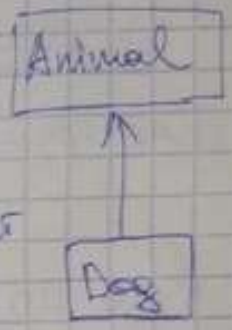
Лекция (29.11.21)

Наследование в ООП (об.-ориент. программирование).

Наследование позволяет создать новый класс на основе уже существующего, при этом свойства и функциональность существующего класса заимствуются новым классом.

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



получается, что класс Dog наследует класс Animal

```
class Dog : public Animal {
  public:
    Dog ( ); → Dog (string new-name = " ");
    Sound ( );
}
```

Теперь когда мы перейдём к main:

```
int main ( ) {
  Dog Bobik ("Bobik"); // Т.к мы исп. конструктор, кот. на вход получает строку, только конструктора Dog нас нет → делаем будет вызван конструктор // Dog ("Bobik");
  Dog кошка; // здесь будет вызван // конструктор Dog ( );
}
```

Можно сделать:

Animal (string new-name = " "): name (new-name) { }; - значит наш конструктор инициализирует поля именами этих аргументов

Добавим к классу Dog:

public: string type;

Когда мы создаем класс Dog у него есть части, которые относятся непосредственно к нему и часть унаследованная от класса Animal. Формально создаются два объекта, но они как бы друг с другом слиты. Хотя мы вызываем только конструктор Dog неявно вызывается и конструктор Animal

! При создании объекта неявно вызывается конструктор родителя (который по умолчанию) без аргументов

Конструкторы: Сначала будет вызван Animal();
Потом Dog("Бобик").

Если наследование больше, то схема та же. Потому все объекты будут инициализированы корректно.

Если есть наследование, то обязательно создавать конструктор без аргументов.

Но если хотим вызвать конкретный конструктор, то:

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

Т.к. изначально имя отн. к классу Animal, то логично, чтобы этот тип и оставался

Ограничители доступа:

public; – все методы доступны снаружи объекта;
protected; – доступны снаружи класса
private; – доступ только изнутри класса.

Наследование

1) public Animal → Class Dog : public Animal {

public ⟹ public (переходит)
protected ⟹ protected
private ⟹ private (недоступен)

} изменение доступа при наследовании

Если мы хотим сделать наследование private:

2) Class Dog : private Animal {

public ⟹ private
protected ⟹ private
private ⟹ недоступен

} изм. д-па при наслед.

3) Protected:

Class Dog : protected Animal {

public ⟹ protected
protected ⟹ protected
private ⟹ недоступен

} изм. д-па при наслед.

В связи с этим если в class Animal сделать private: string name; (вместо public), то в class Dog мы не сможем его менять.

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

Если из класса Dog хотим обращаться к объектам класса Animal — это нужно предусмотреть, поместив их в public или protected.

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

Посмотрим код — сделаем Sound() из public → protected:

```
class Dog : public Animal{
  public:
      string type;
  public:
      Dog (string new_name=" ");
  protected:
    using Animal:: Sound();//Изменили уровень доступа к
                           Sound() на protected в Dog
  }
```

Сокрытие или удаление лишнего функционала:

```
class Dog: public Animal{
  public:
      string type;
  public:
    Dog (string new_name =" ");
    Sound() = delete; //скрыли (удалили) метод из класса
                      Dog, но в Animal он все еще доступен
  }
```

Наследование позволяет избежать дублирования кода. Код не должен дублироваться. Пример с кнопкой (круглые, квадратные и т.д.). Ранее мы делали архитектуры программы функциональный т.е. все объекты были в ф-циях), в ООП все разбивается на классы (объекты) и программа описывается взаимодействием объектов (вызов друг друга и наследство). Этот подход позволяет проще создавать более сложные и функциональные программы. Позволяет лучше структурировать программу. Сделать архитектуру более понятной

и прозрачной. Когда большое кол-во объектов - жестко разграничивать их между собой. => Избежать ошибки.

(Есть нескл. ООЯзыков, например, Java.)

