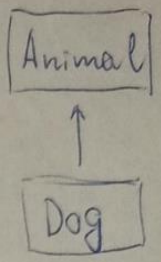
Наследование в ООП

Конструкторы в наследовании

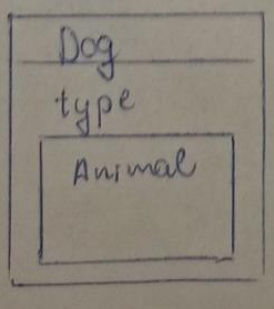
Рассмотрим наследование на примере классов Dog и Animal.

```
class Animal {
public:
  string name;
public:
  Animal(string name = " "): name(new_name) {};
  sound();
}
class Dog : public Animal {
public:
  string type;
public:
  Dog(string new_name = " ");
  sound();
}
int main(){
   Dog Bobik ("Bobik");     // Dog ("Bobik");
   Dog noname;              // Dog()
}
```



Как можно заметить, должны вызваться соответствующие конструкторы для класса Dog. Однако, на самом деле, вызовется еще и конструктор для класса Animal, потому что объект производного класса есть по сути два объекта, слитых воедино по определенным правилам.

Показать это можно следующей диаграммой:



Значит, для 1 строки функции main будут вызваны следующие конструкторы:

```
Animal();
Dog("Bobik");
```

> При создании объектов неявно вызывается конструктор родителя по умолчанию (без параметров)

Поэтому, если есть наследование, то нужно создать конструктор без аргументов

Однако, если нужно вызвать конкретный конструктор, то поступают следующим образом:

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

Ограничители доступа.

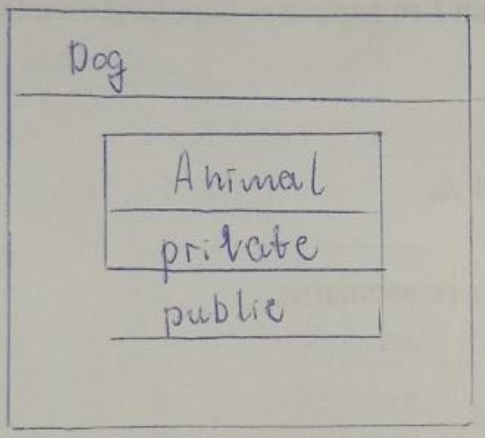
public - доступны снаружи класса
protected - доступны снаружи класса
private - доступны только внутри класса

```
class Dog : public Animal {
    public => public
    protected => protected
    private => недоступен
```

class Dog: private Animal {
public => private
protected => private
private => недоступны

class Dog : protected Animal {
public => protected
protected => protected
private => недоступны

Показать доступность полей класса можно так:



Также можно переписать уровень доступа (если к этому полю есть доступ)

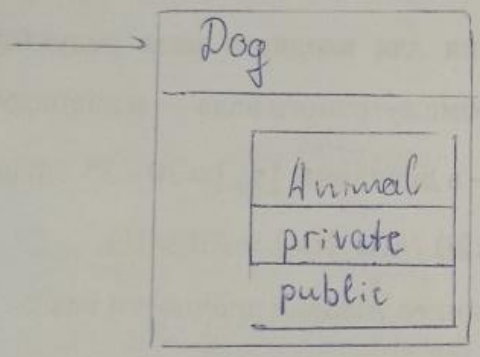
```
class Dog : public Animal {
public:
  string type;
public:
  Dog(string new_name = "");
protected:
  using Animal::sound; // переписали на protected уровень
                       доступа в Dog к методу
                       sound();
}
```

Сокрытие или удаление лишнего функционала.

Мы можем скрыть (удалить) метод при помощи ключевого слова delete.

```
class Dog: public Animal{
public:
  string type;
public:
  Dog(string new_name = " ");
  sound()= delete; // скрыли метод из класса Dog.
};
```

Приведение типов.



