Констр-ры в C++ (чтобы инициализировать поля).

В C++ много чего делается неявно

```
class Matrix {
private:
    int height;
    int width;
    double *data;
public:
```

Констр-ры:

① Если не написан констр-р, то будет создан констр-р по умолчанию
Matrix() {}

② неявно синтезированный констр-р по умолч.
Matrix() {}
Нужен, чтобы проинициализировать все поля. Вызывается при создании объектов массивов.

③ Констр-р с параметрами.
Matrix(int height, int width);

Matrix A; // будет вызван констр-р по умолчанию (неявный).

Matrix A(5,5); // будет вызван

Замечание.
Если написан констр-р с параметрами, то констр-р по умолчанию неявно не будет.
Matrix A; // Error!

Если объект создан, но его конструктор отсутствует.

пример конструктора. Конструкторов с параметрами м.б. много

```
Matrix (int height, int width){
this->height = height;
this->width = width;
data = malloc (height * width * sizeof(double));
}
```

//this - указатель на объект (текущий)

Работу с памятью сразу в конструкторе.

Выделение памяти удобней всего делать в конструкторах.

Освобождать память нужно в деструкторе.

При удалении объекта вызывается деструктор

```
class Matrix {
...
~Matrix () { free(data); } //деструктор
```

будет вызван только, когда не будет объекта

```
double trace ( Matrix A){...};
trace (A); // в ф-ию trace через конструктор копирования передается копия объекта A.
```

Конструктор копирования - вместо этого объекта вызывается конструктор по умолчанию, как пример конструктора объекта.

Когда нужно скоп-ть объект, то у этого класса д.б. констр-р копир-я!

Если он не задан, то будет происходить побитовое коп-е объекта (метод класса) ↙ в кач-ве арг объект по ссылке

```
Matrix(const Matrix & m) { // вызов в создании копии.
  height = m.height;   ← для созд. объекта аналогичного
  width = m.width;       (для копии)
  data = malloc(height * width * sizeof(double));
  memcopy(..); }.
```

Когда нужно ск-ть объект, класса д.б. констр. коп-я. (иначе побитовое коп-е) → будет создан новый объект.

Когда выз-ся констр-р копир-я (неявно):

1) Matrix A; // создать объект из др. объекта
   Matrix B(A); // вызов констр-р копирования.
   передать в ф-ю A
2) trace(A); // вызов констр-р копир-я
3) Return A; // вызов констр-р коп-я. вернуть зн-е из ф-ции.

A = B; // по умолч. происх побитовое присвоение. не констр-р коп-я! не создано новый объект.

[illegible]

Matrix B=A; // присвоение

Matrix B(A); // констр-е

Перегрузка операций: (наоб масса)

class Matrix {

...

...b

A=B

Matrix & operator = (const Matrix &m){

m ~ B

this ~ A

width = m.width;

height = m.height;

free(data);

data = malloc(...);

mem copy(...);

return * this;

}

← перегрузка операции для присвоения.

A=B; // вызывается перегрузка операции

! перегрузку у операций надо делать аккуратно. и перегр операций типа +, - стараться избегать.

* Важно:

делать пр-ку на A=A; самоприсв-е

if(*m == this) return * this

Констр-ры в. б. public, но если хотим запретить какие-то операции, то надо в.

сооб констр-ю private.

* enum будут написаны IDE.

```
class LinearEquation { // Ax=B
    Matrix A;
    Matrix B;
}
LinearEquation eq01; // будет вызван констр-р по умолчанию LinearEquation и вызваны констр-ры по умолч для полей: A, B. Писать констр-р по умолч-ю!

A = B + C;
```

создан объект Atemp = B+C
поскольку после вык-я Atemp удаляется, то память не удалять, а перенести в A

```
A = B+C; // констр-р переимен-я. Вызов
A = B; // операции присваивания.

class Matrix {
    ...
    ...
    Matrix (Matrix && m) {
        width = m.width;
        height = m.height;
        data = m.data;
        m.data = nullptr; // при вызове данных констр не будет
    }
```

! Если констр-р перем не задан, то будет исп-н констр-р копий → полиморфность!

setlocale (LC_ALL, "ru");

- - - - - - - - - - - -

Перегрузка ф-й:

Можем реализовать совершенно разные ф-ции, но все они будут иметь одно и то же имя!

```
int sum(int a, int b){}
int sum (int a, int b, int c){}
double sum (d a, d b, d c){}
```

- - - - - - - - - - - -

Перегр-ка констр класса:

Мы можем инициализировать класс разн спос-ами (как удобнее). (Полиморфизм). Разное поведение в зав-ти от разных ситуаций.

