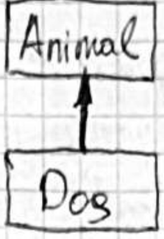
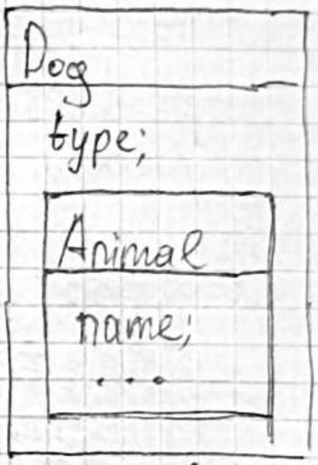
Конспект. ООП в С++

Колесов Сергей
3331506/30401

Наследование в ООП:



```
class Animal {
    public:
        string name;
    public:
                 string new_name="": name(new_name){}
        Animal(); //конструктор
        sound();
}
class Dog: public Animal {   //наследование
    public:
        Dog(string new_name = "");
        Sound();
}
                public:
                    string type; //порода
int main() {
    Dog bobik("Bobik"); //вызов конструктора Dog("Bobik")
    Dog noname; // вызов конструктора Dog()
}
```



порядок вызова

Конструкторы
Animal();
Dog("bobik");

! При создании объектов неявно вызывается конструктор родителя (который без аргументов)

Если присутствует наследие, нужно создавать конструктор без аргументов.
Но если нужно вызвать конкретный конструктор, то:

```
class Dog: public Animal{
    public:
        string type;
    public:
        Dog(string new_name =""): Animal(new_name){};
        Sound();
```

Ограничители доступа;

public - доступны снаружи класса

protected - доступны снаружи класса.

private - доступны только изнутри класса.

При разработке классов лучше использовать private.

Наследование

```
Class Dog: public Animal{
    public => public
    protected => protected       } изменение доступа
                 недоступен      } при наследовании.(public)
    private => ~~private~~
```

```
Class Dog: private Animal{
    public => private            } изменение доступа
    protectted => private        } при наследовании (private)
    private => недоступен.
```

```
Class Dog: protected Animal{
    public => protected          } изменение доступа.
    protected => protected       } при наследовании.
    private => недоступен
```

Для доступа к private, нужно делать public-методы в нужном классе для доступа к ним.

[Можно в наследнике переписать уровень доступа
(если поля нет, то и переписать нельзя, т.е. private не перепис.]

```
Class Dog: public Animal{         |  protected:
    public:                       |      Using Animal:: sound;
        string type;              |  // переписывание на protected
    public:                       |  }  уровень доступа в Dog к
        Dog(string new_name="");  |     методу Sound()
```

Скрытие или удаление лишнего функционала:

```
Class Dog: public Animal{
  public:
      string type;
  public:
      Dog(string new_name = "");
      Sound() = delete; // скрытие метода из класса Dog (удаление
}     это нужно для того, чтобы код не повторялся если он уже есть.
```

ООП помогает структурировать программу

```
Dog bobik("Bobik");
Animal creature("X3 06");
creature = bobik; // в перемен-ую creature скопируется часть
                     переменной bobik, которая относится к Animal
Dog bobik("Bobik");
Animal *creature01 = &bobik; // - указатель  } нужно для вызова
Animal &creature02 = bobik; // - псевдоним   } методов родителя.
   bobik.sound();           // вызывается sound() из Dog
   creature01 -> sound();   // вызывается sound() из Animal
   creature02. sound();     // вызывается sound() из Animal.
void  sound_three_times(Animal & creature){
   creature.sound();
   creature.sound();
   creature.sound();
}
```

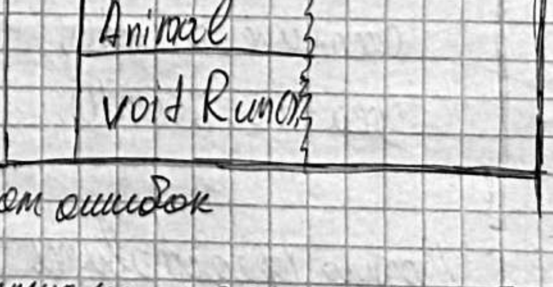
! Можно представить указатель на класс Animal как указатель на класс Dog (если тот действительно указывает на объект класса Dog

