Семенов А.
3331506/90003

Наследование в ООП

$$\begin{tikzcd}\boxed{\text{Animal}} \\ \boxed{\text{Dog}} \arrow[u]\end{tikzcd}$$

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

```
class Dog : public Animal{   // создаем новый класс,
                                конкретизирующий старый;
                                новый класс „по
                                наследству" забирает
                                свойства и методы
                                старого класса
    public:
        Dog (string new_name = " ");
        sound()
    public:
        string type;
                                создаются как бы два
                                класса слитых воедино
                                (в Animal нет type)
}
```

```
int main() {
    Dog bobik ("bobik");     // Dog ("bobik");
    Dog noname;              // не создаем констр. по умолчанию
                             т.к по умолчанию string = " "
```

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

Если !! есть наследование – нужно создавать конструктор без аргументов

Но так же можно вызвать конкретный конструктор. Пример:

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

Сначала вызывается конструктор родителя, затем самого объекта.

<u>Ограничители доступа:</u>

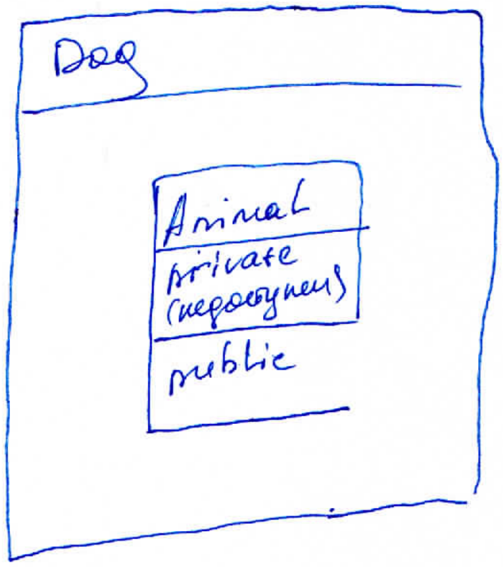
public - доступен снаружи класса
protected - доступен снаружи класса (с ограничением)
private - доступен только изнутри класса

<u>Изменение доступа при наследовании:</u>

class Dog : <u>public</u> Animal {
public -> public
protected -> protected
private -> недоступен

class Dog : <u>private</u> Animal {
public -> private
protected -> private
private -> недоступен

class Dog : <u>protected</u> Animal {
public -> protected
protected -> protected
private -> недоступен



<u>Важно!</u>
Можно в наследнике переписать уровень доступа
(private нельзя переписать, т.к нам нет)

```
class Dog : public Animal {
public:
    string type;
public
    Dog (string new_name = "");
protected:
    using Animal :: sound; // таким образом
                           переписали ур. доступа
                           на protected
}
```

2

Сокрытие или удаление лишнего функционала

```
class Dog : public Animal {
    public:
        string type;
    public:
        Dog (string new_name = "");
    sound () = delete;  // удаление (скрываем)
                          метод в новом классе
}
```

```
Dog bobik ("bobik");
Animal creature ("X305"); // в creature скопируется
                          часть Dog, которая относит-
                          ся к Animal.
```

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

Как вызывать функции по указателям?

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

Необходимо, если нужно работать напрямую с родительским классом

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

ВАЖНО! Можно представить указатель на класс Animal как указатель на класс Dog (если он действительно указывает на объект класса Dog) ~~Dog bobik ("b~~

3

```
Dog bobik ("bobik");
Animal *creature = &bobik
```

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

„понижающее приведение типов"

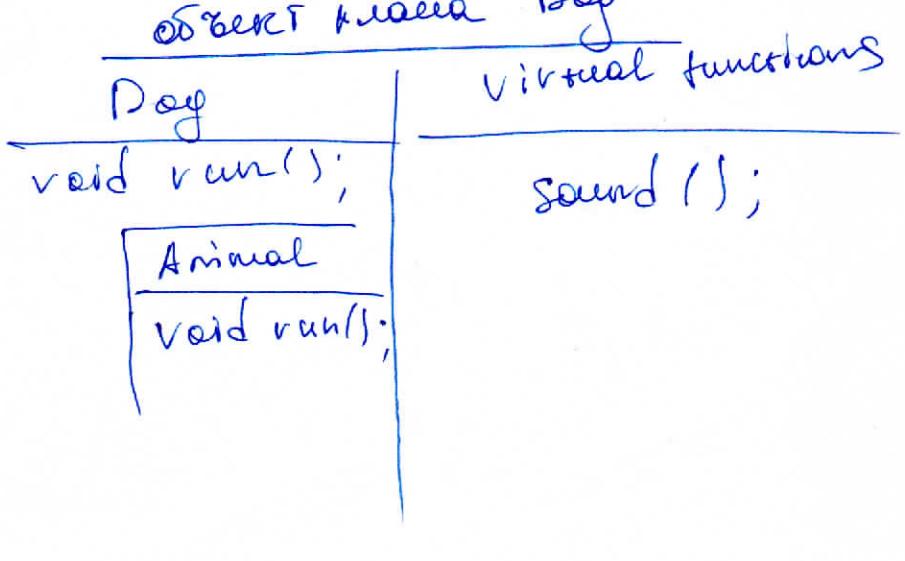
обратное тоже работает („повышающее")

Виртуальные ф-ции

Если есть две функции sound() в Dog и в Animal, то в Dog (наверное) она переопределяется.

