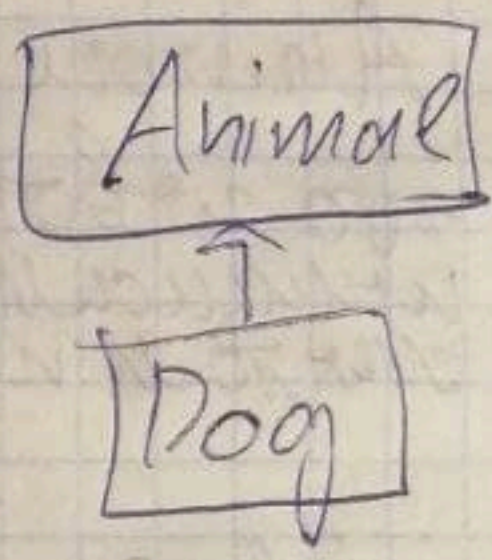
Наследование в ООП.

29.11

```
class Animal {
    public:
        string name;
    public:
        Animal();
        sound();
};
```




```
class Dog: public Animal {
    public:
        Dog(); → Dog(string new_name);
        sound(); };
int main() {
    Dog bobik("Bobik");
};
```

мы вызываем конструктор который на вход получает строку, но у нас этого нет. ошибка.

сделаем это

Если захотим так: Dog(string new_name = "");
то можно как пользовать, так и:

Dog noname;

...

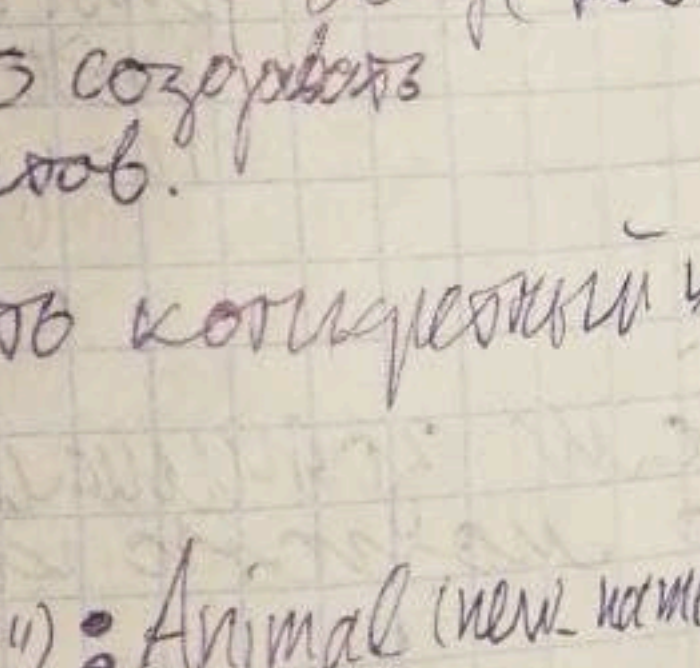
Animal(string new_name = "") : name(new_name) {}

... это говорит что наш конструктор он инициализирует поле name своим аргументом new_name

... class Dog ...
    public:
        string type;
    ...

Мы добавили тобе только в Dog

Теперь создаём Dog:



!!! При создании объектов теперь вызывается конструктор родителя: (по умолчанию)

Animal();
Dog("Bobik");

! Если есть наследование, то создаётся конструктор без аргументов.

Но если хотим вызвать конкретный конструктор, то:

... Dog(string new_name = "") : Animal(new_name) {};
...