* с помощью инструкции dynamic_cast

```
Dog bobik("bobik");
Animal *creature = &bobik;
(dynamic_cast<Dog*>(creature)).sound()}; //вызовется sound()
                                           из класса Dog
```
„Понижающее приведение типов"

для классов Animal и Dog (с переопределённой ф-ей sound())

```
void sound_three_times(Animal & creature){
    creature.sound();      будет вызван sound для Animal
    creature.sound();      Чтобы вызывалась ф-я из Dog
    creature.sound();      Нужно объявить ф-ю виртуально.
}
class Animal{
    public:
        string name;
    public:
        Animal(string new_name=""):name(new_name){};
        virtual sound();
~~}~~ void Run();
}
class Dog: public Animal{
    public:
        string type;
    public:
        Dog(string newname="");
        virtual sound() override;
        void Run();          для защиты от ошибок
}
```



! [~~Если~~ Если вы определили ф-ю виртуально, то она должна быть виртуальной у всех потомков.]

Виртуальные функции это позднее связывание (медленее)

! [Деструкторы должны быть виртуальными если есть наследование]

Final - запрещает в дальнейшем наследовать класс и переопределять функции.

```
virtual sound()  override final; // значит, что больше
...                                 переопределять её нельзя;
```

```
class Dog final : public Animal { // значит что нельзя
...                                  создавать наследников класса Dog
}
```

Для предотвращения ошибок!

Абстрактные классы:

Идея ~~устройства~~ интерфейса: вы пишете, что класс должен делать (методы, переменные), в виде кода - код рабочий (компилируется), но без фактической реализации.

```
class IAnimal {
   public:
      string get_name(){};
      virtual void sound() = Ø; // ф-я является абстрактной
}
```

абстрактные ф-ии должны быть переопределены в наследниках!

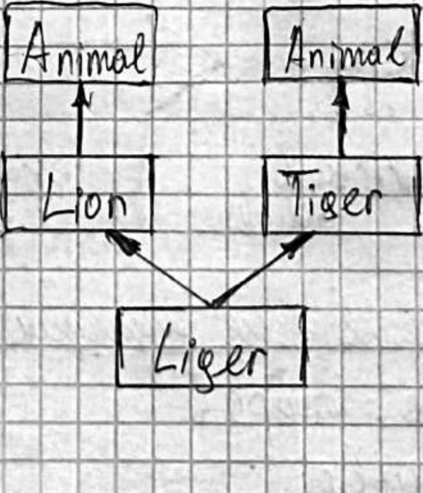
Абстрактный класс - класс у которого есть хотя бы одна абстрактная ф-я.

Объекты абстрактного класса не создаются (ошибка компиляции)

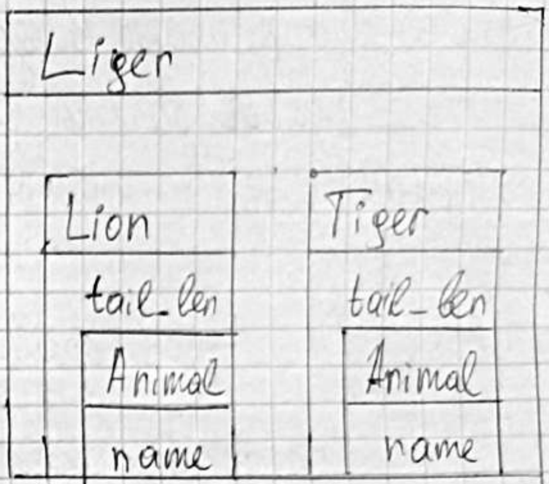
```
class Animal : public IAnimal {
   ...
}
```

Множественное наследование:

