29.11.2021г.

Наследование в ООП

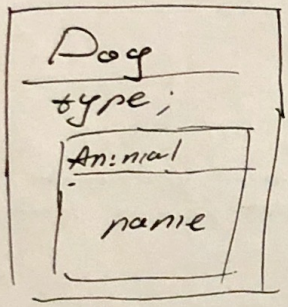
```
class Animal {
public:
    string name;
public:
    Animal();
    sound();
    //Animal(string new_name=""); name(new_name){};
};
```

Можно написать в таком формате, что будет значение инициализации поля name значением new_name

```
class Dog : public Animal {          // Класс Dog наследует
public:                              // класс Animal.
    string type;
    //Dog();
    sound();
    Dog(string new_name="");
};
```

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

При (вызове) создании обоих объектов класса Dog может быть вызван один конструктор, если в нем определено значение имени по умолч.



При создании (одного) объекта класса Dog неявно вызывается конструктор класса Animal (конструктор родителя) по умолчанию

При наличии наследования обязательно необходимо создавать конструктор без аргументов (конструктор по умолчанию)

Если необходимо вызвать конкретный конструктор родителя:

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

Ограничители доступа

public - доступен снаружи класса
protected - доступен классам, которые наследуют
private - доступен только внутри

Наследование

```
Class Dog: public Animal {
    public    => public
    protected => protected
    private   =>
```
] Доступ при наследовании public

```
Class Dog: private Animal {
    public    => private
    protected => private
    private   =>
```
] Доступ при наследовании private

```
Class Dog : protected Animal {
    public    => protected
    protected => protected
    private   =>
```
] Доступ при наследовании protected

Можно в наследнике переписать уровень доступа (private недоступен => изменение уровня метода)

Пример:

```
protected:
    using Animal::sound;
```
| Переписали уровень доступа и sound

Сокрытие или удаление лишнего функционала

```
public:
    Dog(string new_name);
    sound() = delete;
```
| Скрыли метод из класса Dog

```
Dog bobik("bobik");
Animal creature("Хвост");
creature = bobik; // В creature скопируется часть об
                  // bobik, которая относится к Animal
```

```
Dog bobik("Bobik");
Animal *creature1 = &bobik; // Указатель
Animal &creature2 = bobik; // Ссылочный
    bobic.sound(); // Вызовется sound() из Dog
    creature1 -> sound(); // Вызов из Animal
    creature2.sound(); // Вызов из Animal
```

Можно представить ук-ель класса Animal как указатель на класс Dog

dynamic_cast<Dog*>(creature)

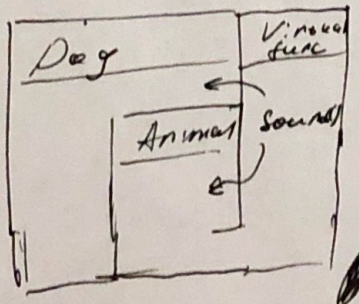
"Понижающее приведение типа"

Виртуальные ф-ии:

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

/* При передаче в качестве параметра объекта Dog если sound относится ф-ии вызовется sound() от Animal
Решение: объявление виртуальной ф-ии */

Виртуальные ф-ии объявляются виртуальными во всех родительских классах */



Виртуальные ф-ии создаются один раз для класса (передаются ссылкой), в то время как невиртуальные ф-ии могут быть несколько

Деструкторы должны быть виртуальными если есть наследование

Ключевое слово OVERRIDE проверяет параметры
final - запрещает дальнейшее переопределение наследов.

Абстрактные классы:

Идея интерфейса заключается в описании класса всех функций без реализации самих методов

Данная идея реализуется через виртуальные классы

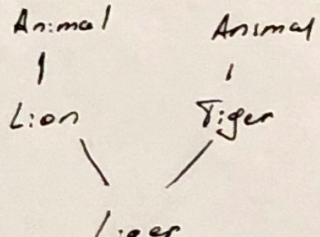
```
class Animal {
    private:
        string name;
        int age;
    public:
        string get_name() {}
        virtual void sound() = 0; // абстрактная фун
};
```

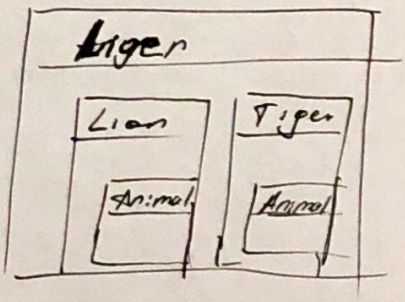
Абстрактный класс - класс у которого есть хотя бы одна абстрактная ф-ия.

Создание объекта абстрактного класса приведет к ошибке компиляции

Множественное наследование:

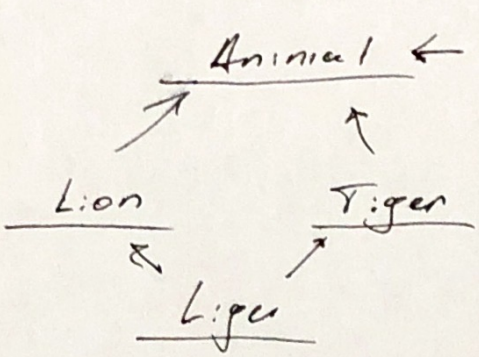
```
Class Animal {
    ...
};
Class Tiger : virtual public Animal {
    int tail_length;
};
class Lion : virtual public Animal {
    int tail_length;
};
class Liger : public Lion, public Tiger {
};
```





Problems: • Два экземпляра Animal
• Два поля tail_length

Нужно устать так, чтобы при наследовании не создавалось неоднозначности в следствии наличия одинаковых полей, методов



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;
};
```

// Для того, чтобы в классе Graph были видны поля и методы класса Node мы делаем его дружественным классу Node

// friend - одностороннее отношение

Анонимные объекты:

```
Dog ("Bobik"); // Создание анонимного объекта
Dog ("Bobik").sound();
```

Анонимные объекты не хранятся в памяти.