Рассмотрим, что произойдет если объекту класса Animal присвоить объект класса Dog.

```
Dog bobik ("Bobik");
Animal creature ("X305");
creature = bobik; // В переменную creature скопируется
                  часть переменной bobik, которая
                  относится к Animal.
```

Рассмотрим теперь, какое поле sound() вызовется в разных ситуациях:

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

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

```
void sound_three_times(Animal &creature){
    creature.sound();      // три раза вызовится
    creature.sound();      // sound() из Animal
    creature.sound();      //
}
```

Такое приведение типов называется повышающим.

Можно представить указатель на класс Animal как указатель на класс Dog (если он действительно указывает на объект класса Dog)

```
Dog bobik("Bobik");
Animal *creature = &bobik;
(dynamic_cast<Dog*>(creature))->sound();
```

вызовется sound из класса Dog.

Такое приведение типов называется понижающим.

Виртуальные функции -

Рассмотрим ранее введенные классы Animal и Dog и функцию sound_three_times.

```
class Animal{
public:
  string name;
public:
  Animal(string new_name = " "): name(new_name){};
  void sound();
};
```

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

```
void sound_three_times(Animal &creature){
    creature.sound();
    creature.sound();
    creature.sound();
}
```

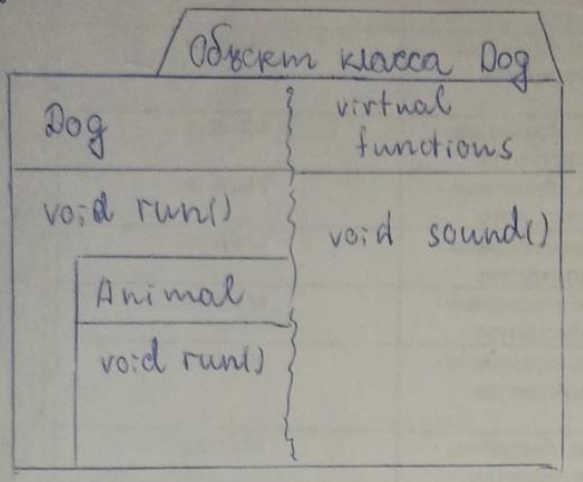
Даже если передать в функцию объект класса Dog, все равно вызовется sound() для Animal. Как сделать так, чтобы вызывалась функция из класса Dog? Нужно объявить функцию виртуальной.

```
class Animal{
public:
  string name;
public:
  Animal(string new_name = "n"): name(new_name){};
  virtual void sound();
  void run();
};
```

```
class Dog : public Animal{
public:
  string type;
public:
  Dog(string new_name = "n");
  virtual void sound() override;
  void run();
};
```

Теперь то, какой метод будет вызван в функции void sound_three_times(Animal &creature), зависит от того, какой объект будет передан в качестве параметра.

Виртуальные функции можно представить следующим образом:



! Если вы определили функцию как виртуальную, то она должна быть виртуальной у всех потомков.

Виртуальные функции это позднее связывание.

! Деструкторы должны быть виртуальными, если есть наследование

Для того, чтобы избежать ошибок, используется ключевое слово override, которое говорит компилятору, что метод переопределен.

Кроме того, есть ключевое слово final, которое запрещает в дальнейшем использовать класс и переопределять функцию.

```
virtual sound() override final; // больше переопределять нельзя
```

```
class Dog final : public Animal { // нельзя создавать наследников класса Dog
};
```

Абстрактные классы:

Идея интерфейса: Вы пишите, что класс должен делать (методы, атрибуты), в виде кода - код рабочий (компилируемый), но без фактической реализации.

```
class IAnimal{
public:
  string get_name(){};
  virtual void sound = 0; //функция является абстрактной
}
```

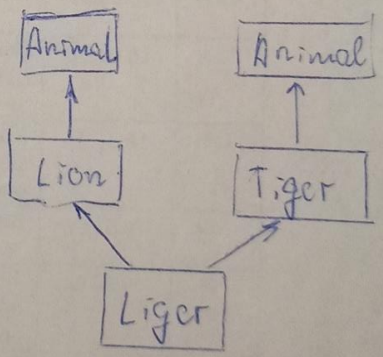
Абстрактный класс - класс, у которого есть хотя бы одна абстрактная функция.