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

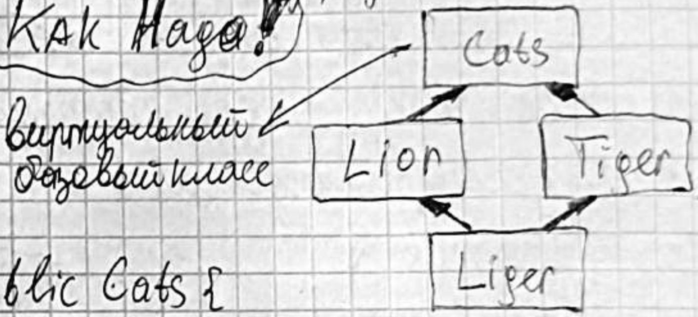


Проблемы: 1) Два экземпляра Animal внутри класса
2) Два одинаковых поля tail_length
Нужно делать так, чтобы не было неопределённости, т.е. не должно быть одинаковых полей и методов у непосредственных родителей (и вообще у родителей)

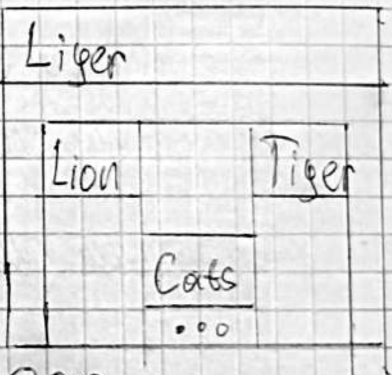


```
class ~~Animal~~ Cats {
    string name;
    int tail_length;
}
```

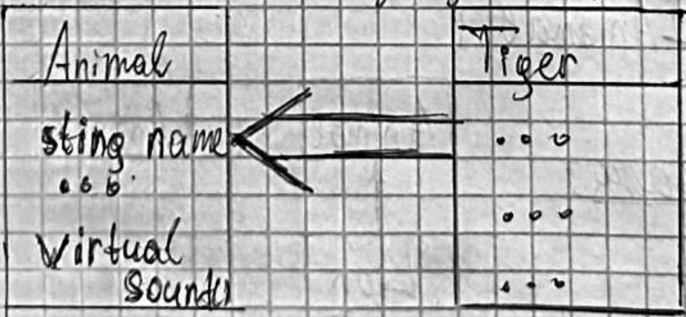
! Как Надо !



```
class Tiger: virtual public Cats {
  ...
}
class Lion: virtual public Cats{
  ......
}
class Liger: public Lion, public Tiger{
  ...
}
```



Как выглядит структура программы с ООП



UML
(способ представления ООП)

Дружественные функции и классы

```
class Graph{
    public:
        Note* ~~R~~root;
    public:
        Note* Search(Note *note
}
```

```
class Note {
    private:          // т.к. поля private к ним нет доступа.
        void* data; //      для доступа используем дружественный клас
        std::list <Note*> neighbors;
        friend class Grapth; //теперь из класса Graph, будут
}                                  видны поля и методы класса Note
```

т.е. это почти один класс, но разделённый на два.

friend - дружба односторонняя, то есть:

„Note is Friend of Graph"

но не наоборот, из Note private методы и переменные Graph недоступн

A-friend of B
B-friend of C => A - не друг C (не транзитивность

```
class Graph {
    private:
        Note* root;
    public:
        Note* Search(Note *note);
}
class Note {
    private:
        void *data;
        stg: list <Note*> neighbors;
        friend Note* Graph::Search(Note* note);
}
```

Метод Search - друг класса Note, т.е. из него доступны private поля Note

Анонимные объекты

```
Dog("Bobik");   // создание анонимного объекта, он не хранится
Dog bobik = Dog("bobik");          и сразу удаляется
Dog("Bobik").sound(); // вызов метода, после чего объект исчезнет.
```

Наследование:

1. Модификаторы доступа при наследовании
   - public
   - private
   - protected
2. Порядок вызова конструкторов при наследовании
3. Порядок вызова деструкторов при наследовании
4. Повышающее приведение типов (к наследнику)
5. По~~выша~~ нижающее приведение типов (к родителю
6. Обрезка объектов (animal = dog);
7. virtual методы (virtual деструктор)
8. Абстрактный класс (интерфейс)
9. Множественное наследование (виртуальные классы)
10. Дружественные методы и классы
11. Анонимные объекты;