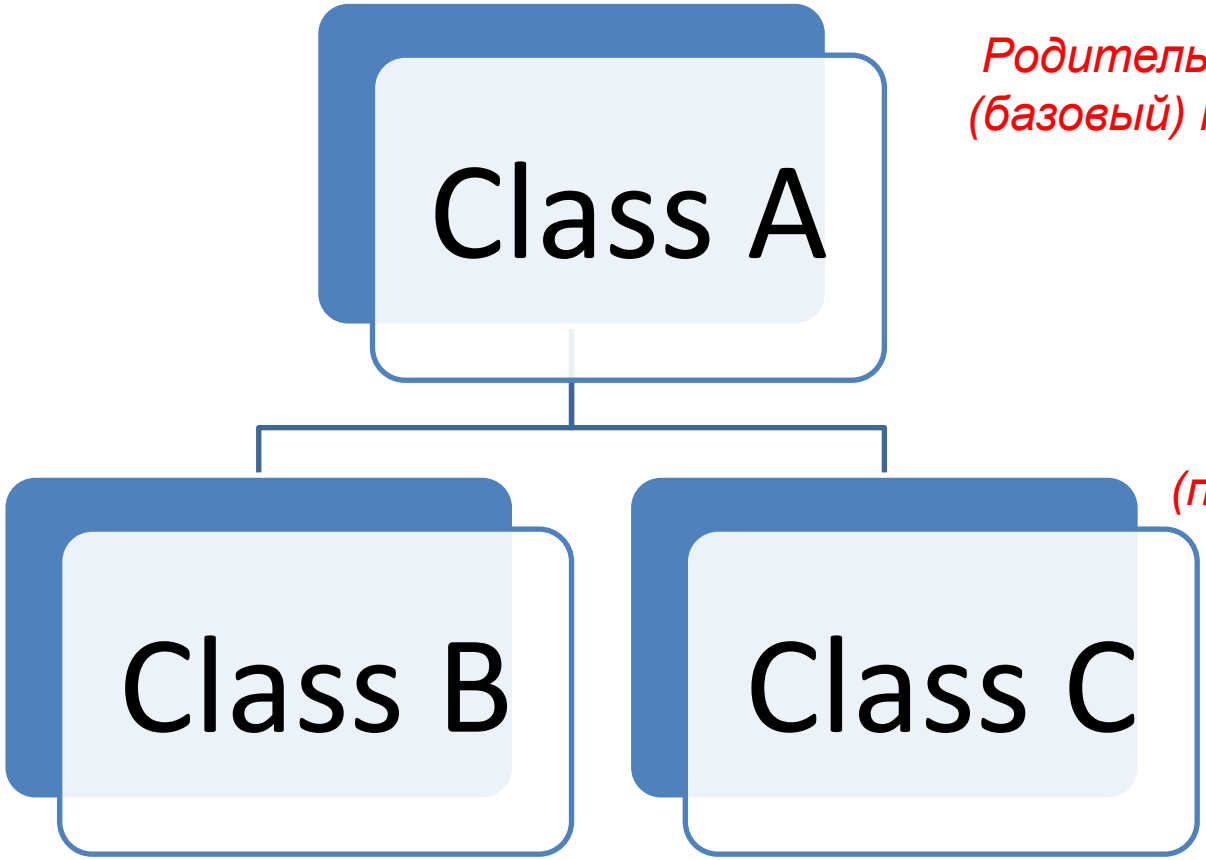
Наследование

28 июня 2017 г.