Объекты абстрактного класса не создаются (ошибка компиляции)

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

Множественное наследование

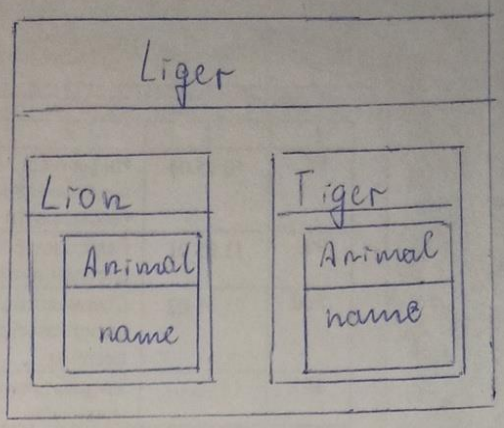
```
class Animal{
  string name;
}

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 переопределений, т.е. не должно быть одинаковых полей и методов у непосредственных родителей (и вообще родителей)

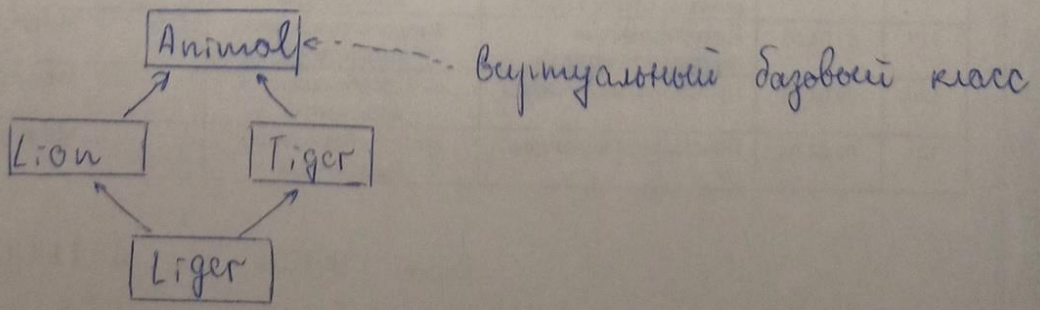
Решение проблемы:

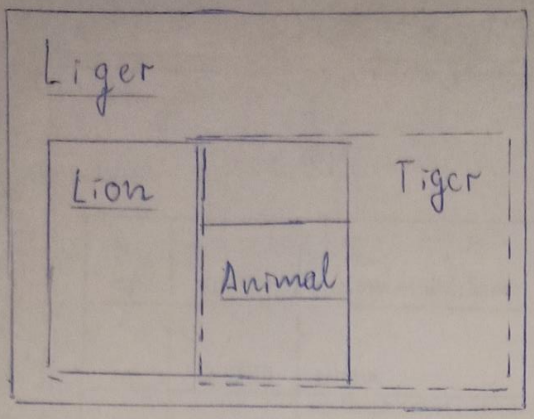
```
class Animal{
    string name;
    int tail_length;
}
```

```
class Tiger : virtual public Animal{
    ...
}
```

```
class Lion: virtual public Animal{
    ...
}
```

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





Дружественные функции и классы.

```
class Graph{
private:
   Node* root;
public:
   Node* search (Node * node);
}
class Node{
private:
   void * data;
   std:: list < Node*> neibors;
   friend class Graph;   // теперь из класса Graph
}                           будут видны поля класса
                            Node.
```

friend - дружба односторонняя

Node is friend of Graph, но не наоборот.

Из Node private - методы и поля Graph недоступны.

A - friend of B

B - friend of C

A - не друг C

```
class Graph{
private:
  Node* root;
public:
  Node * search(Node * node);
};

class Node{
private:
  void * data;
  std::list<Node*> neibors;
  friend Node* Graph::search(Node * node);
};
```

метод друг класса Node, т.е. из него доступны private поля Node.

Анонимные объекты

```
Dog("Bobik"); // создание анонимного объекта
Dog bobik = Dog("Bobik");
Dog("Bobik").sound(); // можно вызывать метод объекта (но после объект исчезает).
```