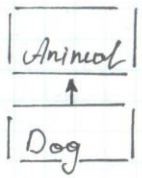
Наследование в ООП

Пусть есть класс Animal:

```
class Animal {
public:
    string name;
public:
    Animal ();
    sound ();
}
```



Мы хотим создать классы, которые будут конкретизировать животных, например, класс Dog, Cat и т.д.

```
// класс Dog:
// он наследует св-ва и методы
// класса Animal

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

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

// создается объект Bobic класса Dog;
при его создании вызывается
конструктор Dog (string text);
НО у нас нет такого конструктора
=> ОШИБКА

Как сделать правильно ? :

```
class Animal {
  public :
    string name;
  public :
    Animal ();
    sound ();
}
```

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

Позволяет сделать из одного конструктора два

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

bobik такой, noname такой

Уменьшим объём кода.

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

Теперь конструктор Animal инициализирует string name = new_name;

§1. Порядок вызова конструкторов

Добавим в Dog породу собаки:

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

Как теперь выглядит класс Dog:

Т.е когда создаём объект класса Dog, у нас создаётся объект класса Animal и объект класса Dog.

=> Два объекта, слитых друг с другом

Несостыковка!

Когда ~~вызываем~~ создаём объект класса Animal должен быть вызван конструктор Animal. НО когда создаётся объект класса Dog, не видно, чтобы вызвался конструктор Animal.

=> На самом деле этот конструктор вызывается, но неявно.

При создании объекта неявно вызывается конструктор родителя (который по умолчанию, т.е без аргументов)

Создаём Бобика!
// Вызван конструктор Dog, но перед его запуском вызван конструктор Animal.
т.е Animal → Dog → создан объект Бобик

```
конструкторы:
    Animal();
    Dog("Bobic");
```

=> Все ~~конструкторы~~ объекты будут корректно инициализированы.

=> Если есть наследование то необходимо создавать конструктор без аргументов

=> Но если нужно вызвать конкретный конструктор родителя, то можно сделать так:

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

Здесь new_name - св-во родительского класса (Animal) => => его лучше инициализировать в конструкторе Animal(new_name).

Так можно управлять цепочкой конструкторов, которая создаёт наш объект.

Это может пригодиться особенно если наследуется ряд переменных, которые нужно инициализировать.

§2 Модификаторы доступа

Ограничители доступа:

public - ф. методы и св-ва доступны снаружи класса

protected - — и —

private - доступны только изнутри класса

// Лучше всего использовать private

Наследование: (Изменение доступа при наследовании)

class Dog : public Animal {}

[public => public
protected => protected
private => недоступен

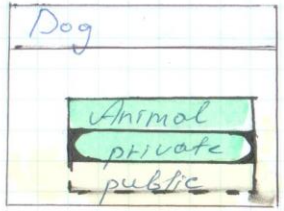
class Dog : private Animal {}

[public => private
protected => private
private => недоступен

class Dog : protected Animal {}

[public => protected
protected => protected
private => недоступен

Пример:



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

На примере Dog:
(Хотим сделать sound() => protected)

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

// делаем sound(), определённый в // Animal -> protected.

Переписан ур. доступа на protected в Dog к методу sound();

§3 Сокрытие или удаление лишнего функционала:

```
// Хотим удалить ф-ю sound()

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

// Скрыли (удалили) метод из
// класса Dog.
// (В Animal он всё еще доступен)
```

§4 Создадим объекты класса Animal и Dog:

(Например в main())

```
int main() {
    Dog Bobik("Bobik");
    Animal creature("X305");
    creature = Bobik; }
```

В переменную creature скопируется часть переменной Bobik, которая относится к Animal.

§5 Повышающее приведение типов (к наследнику)

Создадим ссылки:

```
Dog Bobik("Bobik");
Animal * creature01 = &Bobik;
// создан указатель на класс Animal;
// Ещё присвоен указатель на класс Bobik.
// копирования не происходит, т.к это
// указатель
Animal & creature02 = Bobik;
// создаётся псевдоним (ссылка!),
// который указывает на Bobik.
//
// Вызовем sound()
• Bobik.sound();
• creature01 -> sound();
• creature02.sound();
```

Вопрос: какой sound() вызовется?

В 1-м случае вызовется sound из Dog;

Во 2-м - sound из Animal

В 3-м - ——— и ———

=> Это нужно, когда мы хотим работать с объектами, но при этом можем захотеть вызвать ф-ионал родителя, а не наследника

Пример! Хотим написать ф-цию, которая будет одинаково действовать для всех объектов типа

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

Теперь в эту ф-ю можно передать объект не только Animal, но и любой другой объект, который наследовался Animal.

§6 Понижающее приведение типов

Можно также идти вниз по наследованию.

> Можно представить указатель класса Animal, как указатель на класс Dog (если он действительно указывает на объект класса Dog.

Для этого используется конструкция с dynamic_cast.

Пример:

```
(dynamic_cast<Dog*>(creature)).sound(){
...}
```

В контексте:

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

(вызывается sound() из Dog

Это "понижающее приведение типов."

§7 Virtual методы (virtual деструктор)

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

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

(Мы переопределили ф-ю sound

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

Если передадим сюда объект класса Dog, то все равно будет вызван sound для Animal (произойдёт приведение типов)

Но как сделать так, чтобы здесь вызывалась ф-я из класса Dog? (мы же её переопределили)

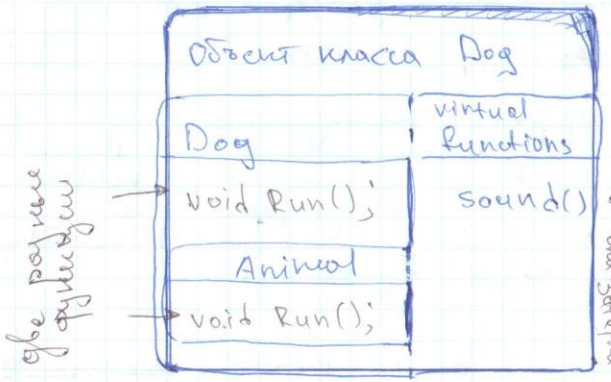
⇒ Нужно объявить её виртуальной:

```
class Animal {
public:
    string name;
public:
    Animal(...):...;
    virtual sound();
    void Run();
};
```

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

⇒ Ф-ия передаётся; она одна для всех классов

Добавим какую-нибудь ф-ю, чтобы лучше видеть различие. Например ф-ю Run()



Если теперь в

```
void sound_three_times(Animal &creature){}
```

подать на вход Dog, то будет использована Dog-овая sound().

Если мы определили ф-ю как виртуальную, то она должна быть виртуальной у всех потомков.

Виртуальные ф-ии – это **позднее связывание**.

Деструкторы должны быть виртуальными, если есть наслед-е

Без виртуального деструктора:

- Конструктор родителя
- ↓
- Конструктор наследника
- ↓
- ⋮
- ↓
- Деструктор наследника
- Деструктор родителя

} Может привести к ошибке

⇒ Лучше всегда делать деструктор виртуальным.

Важно, чтобы у virtual ф-ий были одинаковые параметры и возвращаемое значение, иначе это будут две разные ф-ии

Чтобы избежать ошибки, можно использовать override:

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

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

Можно также использовать слово **final**, для запрета дальнейшего переопределения ф-ии, ими наследования класса

Пример:

- virtual sound() override **final**

  ↑ ↑

  больше переопределять её нельзя => дальше во всех наследующих наследниках будет использоваться эта версия sound()

- class Dog **final**: pubic Animal {...}

  ↑

  Нельзя создавать наследников класса Dog.

API

§8. Абстрактные классы (интерфейсы)

Идея интерфейса: пишем, что класс должен делать (методы, переменные, в виде кода — код рабочий (компилируется), но без фактической реализации методов

(Для работы в команде)

Это реализуется через виртуальные классы.

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

(показывает, что это интерфейс)

Здесь ф-я sound является абстрактной. Это применяется только для virtual ф-ий, т.к. их нельзя опр. заранее.

Абстрактный класс - тот, у которого есть хотябы одна абстрактная ф-ия.

У абстрактного класса нельзя создавать объекты (ошибка компиляции)

If в наследнике абстрактные ф-ии не переопределены, то он тоже абстрактный.

=> затем создаём

class Animal : public IAnimal {...}

=> и переписываем в ней ф-ии

В интерфейсе переменные обычно не нужны.

§9 Множественное наследование (виртуальные классы)

Создадим несколько классов:

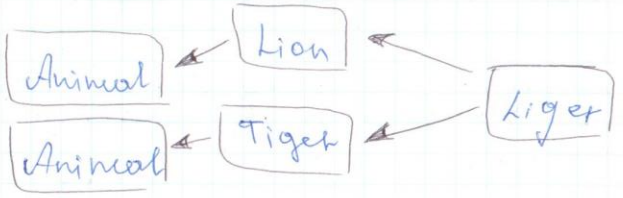
```
class Animal {
    string name;
};
```

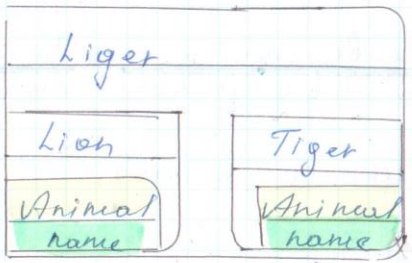
```
class Tiger : public Animal {
    int tail_length; !
};
```

```
class Lion : public Animal {
    int tail_length; !
};
```

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

Схема наследования:





Проблемы:

1. Два экземпляра Animal внутри класса

=> Проблема синхронизации данных

2. Два одинаковых поля tail_length

При множественном наследовании нужно избегать неопределённости, т.е не должно быть одинаковых полей и методов у непосредств-х родителей (и вообще у родителей).

Если наследуем интерфейс, то всё хорошо, т.к при этом не возникает конфликтов.

Решение:

X Плохое: сделать поля Node public

V Хорошее: сделать Node дружественным к классу Graph

Теперь из класса Graph видны поля и методы класса Node.

friend – односторонняя дружба

Node is friend of Graph, но не наоборот

Из Node private методы класса Graph недоступны!

] A - friend of B |=> A - не друг B
] B - friend of C

Можно также сделать дружественным не весь класс, а отдельный метод