- - - - - - - - - - - -

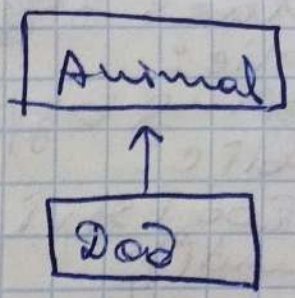
Деструктор класса (при разруш объекта класса) (только 1) (правильным образом освоб-ть ресурсы, кот выделились в классе).

~ Имя класса () {}

Дестр-р не имеет пар-ры!

(Вызыв-ся, когда, например, объект исчезает из зоны видимости)

Наследование в ООП.




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

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

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

! При создании объектов явно вызывается конструктор родителя.
: констр-ра (нет по умолч-ю).

```
Animal();
Dog("bobik");
```

⇒ Если есть наслед, то дочер-ю созд-ть констр-р без аргументов.
вызвать
Но если хочу конкретный констр-р, то

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

Ограничи достъпа:

public - достъпна отвсякъде
protected - достъпна само вътре в класа
private - достъпна само вътре в класа

Наследяване

class Dog : public Animal {

public => public
protected => protected
private => недостъпен.

} запазване достъпа при наслед-ие.

class Dog : private Animal {

public => private
protected => private
private => недостъпен

}

class Dog : protected Animal {

public => protected
protected => protected
private => недостъпен

}

Можно в наслед. переписать уровень доступа. (если поля нет, то и переписать нельзя, т. е private не переписыв-ся).

если в Animal: sound()
public: хочу в dog
protected sound()

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

protected:
  using Animal:: sound; // переписали на protected ур-не доступа в Dog у нас. sound();
}
```

Сокрытие или удаление лишнего функц-ла:

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

```
Dog bobik{"bobik"};
Animal creature("хэз");
creature=bobik; // в переменную creature сохран-ся часть переменной bobik, кот. относится к Animal.

Dog bobik("bobik");
Animal *creature01=&bobik; // указ-ль
Animal &creature02= bobik; // ссылка
bobik.sound(); // вызов-ся Sound из Dog.
creature01->sound(); // вызов-ся sound из An.
creature02.sound(); // вызов sound из Anim

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

! Можно предоставить указатель на класс Animal как указатель на класс Dog (если он действительно указывает на объект класса Dog).

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

вызовется sound() из класса Dog

"Полиморфное приведение"

```
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();
}
```

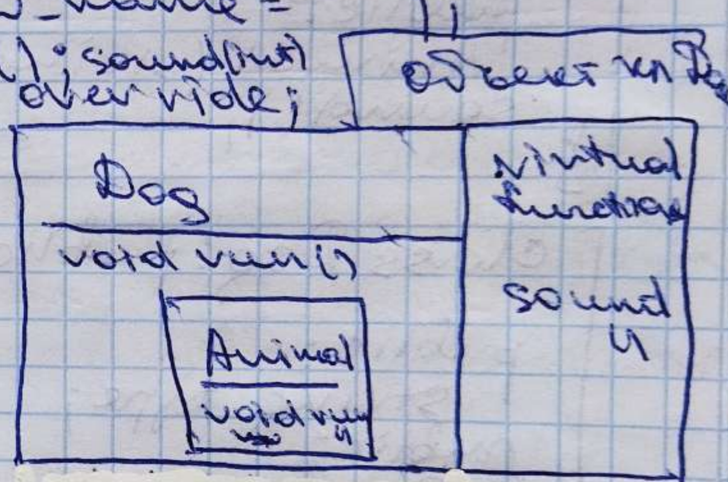
Dog => sound для Animal.

Как сделать так, чтобы вызывалась ф-я из класса Dog?

Нужно объявить эту 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() : sound(int) override;
void run();
}
```



Вирт ф-я - 1 для нашего объекта

! Если определили ф-ю как вирт-ю, то она д.б. виртуальной у всех потомков

Виртуальная ф-я это позднее связ-е

! Дестр-ры д.б. виртуальными, если есть наслед-е

final - запрещает в дальнейшем наследовать класс и переопр. ф-ю;

virtual sound () override final;

↑ ↗ дальше переопр. ее нельзя.

```
class Dog final: public Animal {
...
};
```
↑ нельзя создать-ть наследников класса Dog.

Абстрактные классы:

Идея интерфейса: нашему что класс должен делать (методы, переменные), но в виде кода - код работает (компилир-ся), но без одной реализации.