Наследование



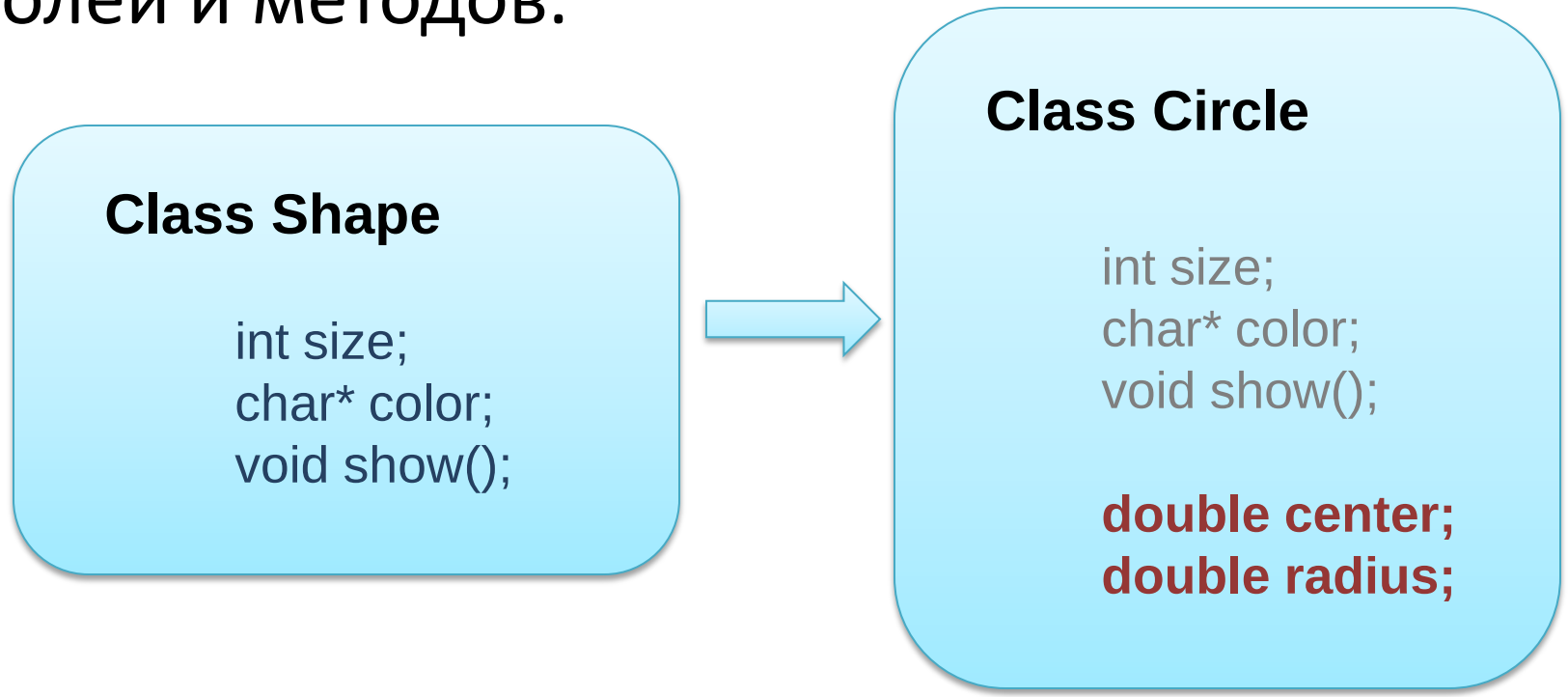
Родительский (базовый) класс

Дочерние (производные) классы

Иерархия классов

Наследование

Производные классы получают «по наследству» данные и методы своих базовых классов и расширяют их функциональность за счет своих полей и методов.



Наследование: синтаксис

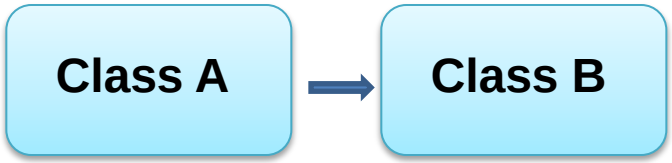
```
class A
{
public:
    A();
    virtual ~A();
};

class B : public A
{
public:
    B();
    ~B();
};
```

Конструкторы и деструкторы

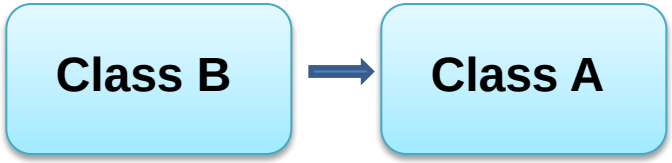
1. Конструктор базового класса всегда вызывается и выполняется **до** конструктора производного класса.

2. Деструктор базового класса выполняется **после** деструктора производного класса.

