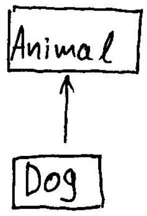
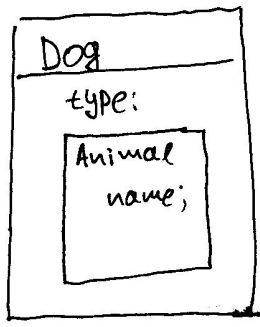
Наследование в ООП

```
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(0);
}
```

конструкторы:
Animal();
Dog("bobik");

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

⇓

если есть наследование, нужно создавать конструкторы БЕЗ АРГУМЕНТОВ.

Если необходимо 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

Наследование

Class Dog : public Animal {

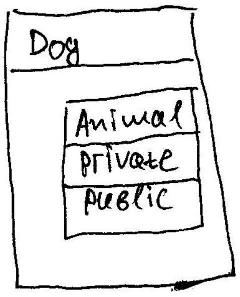
Public => public
Protected => protected
Private => недоступен
	изменение доступа при наследовании

Class Dog : private Animal {

Public => private
protected => private
private => недоступен
	Изменение доступа при наследовании

Class Dog : protected Animal {

Public => protected
Protected => protected
Private => недоступен
	Изменение доступа при наследовании.



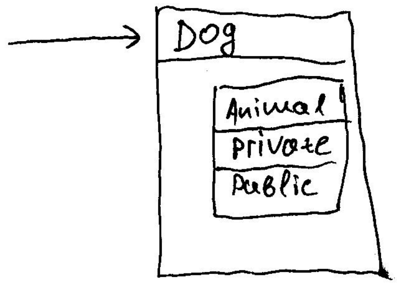
Можно в наследии переписать уровень доступа
(Если поля нет, то и переписать нельзя, т.е. private)

```
Class Dog: public Animal {
  public:
    string type;
  public:
    Dog (string new_name = " ");
  protected:
    using Animal::sound; // переменили 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.
  }
```



```
Dog bobik ("Bobik");
  Animal creature ("X305");
    creature = bobik; // в переменную creature скопируется часть
                        переменной bobik, которая относится к Animal
  Dog bobik ("Bobik");
  Animal *creature01 = &bobik;
  Animal &creature02 = bobik;
     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(); // вызов sound() три раза из Animal
     creature.sound();
     creature.sound();
  }
```

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

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

"Понижающее приведение типов"   вызывается sound() из класса Dog

```
[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();
}
```

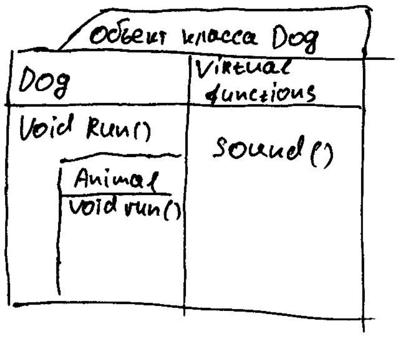
```
Void sound_three_times(Animal & creature)
   creature. sound();
   creature . sound ();
   creature . sound();
}
```

Dog => sound для Animal

Как сделать так, чтобы вызывалась функция из класса Dog?
Нужно объявить ф-ю 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 ();
```



! Если функция определена как виртуальная, то она должна быть виртуальной у всех потомков.

Виртуальные функции это позднее связывание

! Деструкторы должны быть виртуальными, если есть наследование

final - запрещает в дальнейшем наследовать класс и переопределять функцию

```
Virtual sound() override final;
```

дальше переопределять ее нельзя.

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

Нельзя создавать наследников класса Dog.

Абстрактные классы:

Идея интерфейса: вы пишете, что класс должен делать (методы, атрибуты) в виде кода — код рабочий (компилируется), но без фактической реализации

```
Class IAnimal {
  Public:
    String get_name() {};
    virtual void sound() = ∅; // Функция 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 {
  ...
}
```

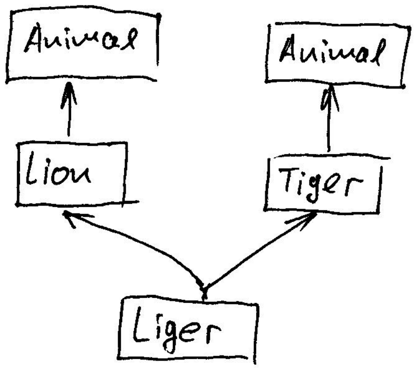
Множественное наследование:

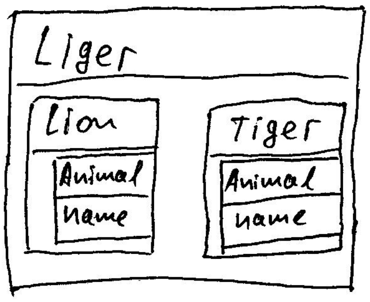
```
Class Animal {
  String name;
}

Class Tiger : public Animal {
  int tail_length;
}

Class Lion : public Animal {
  int tail_length;
}

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





Проблемы

1) Два элемента 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 {
  ...
}
```

$$\begin{tikzcd}
 & \boxed{\text{Animal}} & & \text{“виртуальный базовый класс”} \arrow[ll] \\
\boxed{\text{Lion}} \arrow[ur] & & \boxed{\text{Tiger}} \arrow[ul] & \\
 & \boxed{\text{Liger}} \arrow[ul] \arrow[ur] & &
\end{tikzcd}$$

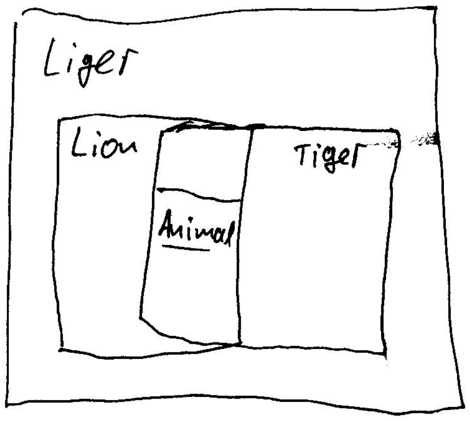
```
Class Animal {
   String name;
   Int  tail_length;
}
```

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

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

```
Class Liger : Public Lion, Public Tiger {

}
```



<u>Дружественные функции и классы</u>

```
class Graph {
 Private:
   Node* root;
 Public
   Node * Search (Node *node);
}
class Node {
 Private
   Void *data;
   Std:: list <Node*> neighbors;
   friend class Graph; // теперь из класса Graph будут видны поля класса 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, т.е. из него доступны private поля Node.

Анонимные объекты

```
Dog ("Bobik"); // создание анонимного объекта
Dog bobik = Dog ("Bobik");
Dog ("Bobik").sound();
```

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. Понижающее приведение типов (к родителю)
6. Обрезка объектов (animal = dog);
7. virtual методы (virtual деструктор)
8. абстрактный класс (интерфейсы)
9. Множественное наследование (виртуальные классы)
10. Дружественные методы и дружественные классы