Происхождение видов

Соловьёв Владимир Валерьевич
Huawei, НГУ, СУНЦ
vladimir.conwor@gmail.com
vk.com/conwor

Между типами “Студент” и “Преподаватель” много общего

Студент

Преподаватель

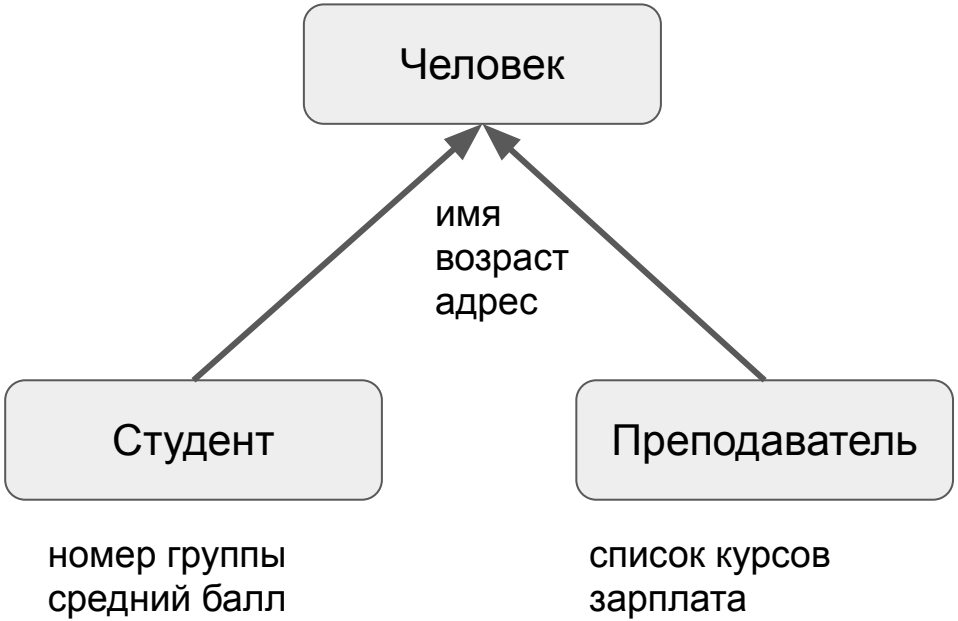
имя
возраст
адрес

Студент

номер группы
средний балл

Преподаватель

список курсов
зарплата

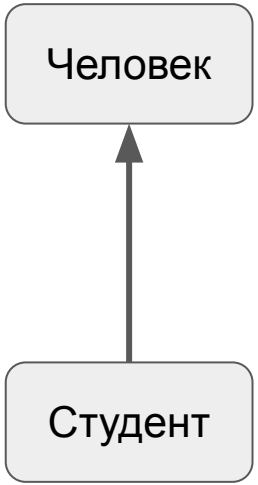

Человек
имя
возраст
адрес
Студент
Преподаватель
номер группы
средний балл
список курсов
зарплата


Человек
Студент
Преподаватель
Бакалавр
Магистрант
Аспирант
Семинарист
Лектор

Отношение наследования

1) Person - базовый класс (предок, суперкласс)

2) Student - производный класс (потомок, подкласс)



Отношение наследования

```
struct Person {
        char* name;
        int age;
};
```

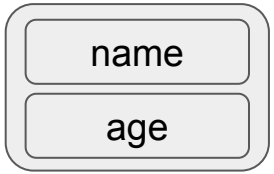
```
struct Student : public Person {
        float gpa;
};
```

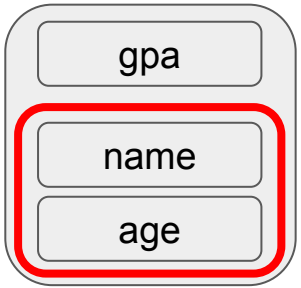
```
Student s;
s.name = “Dima”;
s.age = 18;
s.gpa = 4.75;
```

Отношение наследования

```
struct Person {
        char* name;
        int age;
};
```

```
struct Student : public Person {
        float gpa;
};
```

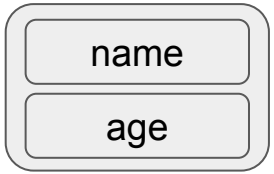


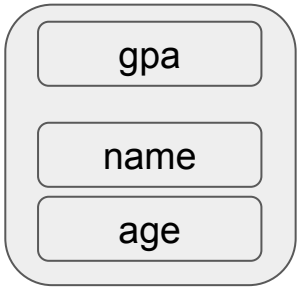


Отношение наследования

```
struct Person {
        char* name;
        int age;
};
```

```
struct Student : public Person {
        float gpa;
};
```





Отношение наследования

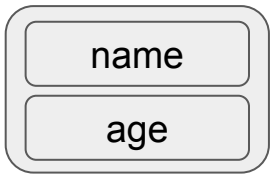
Можно представлять, что в объекте класса-наследника есть вложенный объект базового класса