Конструкторы:



Деструкторы:



Конструкторы и деструкторы

В каком порядке будут вызваны конструкторы и деструкторы объектов?

```
class Shape{ };
class Rectangle : public Shape{ };

void init() {
      Rectangle r;
}
…
init();
```

Конструктор с параметрами в родительском классе

```
class A
{
public:
      A(int x);
};

class B: public A
{
public:
      B(int x, int y);
};
```

```
B::B(int x, int y) :
          A(x)
{
          …
}
```

```
/*
В листе инициализации
вызывается конструктор
базового класса с параметрами
*/
```

Что будет выведено при вызове f()?

```
class Base {
public:
    Base() { std::cout << "Class Base - constr" << std::endl; }
    ~Base() { std::cout << "Class Base - destr" << std::endl; }
};

class Derived : public Base {
public:
    Derived( ) { std::cout << "Class Derived - constr" << std::endl; }
    ~Derived( ) { std::cout << "Class Derived - destr" << std::endl; }
};

void f( ){
    Base bs;
    Derived dr;
}
```

Типы наследования по спецификаторам доступа

- открытое (public)
- защищенное (protected)
- закрытое (private)

Типы наследования

Доступ в базовом классе	Доступ в производном классе		
	спецификатор наследования		
	private	*protected*	*public*
private	недоступно	недоступно	недоступно
protected	private	protected	protected
public	private	protected	public

Наследование (доступ)

```
class Derived : (public | protected | private) Base { };
```

- по умолчанию в классах используется закрытое (private) наследование
- по умолчанию в структурах используется открытое (public) наследование
- хотя все унаследованные private-поля недоступны непосредственно в объектах производного класса, но содержатся в них

Наследование (пример)

```
class Base{
public:
    void a() { std::cout << "Base" << std::endl; }
};

class Derived : public Base {
public:
    void b() { std::cout << "Derived" << std::endl; }
};

void f() {
    Derived dr;
    dr.a();   // ОК
    dr.b();   // ОК
}
```

Наследование (пример)

```
class Base{
public:
    void a() { std::cout << "Base" << std::endl; }
};

class Derived : protected Base {
public:
    void b() { std::cout << "Derived" << std::endl; }
};

void f() {
    Derived dr;
    dr.a();   // Error
    dr.b();   // OK
}
```

Какие функции Base::f доступны?

```
class Base{
    void f1() { std::cout << "Base::f1" << std::endl; }
protected:
    void f2() { std::cout << "Base::f2" << std::endl; }
public:
    void f3() { std::cout << "Base::f3" << std::endl; }
};

class Derived : private Base {
public:
    void g() {
        std::cout << "Derived::g" << std::endl;
        f1();
        f2();
        f3();
    }
};
```

```
void f() {
    Derived dr;
    dr.f1();
    dr.f2();
    dr.f3();
}
```

Какие функции Base::f доступны?

```
class Base{
    void f1() { std::cout << "Base::f1" << std::endl; }
protected:
    void f2() { std::cout << "Base::f2" << std::endl; }
public:
    void f3() { std::cout << "Base::f3" << std::endl; }
};

class Derived : protected Base {
public:
    void g() {
        std::cout << "Derived::g" << std::endl;
        f1();
        f2();
        f3();
    }
};
```

```
void f() {
    Derived dr;
    dr.f1();
    dr.f2();
    dr.f3();
}
```

Какие функции Base::f доступны?

```
class Base{
    void f1() { std::cout << "Base::f1" << std::endl; }
protected:
    void f2() { std::cout << "Base::f2" << std::endl; }
public:
    void f3() { std::cout << "Base::f3" << std::endl; }
};

class Derived : public Base {
public:
    void g() {
        std::cout << "Derived::g" << std::endl;
        f1();
        f2();
        f3();
    }
};
```

```
void f() {
    Derived dr;
    dr.f1();
    dr.f2();
    dr.f3();
}
```

Указатель на базовый класс

- указатель на объект базового класса может указывать на объект производного класса (наоборот – нельзя)
- позволяет создавать массивы, в которых есть элементы и базового, и производного классов

```
class Base { … };
class Derived : Base { … };

Derived derObject;
Base* pBaseObject = &derObject;  // OK
Base* pBaseObj = new Derived;    // OK
```

Указатели на базовый класс (пример)

```
class Cat {
public:
    void say() { std::cout << "Meow!" << std::endl; }
};
class Kitten : public Cat {
public:
    void say() { std::cout << "Mimimi!" << std::endl; }
};
void f() {
    Cat* myPets[3];
    myPets[0] = new Cat;
    myPets[1] = new Kitten;
    myPets[2] = new Kitten;
    …
};
```

```
void g() {
    for (int i = 0; i < 3; ++i){
            myPets[i]->say();
     }
}
// Все говорят “Meow!”
```

