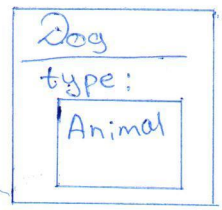
Наследование в ООП

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

При создании конструкторов неявно вызывается конструктор родителя (без аргументов)

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

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




```
int main() {
  Dog Bobik("Bobik");
  Dog noname;
}
```

Вызванные конструкторы:
Animal();
Dog("Bobik");

Если надо вызвать конкретный конструктор:

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

Ограничители доступа:

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

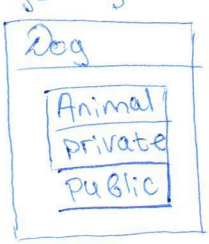
Изменение доступа при наследовании

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

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

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

Визуализация доступа:



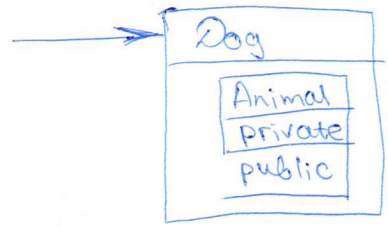
Можно в наследнике переписывать уровень доступа (если поля нет, то и переписать нельзя, т.е. private не пустится):

```
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:

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

Приведение типов



Присваиваем объекту класса Animal
объект класса Dog

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

```
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
```

Такое приведение — понижающее.

```
Void sound_three_times (Animal & creature) {
   creature.sound();
   creature.sound(); // 3 раза вызовется sound()
   creature.sound();    из Animal
}
```

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

```
Dog Bobik ("Bobik");
Animal * creature = &Animal;
(dynamic_cast < Dog* > (creature)).sound(); // вызовется sound() из класса Dog.
```

Такое приведение — повышающее

Виртуальные функции

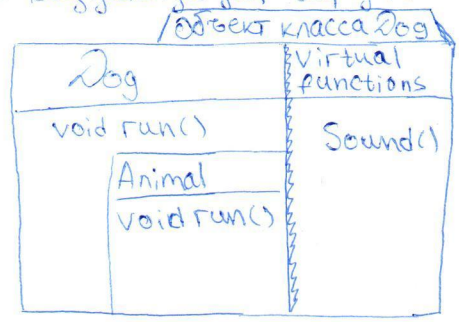
```
class Animal {
 public:
   string name;
 public:
   Animal(string new_name = "") : name(new_name){};
   sound();
}
class Dog : public Animal {
 public:
   string type;
 public:
   Dog(string new_name = "");
   sound();
}
```

```
Void sound_three_times(Animal & creature){
   creature.sound()
   creature.sound()
   creature.sound()
}
```

Если дать передать в функцию объект класса Dog, то вызовется sound() для Animal.
Чтобы вызвалась функция из класса Dog нужно объявить функцию виртуальной

```
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 new_name = " ");
  virtual sound() override
  void run();
}
```

- с virtual методу, вызываемый в ф-ии void sound_three_times(Animal &сосатые) зависит от того, какой объект будет передан в качестве параметра.
- Визуализация виртуальных ф-ий:




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

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

- Если функция была определена как виртуальная то она должна быть виртуальной у всех потомков
- Виртуальные ф-ии — позднее связывание
- Override — говорит, что метод переопределен (во избеж. ошибок).
- Деструкторы должны быть виртуальными если есть наследование.
- final — запрещает в дальнейшем наследовать класс и переопределять функцию.

Абстрактные классы

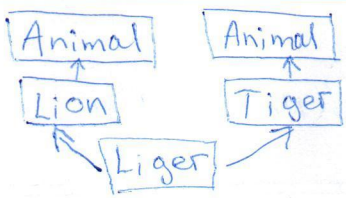
Идея интерфейса: пишется, что класс должен делать (методы, переменные), в виде кода — код работает (компилируется), но без фактической реализации.

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

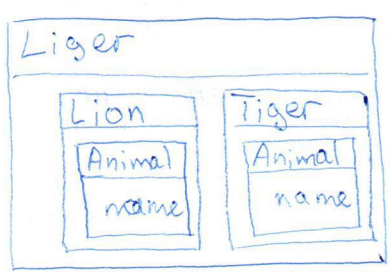
- Абстрактный класс — класс, у которого есть хотя бы одна абстрактная функция
- Объекты абстрактного класса не создаются (ошибка компиляции).

```
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 Tiger, public Lion {
...
}
```



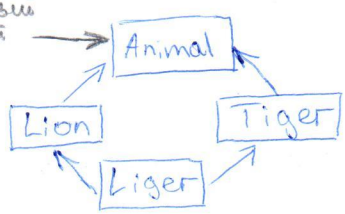
Проблемы:

1. 2 экземпляра Animal внутри класса
2. 2 одинаковых поля tail_length

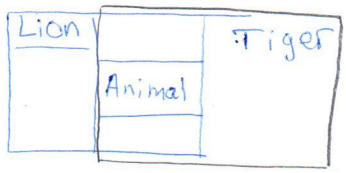
Нужно делать так, чтобы не было переопределений, то есть не должно быть одинаковых полей и методов у непосредственных родителей (и вообще родителей)

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

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

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

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

виртуальный базовый класс



Liger



Дружественные функции и классы

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

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

- 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, то есть из него доступны private поля Node

Анонимные объекты

```
Dog("Bobik"); // создание анонимного объекта
Dog bobik = Dog("Bobik");
Dog("Bobik").sound(); // можно вызвать методы объекта, но потом объект пропадает
```