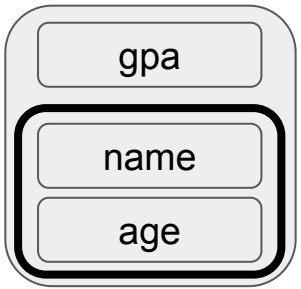
Но фактически, в него вкладываются поля базового класса на одном уровне со своими полями

Агрегирование (композиция)

```
struct Person {
        char* name;
        int age;
};
```

```
struct Student {
        Person personPart;
        float gpa;
};
```





Агрегирование (композиция)

```
struct Person {
        char* name;
        int age;
};
```

```
struct Student {
        Person personPart;
        float gpa;
};
```

```
Student s;
s.personPart.name = “Dima”;
s.personPart.age = 18;
s.gpa = 4.75;
```

Наследование и области видимости

```
struct Base {
private:
        int x;
public:
        int y;

        void foo() {
                this->x = 42;
                this->y = 37;
        }
};
```

```
struct Derived : public Base {
        void bar() {
                this->x = 42; // есть, но недоступно!
                this->y = 37;
        }
};
```

```
Base b;
b.x = 42;
b.y = 37;
```

```
Derived d;
d.x = 42;
d.y = 37;
```

Наследование и области видимости

```
struct Base {
private:
        int x;
public:
        int y;
protected:
        int z;

        void foo() {
                this->x = 42;
                this->y = 37;
                this->z = 28;
        }
};
```

```
struct Derived : public Base {
        void bar() {
                this->x = 42; // есть, но недоступно!
                this->y = 37;
                this->z = 28;
        }
};
```

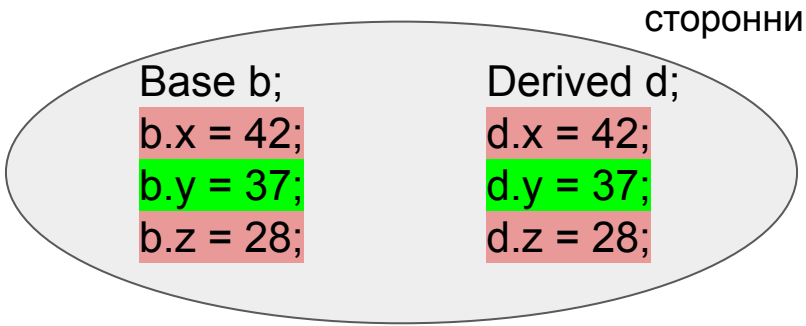
```
Base b;
b.x = 42;
b.y = 37;
b.z = 28;
```

```
Derived d;
d.x = 42;
d.y = 37;
d.z = 28;
```

Наследование и области видимости

```
struct Base {
private:
        int x;
public:
        int y;
protected:
        int z;

        void foo() {
                this->x = 42;
                this->y = 37;
                this->z = 28;
        }
};
```

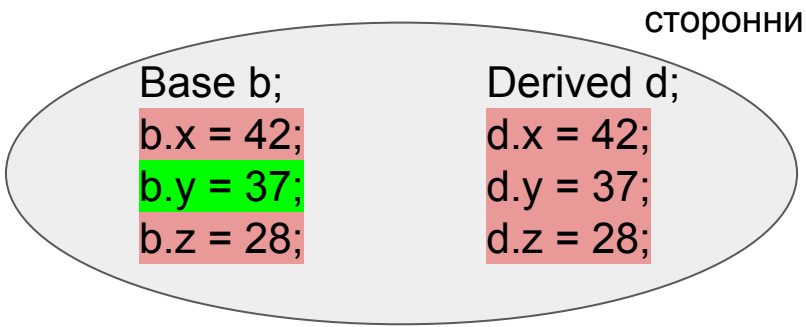
```
struct Derived : public Base {
        void bar() {
                this->x = 42; // есть, но недоступно!
                this->y = 37;
                this->z = 28;
        }
};
```



Наследование и области видимости

```
struct Base {
private:
        int x;
public:
        int y;
protected:
        int z;

        void foo() {
                this->x = 42;
                this->y = 37;
                this->z = 28;
        }
};
```

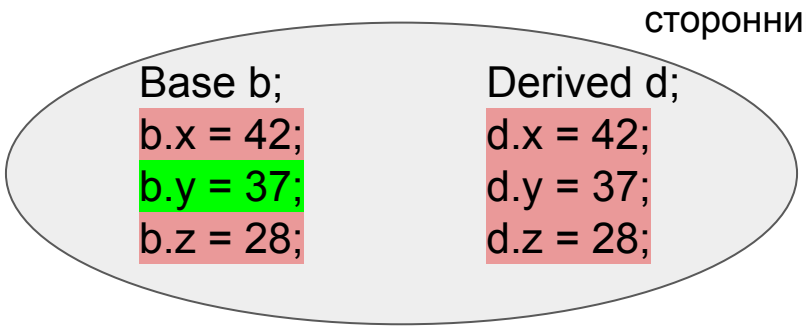
```
struct Derived : private Base {
        void bar() {
                this->x = 42; // есть, но недоступно!
                this->y = 37;
                this->z = 28;
        }
};
```