В примере sound_three_times(Animal &creature) вызовется sound() из Animal. Если хотим sound() из Dog то нужна виртуальная ф-ция.

```
class Animal {
    =//=
    public:
      virtual sound(); void run();
}
class Dog: public Animal {
    —//—
    public:
      virtual sound(); void run();
}
```



Важно! Если определили функцию как виртуальную, то она д.б виртуальная везде (во всех наследниках)

Вызов виртуальной функции называется поздним связыванием (работают чуть медленнее)

Оч. Важно! Деструкторы д.б. виртуальными (если есть наслед-е

Нужно создавать виртуальные ф-и с одинаковыми параметрами, иначе будет ошибка, которую не покажет компилятор, т.к это все еще корректное определение:

```
class Animal {
    virtual sound();
}
```

```
class Dog: public Animal
{ virtual sound
        (int N)
}
```

НО: можно написать ключевое слово override:

```
class Dog: public Animal { ....
virtual sound() override; }
```

тогда компилятор обновит ошибку и скажет об этом.

Так-же есть ключ. слово final, запрещающее наследовать класс и переопределять функцию:

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

Так же для класса

```
class Dog final : public Animal {
    ....
}
```

↑ нельзя создавать наследников класса Dog

Интерфейс или абстрактные классы (API)

Идея интерфейса: пишем, что класс должен делать (методы, переменные), в виде кода - код рабочий (компилится), но без фактической реализации.

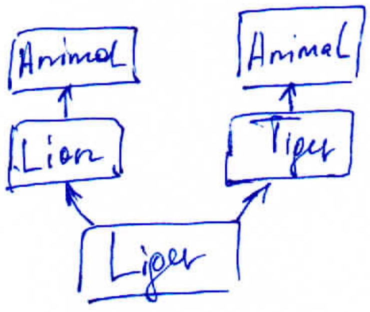
5

```
class Animal {
  private:
     String Name;
     int age
  public:
     string get_name() {};
     virtual void sound() = 0 // ф-я абс.
```

Абстрактный класс - класс с абстрактной/ми ф-ей/ми. Компилятор не даст создавать объекты абстрактного класса.
Необходимо в частности для работы в коллективе где заранее нужно договориться о методах и св-х.

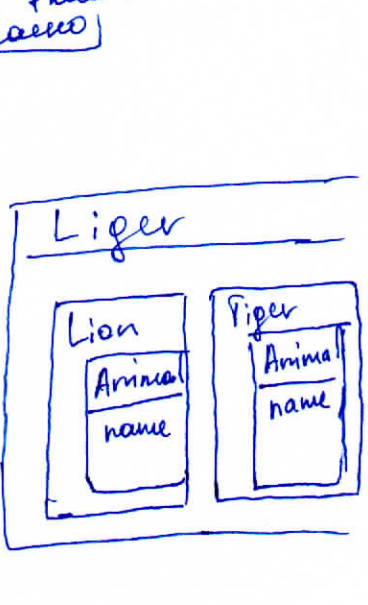
Множественное наследование

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



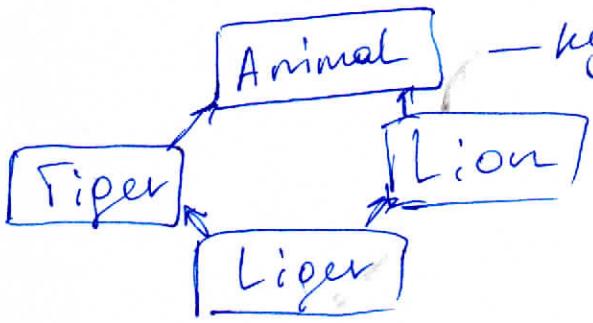
ВАЖНО! Игрушки дьявола, лучше не пользоваться ими
класс + интерфейс. класс + класс (опасно)

Проблемы:
1) Два экземпляра Animal внутри класса
2) Два одинаковых поля tail_length



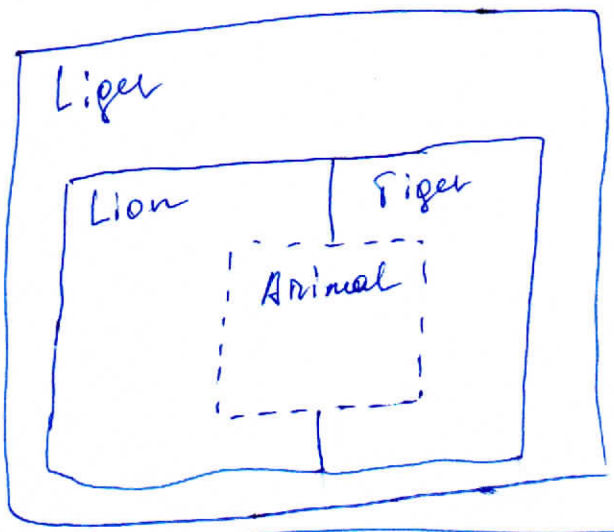
Можно создать еще один класс и древнего предка кошачьих и потерялось хвост.

Нужно сделать так, чтобы не было неопр-ти,
т.е не должно быть одинаковых полей и
методов у непоср. родителей (и вообще у родителей)



— нужно сделать виртуальным

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



Дружественные ф-и и классы

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

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

важно. friend – только в одну сторону.

A friend of B
B - frien of C
A - не друг C.

Также дружеств. можно делать только отдельному метод

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

Анонимные объекты

Dog ("Bobik"); // создание анонимного объекта нельзя получить ссылку, но можно присвоить

```
Dog Bobik = Dog ("Bobik");
Dog("Bobik"). soud () ; // "гавкнул и пропал"
```