ООП - методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из кот. является экземпляром опред. класса, а классы образуют иерархию наследования. Помогает избежать сложности кода.


```
Dog Bobik ("Bobik");
Animal creature ("Хозя");
```

Можем написать

```
creature = Bobik; // переменную creature скопируется часть переменной Bobik, кот. относится к Animal; это м.б. удобно;
```

Создание ссылок:

```
Dog Bobik ("Bobik");
Animal *creature01 = &Bobik;
Animal &creature02 = Bobik;

Bobik.sound(); //вызовется sound из Dog;
creature01 -> sound(); // вызывает sound из Animal;
creature02.sound(); // из Animal
```

Если хотим сделать, чтоб вызывалось сразу, можем прописать только для родителя

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

Можно представить указатель класса Animal как указатель на класс Dog (если он действительно указывает на объект класса Dog). Это делается с помощью конструкции:

```
(dynamic_cast < Dog* > (creature)).sound();
```

указатель на объект.

будет вызван sound() из класса Dog

Это наз. "конструкция приведения типов".

Можем сделать так, чтобы в класс Dog обращались именно к Animal, а из Animal к Dog.

Виртуальная ф-ция

```
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(); // если бы ф-ции Sound() здесь не было, то компилятор нашел бы sound() по иерархии вверх, т.е. в Animal, но мы ее переопределили.
}
```

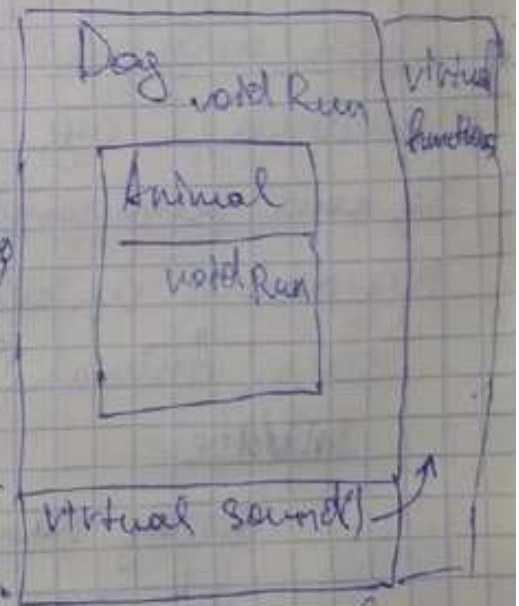
```
void sound_three_times(Animal & creature){
    creature.sound();
    creature.sound();
    creature.sound();
}
```

Dog => sound для Animal
Надо сделать, чтобы для Dog
↓
Нужно объявить эту ф-цию виртуальной.

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

Virtual будет означать что ф-ция перезапишется на самом деле, т.е. если объявили ф-цию виртуальной то тогда эта ф-ция станет одна для всех классов → будет общая

Виртуальные ф-ции общие для всех вложенных классов.



Если определили ф-цию sound виртуальную то она д.б. такой везде, у всех потомков.

Виртуальные ф-ции это поздние связывания, они работают немного медленнее, чем обычные.

Деструкторы должны быть виртуальными. Деструктор выполняет освобождение используемых объектом ресурсов и удаление динамических переменных. Если наследование → виртуальные.

Если делать отдельные → могут быть нарушения в логике работы с памятью.

Если в Dog записать virtual sound (int N), то мы не переопред. ф-цию, а создадим новую.

Но если написать override, тогда компилятор найдет, что мы переопр. нужную ф-цию

Теперь если добавить int N компилятор сообщит об ошибке, ведь на самом деле ф-ция непереопределена.

Ключ. слово final – запрещает в дальнейшем наследовать класс или переопред. ф-цию.

virtual sound () override final;

↑ больше переопред. нельзя

class Dog final: public Animal {
...
};

↑ Нельзя создавать наследников Dog

Абстрактные классы:

Когда много людей пишут код вместе, что нужно сделать – договориться как должен выходить код — интерфейс.

