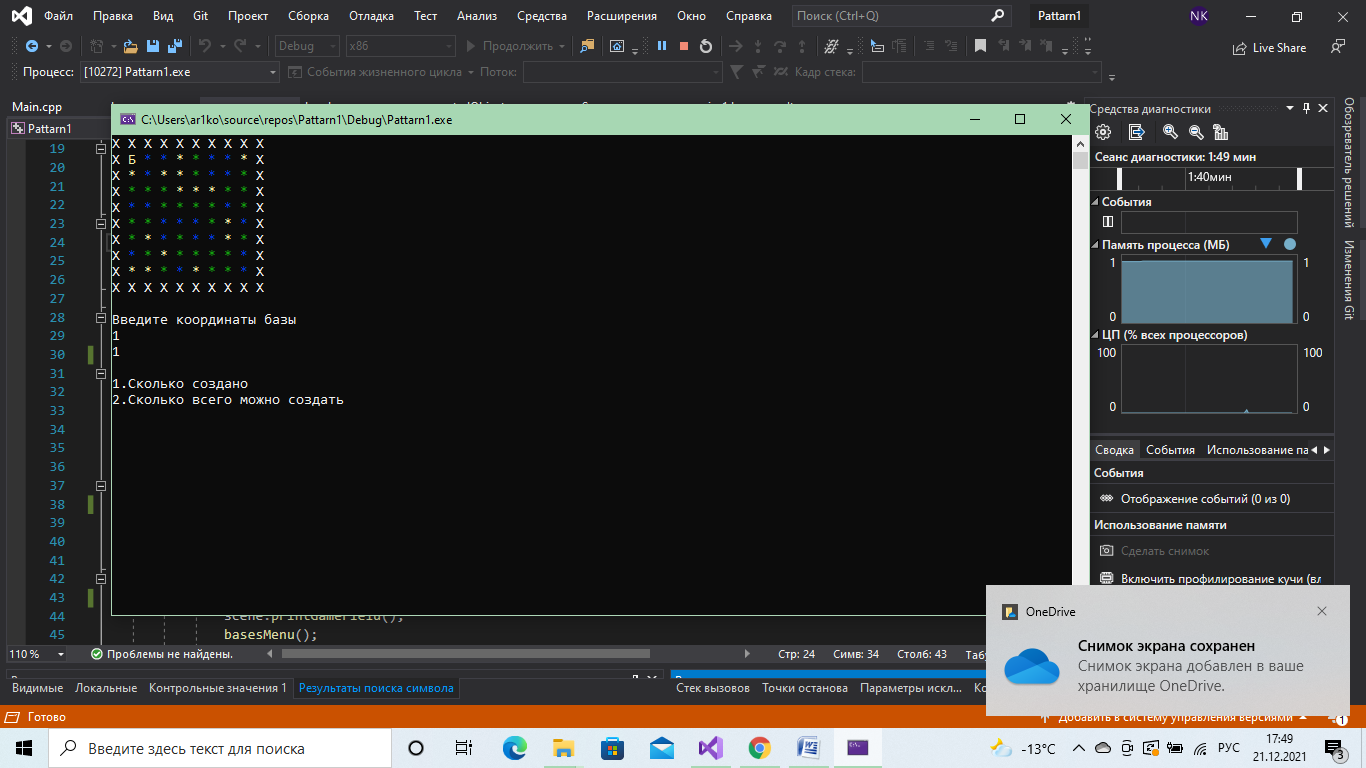


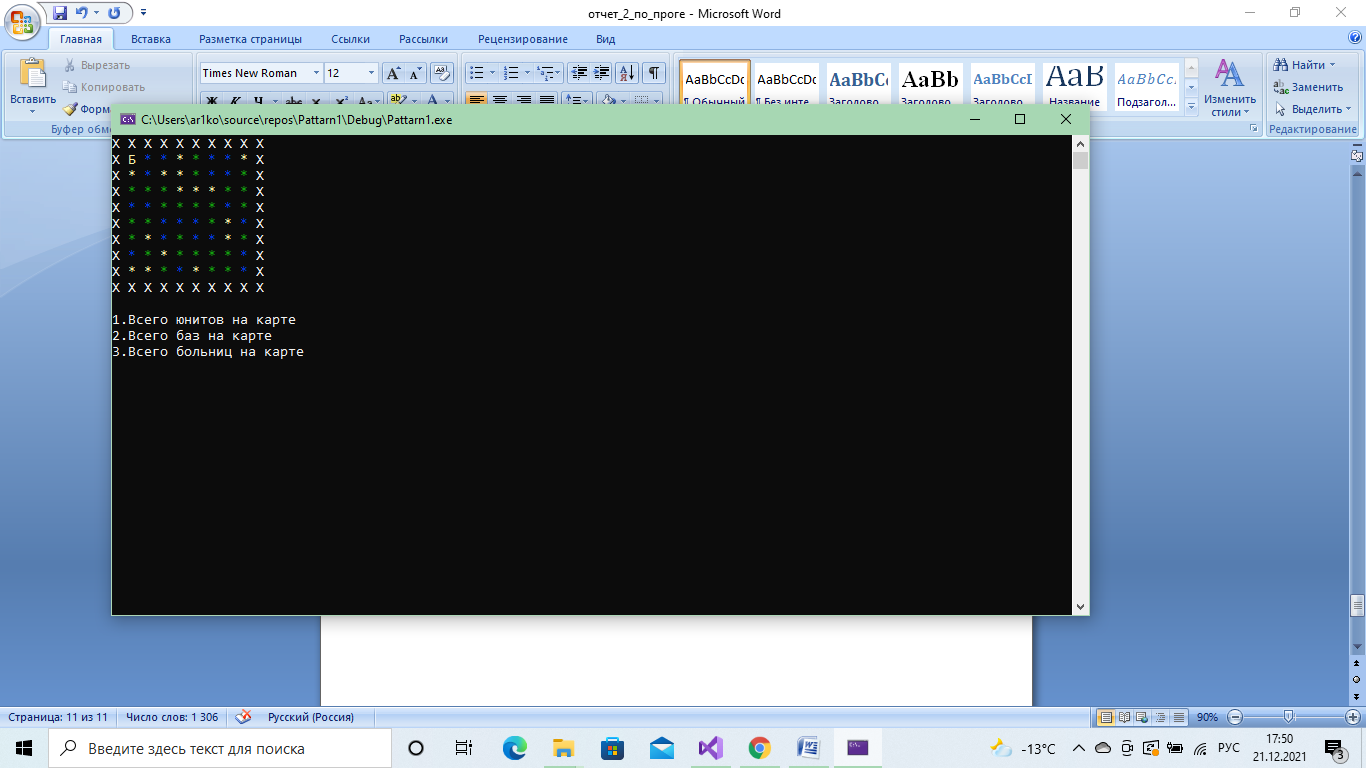
8. Пример , когда юнит не может находиться на определенном ландшафте



9.Управление базой







10. Управление юнитами