Виртуальный метод

- метод, который можно переопределить в наследуемых классах так, что конкретная реализация выбирается при выполнении кода
- задается в базовом классе через ключевое слово virtual, в производном – можно override
- virtual действует до конца цепочки наследования

```
class Cat {
    …
    virtual void say();
};
```

```
class Kitten : public Cat {
    …
    void say() override;
};
```

Виртуальный метод (пример)

```
class Cat {
public:
   virtual void say() { std::cout << "Meow!" << std::endl; }
};
class Kitten : public Cat {
public:
   void say() override { std::cout << "Mimimi!" << std::endl; }
};
void f() {
   Cat* myPets[3];
   myPets[0] = new Cat;
   myPets[1] = new Kitten;
   myPets[2] = new Kitten;
   …
};
```

Ключевое слово override

- указывает, что метод переопределен

```
class Cat {
public:
    virtual void say() { std::cout << "Meow!" << std::endl; }
};
class Kitten : public Cat {
public:
    void say(const std::string& str) override {    // Error
        std::cout << str << std::endl;
    }
    void say() override {
        std::cout << "Mimimi!" << std::endl;      // OK
    }
};
```

Ключевое слово final

- указывает, что метод нельзя дальше переопределить или что от класса нельзя наследовать

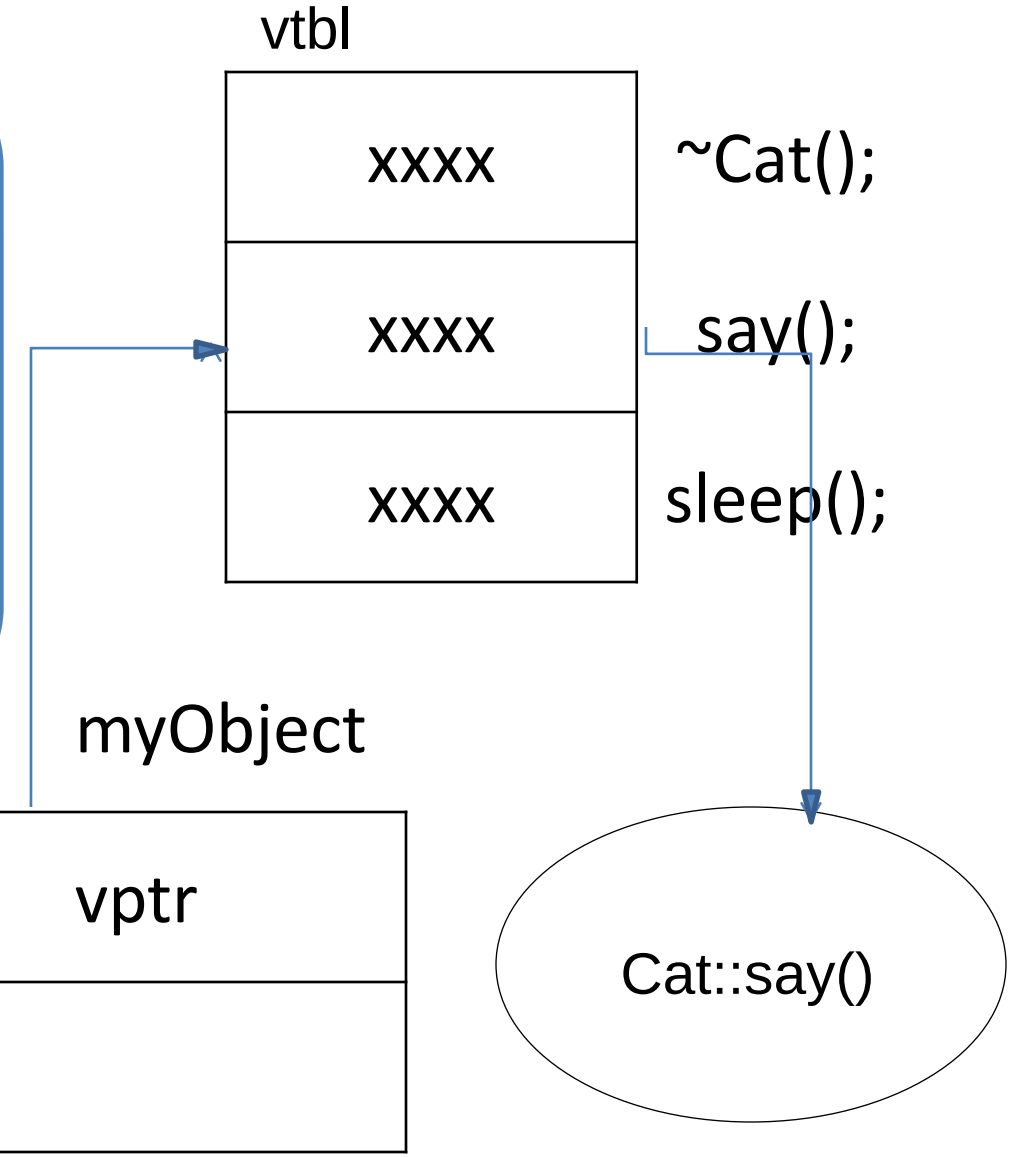
```
class Cat {
public:
    virtual void say() final { std::cout << "Meow!" << std::endl; }
};
class Kitten : public Cat {
public:
    void say() override {    // Error
        std::cout << "Mimimi!" << std::endl;
    }
};
```