Идея интерфейса: писать что класс должен делать (методы, переменные), в виде кода – код рабочий (можно компилировать), но без реал. реализации методов. Тогда все видят, что должно представлять из себя программа. Это реал. через вирт. классы.

class Animal {
private
    string name;
    int age;
public:
    string get_name(); ← если так, то её нужно реализовывать
    virtual void going() = 0; // это значит, что ф-ция абстрактная

}

Абстрактный класс — класс, в кот. есть хотя бы одна абстр. ф-ция. Заготовка для наслед. класса. Наследники тоже абстрактные. Объекты абстр. класса не создаются (ошибка компиляции).

Абстр. ф-ция только вирт. ф-ция.

Абстр. класс удобен для создания интерфейсов.

```
class Animal : public IAnimal {
  ...
};
```

В интерфейсе обычно описывают только методы (ф-ции).

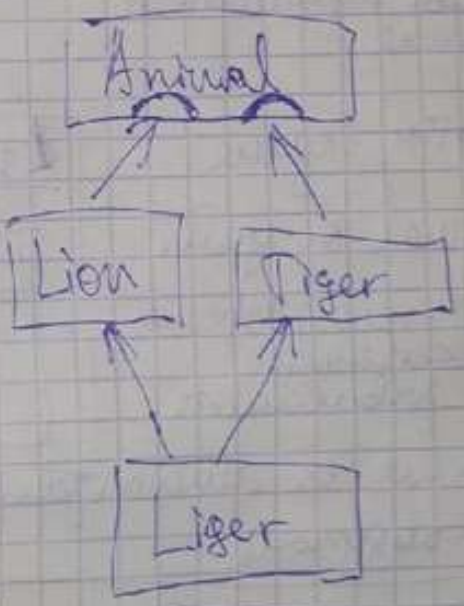
Множественное наследование: Создадим несколько классов:

```
class Animal {
  ...
};

class Tiger : public Animal {
  int tail_length
};

class Lion : public Animal {
  int tail_length;
};

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



Проблемы:

① Два экземпляра Animal внутри класса

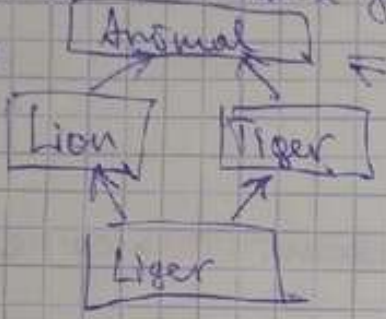
② Два одинаковых поля tail_length

При множественном наследовании нужно делать так, чтобы не было неоднозначности, т.е. не должно быть одинаковых полей и методов у непосредственных родителей. ← и вообще родителей

Если захотим вызвать ф-цию Sound(), то пойдет вверх по иерархии и не совсем понятно что найдет.

Для этого удобно использовать интерфейс. В примере можно создать класс хищных зверюшек и туда поместить хвост.

Итак:



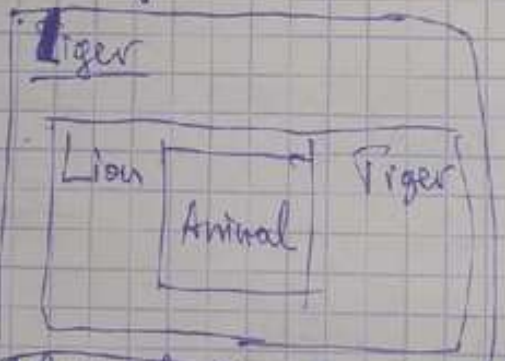
← "виртуальный базовый класс"

Для этого:

class Tiger: virtual public Animal;
...

class Lion: virtual public Animal;

Тогда tail_length можно перенести в Animal



В этом случае Animal общий.

Так можно разрешать множественную зависимость.

Но лучше её не использовать вообще. Или очень хорошо перед этим подумать.

В реальности обычно:

UML — правила записи таких схем.

Дружественные ф-ции и классы:

Чем больше зависимостей, тем плохой код.

