Наследование в ООП | Есть 2 класса:

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

Инициализируем name значением new_name

При создании объектов-наследников неявно вызывается конструктор по умолчанию родителя (тот, что без аргументов).

Поэтому, если в коде есть наследование - стоит создавать такие конструкторы

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

эта часть показывает, что Dog - наследует класс Animal

Однако, когда нам нужен конкретный конструктор, пишем:
Dog(string new_name = " "): Animal(new_name) {};

Ограничители доступа: Поля public и protected доступны снаружи классов, private - нет.

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

```
class Dog: private Animal {
    public -> private;
    protected -> private;
    private -> здесь недоступен
};
```

```
class Dog: protected Animal {
    public -> protected;
    protected -> protected;
    private => здесь недоступен
};
```

Можно переписывать уровни доступа, если надо (private - нельзя, т.к. он недоступен -> его нет -> -> нечего переписывать)

```
§ в class Dog: public Animal {
protected:
    using Animal::sound(); — теперь в Dog он protected, а по умолчанию был бы public.
};
```

Сокрытие или удаление лишнего функционала:

```
class Dog: public Animal {
public:
    sound() = delete - скрыли / удалили метод из Dog
```

```
Dog bobik("bobik");
Animal creature("хзОбъ");
creature = bobik // в "creature" копируется кусок переменной "bobik", относящийся к Animal
```

```
Dog bobik("bobik");
Animal *creature01 = &bobik;
Animal &creature02 = bobik;
bobik.sound(); // „sound" из Dog
creature01 -> sound(); // „sound" из Animal
creature02.sound(); // „sound" из Animal
```

Это было „повышающее" приведение типов.

Можно представить указатель на класс Animal как указатель на класс Dog (если это правда так)!

```
Dog bobik("bobik");
Animal *creature = &bobik;
dynamic_cast<Dog*>(creature)->sound(); // вызовется „sound" из Dog.
```

Это - „понижающее приведение типов."

Виртуальные функции | Для наших классов введём ф-ю: void sound_three_times(Animal &creature)
{
creature.sound();
creature.sound();
creature.sound();
}

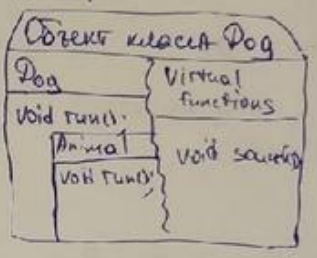
Даже передав в ф-ю объект Dog'а, вызовется sound() для Animal. Как сделать так, как вызывать sound() из Dog?
Нужно объявить ф-ю виртуальной!

```
class Animal {
public:
    ...
public:
    ...
    virtual sound();
    void run();
};
```

и

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

После этого, вызов метода будет определяться объектом, передаваемым в ф-ю.
Вот так „вызядят" виртуальные ф-ии:



Если вы определяете ф-ю как виртуальную, тогда у.б. виртуальной у всех потомков. !
Виртуальные ф-ии - это поздние связывание.
Деструкторы должны быть виртуальными, если есть наследование.
Кроме „override", показывающего, что метод переопределён, есть еще „final", запрещающий последующее наследование класса или переопределение ф-ии.
virtual sound() override final - больше переопределить нельзя.
class Dog final: public Animal {} - нельзя создавать наследников класса Dog.

Абстрактные классы Идея интерфейса: вы пишете, что класс должен делать (методы, применение), в виде кода - этот код рабочий (компилируемый), но без фактической реализации.

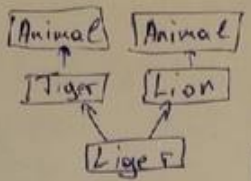
```
class IAnimal{
public:
    string get_name() {};
    virtual void sound = 0; - вот эта ф-я является абстрактной.
};
```

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 {
    ...
};
```

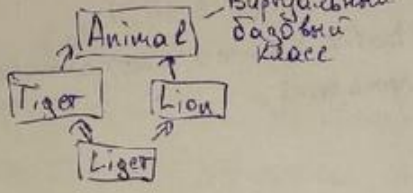


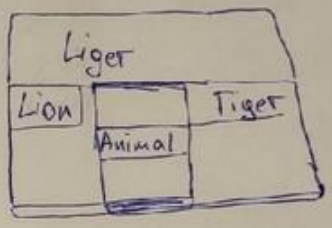
Возникают проблемы:
1) Два экземпляра Animal внутри Liger
2) Два одинаковых поля tail_length

Нужно делать так, чтобы не было переопределения, т.е. не должно быть одинаковых полей и методов у непосредственных родителей (и вообще родителей).

Решение:

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





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 Graph будут видны поля класса Node. (т.е. это почти это 1 класс)

~~Friend~~ 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*> neighbors;
    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 вызвать методы объекта, однако потом объект пропадает.