Можем управлять так нужной конструктор

Ограничители доступа:

public - доступны снаружи класса

protected - доступны снаружи класса

private - только из внутри класса

Наследование:

class Dog : public Animal {

public ⇒ public

protected ⇒ protected

private ⇒ ~~private~~ недоступен в новом классе

} изменение доступа при наследовании

... : private Animal {

public ⇒ private

protected ⇒ private

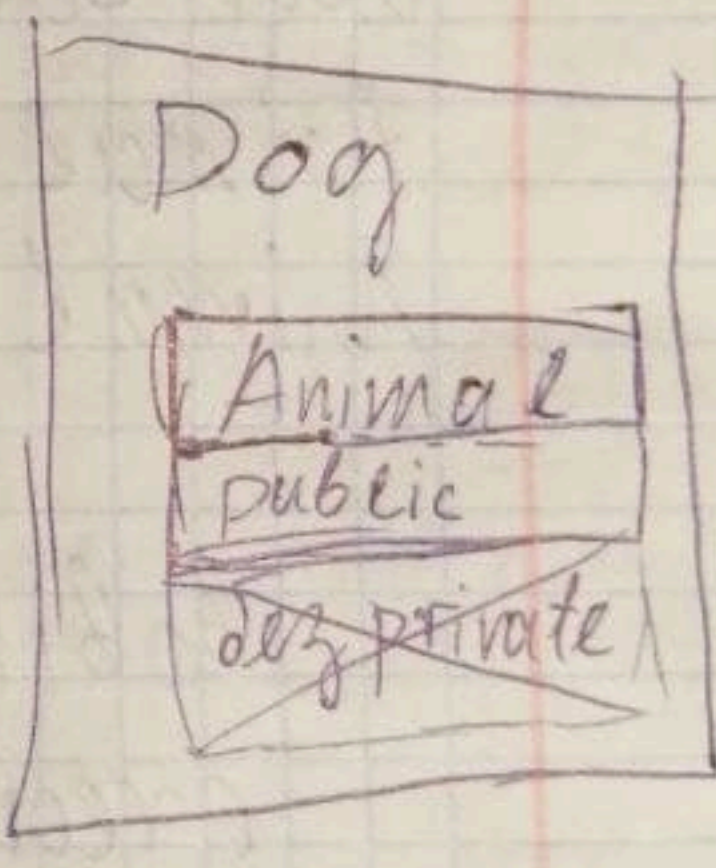
private ⇒ недоступен

... : protected Animal {

public ⇒ protected

protected ⇒ protected

private ⇒ недоступен



Можно в наследовании переписать уровни доступа (если нельзя - переписать нельзя, т.е. private не переопределяется)

в Dog:

protected:

using Animal::sound; // переписали на protected уровень доступа в Dog.

Сокрытие или удаление чистого функционала:
в конкретном классе
в Dog:

..., sound() = delete;

—

Dog Bobik("Bobik");
Animal creature{"X305"};

creature = bobik; // впоследствии creature окажется
снова теперь настоящей bobik,
которая относится к Animal.

Dog bobik("Bobik");
Animal *creature01 = &bobik;
Animal &creature02 = bobik;

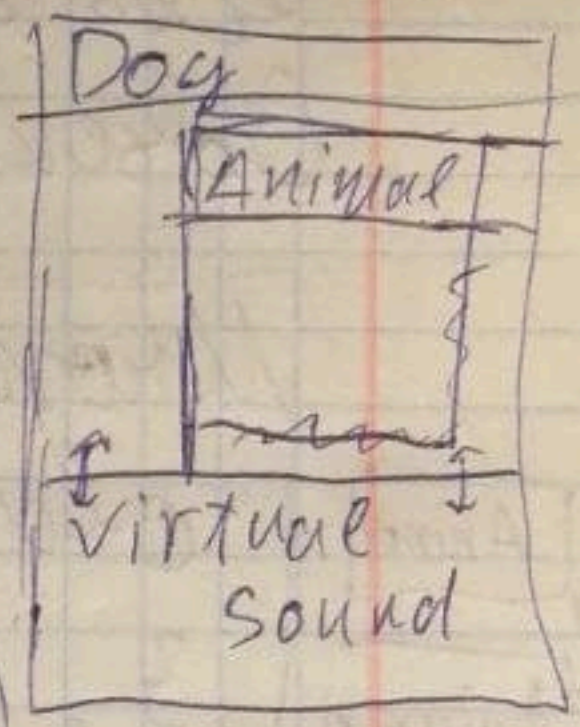
bobik.sound(); // вызовется sound из Dog
creature01 -> sound(); // из Animal
creature02.sound(); // из Animal

! Можно представить указатель на класс Animal как ук-ль на класс Dog (если определённого указывает на объект класса Dog)

dynamic_cast<Dog*>(creature).sound
вызовется sound из класса Dog

Чтобы вызвать sound из Dog, а не родителя Animal нужно сделать её виртуальной в Animal и Dog.

```
... virtual sound();
```



! Если функция виртуальная, то она такова у всех наследников сразу.