```
class Graph {
private:
    Node* Root;
public:
Node* Search(Node* node);
};
```

```
class Node {
private:
    void * data;
    std::list<Node*> neighbors;
friend class Graph
};
```

(2)

// Мы не можем реализовать Search, т.к. не сможем обратиться к Node, т.к. private

Решение: ① Исправить private на public → плохой вариант

② Дружественные классы (мы как бы реализуем одно класса внутри класса)

Теперь из class Graph будут видны поля и методы 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.

Такие если A friend of B, B friend of C. A не friend C. (Не транзитивно!)

Работать такими нужно аккуратно, т.к. меняем область видимости.

Такие можно делать дружественным класс отдельный метод:

```
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 => из него доступны private члены Node

Много использовать friend не стоит — негативно влияет на надежность.

Анонимные объекты:

Dog("Bobik"); — создание анонимного объекта, он создаётся, но не будет привязан ни одной переменной

Dog Bobik = Dog("Bobik");

↑ скопируется

Dog("Bobik").Sound(); - можно вызывать ф-цию.

Помогает избегать создания лишних переменных.

Рассмотрим:

Наследование: ① Модификатор доступа при наследовании (public, private, protect)

② Порядок вызова конструкторов при наследовании;

③ Пор. вызова деструкторов при наслед.;

④ Новым. поведение типов (к наследнику);

⑤ Полим. поведение типов (к родителям);

⑥ Обрезка объектов (Animal = Dog);

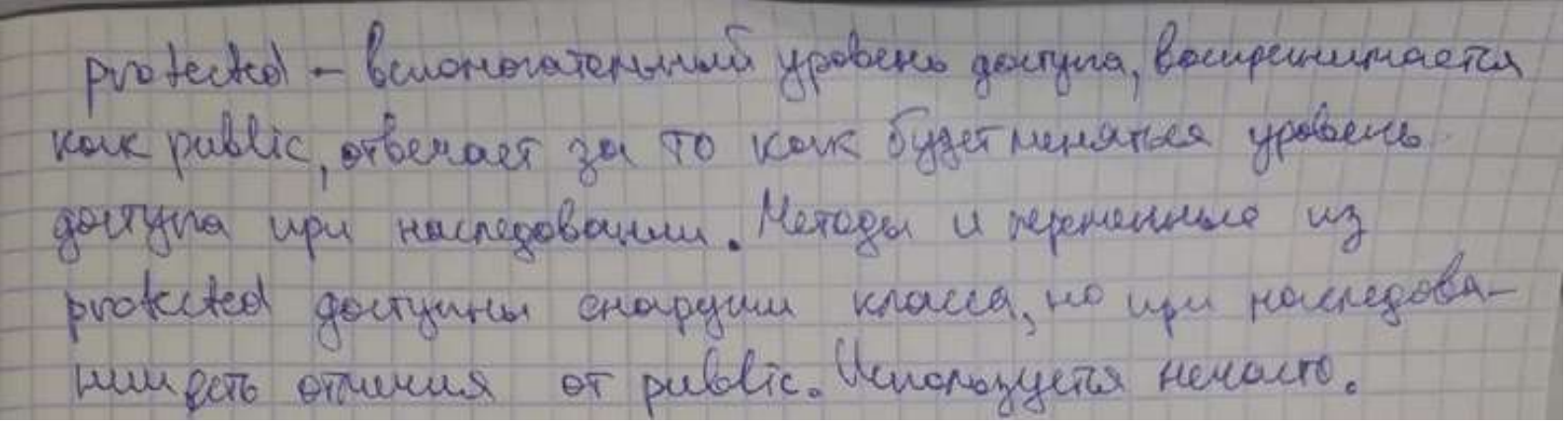
⑦ Virtual методы (virtual деструктор);

⑧ Абстрактный класс (интерфейсы);

⑨ Множественные наслед-я (вирт. классы);

⑩ Друж. методы и классы;

⑪ Анонимные объекты.



Пример наследования:

```
#include <iostream>
#include <string>
class Person{
public:
    std::string name;
    int age;
    Human(std::string n_name = "", int n_age = 0)
        : name(n_name), age(n_age){};
    std::string getName() const { return name;
    int getAge() const { return age;
};
class BasketballPlayer : public Person{
public:
    double gameAverage;
    int points;
    BasketballPlayer(double n_gameAverage = 0.0, int n_points = 0)
      : gameAverage(n_gameAverage), points(n_points){};
};
```