```
class IAnimal {
public: private:
    string name;
    int age;
public:
    string get_name() { };
    virtual void sound() = 0; // ф-я явл. абстрактной
};
```

Абстр. класс - класс у кот есть хотя бы 1 абстр. ф-я.

Объекты абстрактно не создаются. (ошибка компиляции).

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

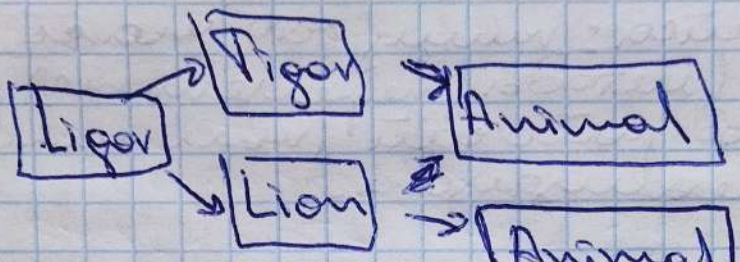
Множественное наслед-е:

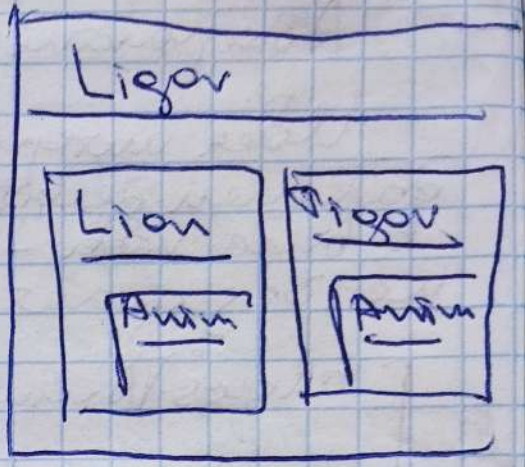
```
class Animal {
...
};
```

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

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

```
class Ligov: public Lion, public Tigov {
}
```





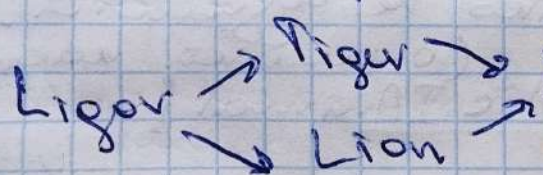
Проблемы:

① Два экземпляра Animal внутри класса.

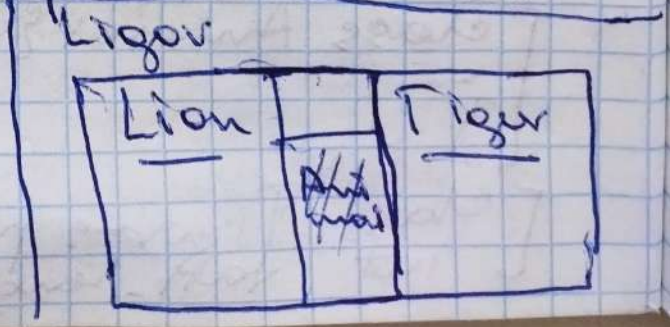
② Два одинаковых поля tail_length.

—

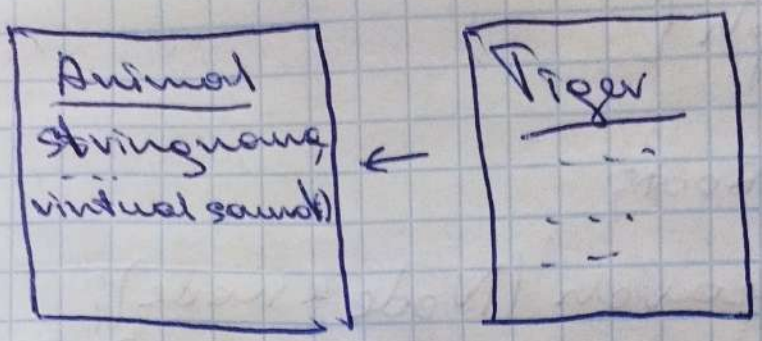
Нужно делать так, чтобы не было повторов, т.е. не 2 одинаковых полей и методов у непосредственных родителей. (и вообще у родителей)



class Lion: virtual public Animal.



UML



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; // теперь в class Graph будут видны поля и методы class Node.
};
```

friend - дружба одност. т.е. это почти как в жизни.

Node is friend of Graph.

но не наоборот. в Node private методы и поля Graph недост-ны

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 другого класса Node, т.е. нужен доступ к private полям Node.

Анонимные объекты.

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

Конс-е:

① Модификаторы доступа при наслед.
   public
   private
   protected

② Порядок вызова констр-ов при наслед

③ Порядок вызова деструкт при наслед.

④ Правила приведения типов (к наследнику)
⑤ Понимание приведения типов (к родителю)
⑥ Определение объектов (animal = dog);
⑦ virtual методы (virtual деструктор)
⑧ абстрактный класс (интерфейсы).
⑨ Множественное наследование (внутренние классы)
⑩ Дружественные методы и дружественные классы
⑪ Анонимные объекты