Виртуальная функция - это позднее связывание (медленнее)

Деструкторы должны быть виртуальными если есть наследование.

Чтобы избежать ошибки перегружения следующие после ф-и:

```
sound() override;
```

Если ещё final - запрещает наследовать виртуальность, или переопределять функцию.

```
virtual sound() override final;
```

точно переопределяет. Так компилятор отловит переопределение

больше переопределять нельзя.

можно также для классов

```
class Dog final: ...
```

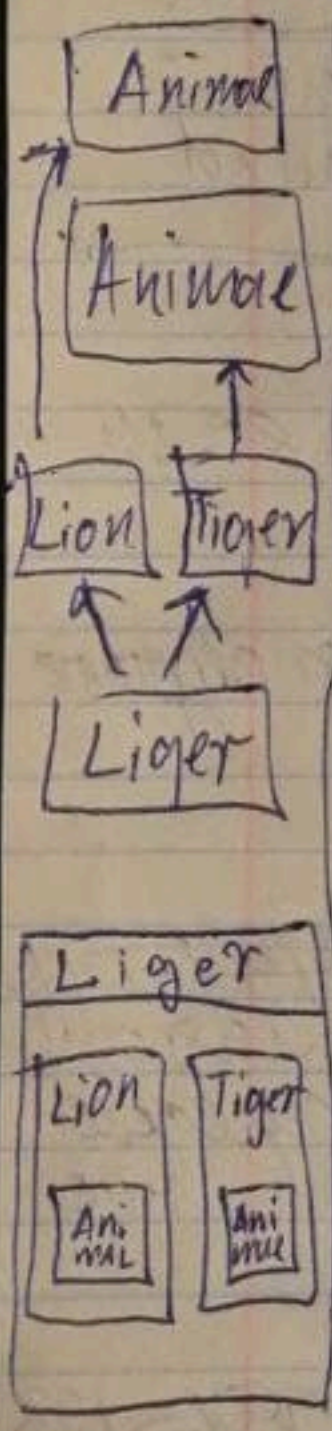
Абстрактные классы: Идея в том, что мы пишем что класс должен делать (его методы,

переменные? в виде кода (работчего), но без фактической реализации методов.

Абстрактный класс - класс у которого есть хотя бы одна абстрактная ф-я.

Объекты абстрактного класса не создаются (ошибка компиляции)

Множественное наследование:




```
class Animal{
    ...
};
class Tiger: public Animal{
    int tail_length;
};
class Lion: public Animal{
    int tail_length;
};
class Liger: public Lion, public Tiger{
};
```

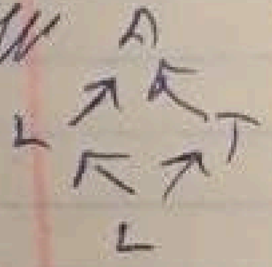
Проблемы:

① Два экземпляра Animal

② Дел одинаковых полей tail_length

Нужно сделать так, чтобы не было неопределённости т.е. нужно быть одинаковых полей и методов у родителей, и родителей.

как способ решения проблемы – виртуальный базовый класс (Animal у нас и тогда хвост)



UML – свод правил

Дружественные функции и классы

```
class Graph{
    private:
        Node* Root;
    public:
        Node* Search(Node *node);

}
class Node{
    private:
        void* data;
        std::list<Node*> neighbors;
        friend class Graph; // теперь из class Graph будут видны поля и методы class Node, т.е. это почти как один класс.
}
```

всё закрыто, кроме для

friend – дружба односторонняя
также можно чисто отдельный метод.

Анонимные объекты:

Dog ("Bobik"); // создание анонимного объекта