Таблица виртуальных методов

- создается для каждого класса, если в нем есть хотя бы один виртуальный метод
- каждый объект класса содержит указатель на таблицу виртуальных методов для своего класса
- в таблице содержится массив указателей на виртуальные методы (функции)
- вызов виртуального метода происходит через указатель на таблицу и на метод в ней

Таблица виртуальных методов

```
class Cat {
public:
    virtual ~Cat();
    virtual void say();
    virtual int sleep();
};
```



Виртуальные методы: уточнение

- при вызове виртуальных методов в конструкторах и деструкторах виртуальность не работает, потому что таблица виртуальных функций либо еще не создана, либо уже разрушена
- при наследовании деструкторы должны быть виртуальными

Что будет выведено при вызове f()?

```
class Base {
public:
    Base() { std::cout << "Class Base - constr" << std::endl; }
    ~Base() { std::cout << "Class Base - destr" << std::endl; }
};

class Derived : public Base {
public:
    Derived( ) { std::cout << "Class Derived - constr" << std::endl; }
    ~Derived( ) { std::cout << "Class Derived - destr" << std::endl; }
};

void f( ){
    Base* bs = new Derived;
    delete bs;
}
```

Что будет выведено при вызове f()?

```
class Base {
public:
    Base() { std::cout << "Class Base - constr" << std::endl; }
    virtual ~Base() { std::cout << "Class Base - destr" << std::endl; }
};

class Derived : public Base {
public:
    Derived( ) { std::cout << "Class Derived - constr" << std::endl; }
    ~Derived( ) { std::cout << "Class Derived - destr" << std::endl; }
};

void f( ){
    Base* bs = new Derived;
    delete bs;
}
```

Наследование: деструкторы

Не забывайте добавлять перед деструктором базового класса ключевое слово virtual

Связывание

Объединение объекта и вызова метода

Раннее

- происходит на этапе компиляции
- учитывается точный тип объекта
- более быстрое, требует меньше памяти

Позднее

- происходит при выполнении программы
- достигается через виртуальные методы и наследование
- более медленное, но более гибкое

Раннее связывание (пример)

```
class Cat {
public:
   void say() { std::cout << "Meow!" << std::endl; }
};
class Dog {
public:
   void say() { std::cout << "Arrrgh!" << std::endl; }
};
void f() {
   Dog angryDog;
   angryDog.say();     // раннее связывание
};
```

Позднее связывание (пример)

```
class Cat {
public:
   virtual void say() { std::cout << "Meow!" << std::endl; }
};
class Kitten : public Cat {
public:
   void say() { std::cout << "Mimimi!" << std::endl; }
};

void f() {
   Cat* myPet = new Kitten;    // позднее связывание
   myPet->say();
   …
};
```

Абстрактный класс

- класс, в котором хотя бы один метод определен как чисто виртуальный, т. е. не имеет тела (= 0)
- нельзя создать объект абстрактного класса
- если в дочернем классе чисто виртуальный метод из родительского класса не имеет реализации, то дочерний класс тоже будет абстрактным

```
class Shape {
    virtual float getArea() = 0;
};
```

Чисто виртуальный деструктор

- нужен тогда, когда мы хотим сделать класс абстрактным, но в нем нет подходящих методов
- объявляется с ключевым словом virtual
- должен иметь пустую реализацию

```
class AbstractClass {
    AbstractClass() { }
    virtual ~AbstractClass() = 0;
};
AbstractClass():: ~AbstractClass() { }
```

Вопросы?