Наследование и области видимости

```
struct Base {
private:
        int x;
public:
        int y;
protected:
        int z;

        void foo() {
                this->x = 42;
                this->y = 37;
                this->z = 28;
        }
};
```

```
struct Derived : protected Base {
        void bar() {
                this->x = 42; // есть, но недоступно!
                this->y = 37;
                this->z = 28;
        }
};
```



Наследование и области видимости

```
struct Base {
private:
        int x;
public:
        int y;
protected:
        int z;

        void foo() {
                this->x = 42;
                this->y = 37;
                this->z = 28;
        }
};
```

```
struct Derived : protected Base {
        void bar() {
                this->x = 42; // есть, но недоступно!
                this->y = 37;
                this->z = 28;
        }
};
```

```
struct Somehting : public Derived {
        void baz() {
                this->x = 42;
                this->y = 37;
                this->z = 28;
        }
};
```

Наследование и области видимости

```
struct Base {
private:
        int x;
public:
        int y;
protected:
        int z;

        void foo() {
                this->x = 42;
                this->y = 37;
                this->z = 28;
        }
};
```

```
struct Derived : private Base {
        void bar() {
                this->x = 42; // есть, но недоступно!
                this->y = 37;
                this->z = 28;
        }
};
```

```
struct Somehting : public Derived {
        void baz() {
                this->x = 42;
                this->y = 37;
                this->z = 28;
        }
};
```

Наследование и области видимости

private < protected < public

Область видимости поля (метода) базового класса в классе-наследнике определяется, как минимум из:

1) Области видимости этого поля в базовом классе
2) Способа наследования

Наследование и области видимости

```
struct Base {
baseType:
        int x;
};
```

```
struct Derived : inheritanceType Base
{
// derivedType:
//      int x;
};
```

inheritanceType

baseType	public	protected	private
public	public	protected	private
protected	protected	protected	private
private	private*	private*	private*

* - недоступно даже в классе Derived

Конструкторы при наследовании

```
class Person {
        char* name;
        int age;
public:
        Person(char* n, int a) {
                this->name = n;
                this->age = a;
        }
};
```

```
class Student : public Person {
        float gpa;
public:
        Student(char* n, int a, float g) {
                this->name = n;
                this->age = a;
                this->gpa = g;
        }
};
```

Student s(“Dima”, 18, 4.75);

Конструкторы при наследовании

```
class Person {
        char* name;
        int age;
public:
        Person(char* n, int a) {
                this->name = n;
                this->age = a;
        }
};
```

```
class Student : public Person {
        float gpa;
public:
        Student(char* n, int a, float g) {
                this->name = n;
                this->age = a;
                this->gpa = g;
        }
};
```

Student s(“Dima”, 18, 4.75);

Конструкторы при наследовании

```
class Person {
protected:
        char* name;
        int age;
public:
        Person(char* n, int a) {
                this->name = n;
                this->age = a;
        }
};
```

```
class Student : public Person {
        float gpa;
public:
        Student(char* n, int a, float g) {
                this->name = n;
                this->age = a;
                this->gpa = g;
        }
};
```

Student s(“Dima”, 18, 4.75);

Конструкторы при наследовании

```
class Person {
protected:
        char* name;
        int age;
public:
        Person(char* n, int a) {
                this->name = n;
                this->age = a;
        }
};
```

```
class Student : public Person {
        float gpa;
public:
        Student(char* n, int a, float g) {
                this->name = n;
                this->age = a;
                this->gpa = g;
        }
};
```

Student s(“Dima”, 18, 4.75);

Конструктор класс-наследника обязан вызвать конструктор базового класса. Это может быть конструктор без параметров, но тогда он должен быть в базовом классе.

Конструкторы при наследовании

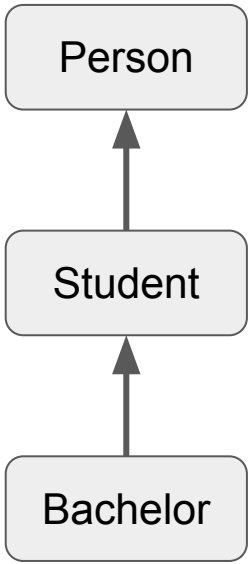
```
class Person {
        char* name;
        int age;
public:
        Person(char* n, int a) {
                this->name = n;
                this->age = a;
        }
};
```

```
class Student : public Person {
        float gpa;
public:
        Student(char* n, int a, float g):Person(n, a) {
                this->gpa = g;
        }
};
```

Student s(“Dima”, 18, 4.75);

Конструкторы при наследовании

Порядок конструирования - сначала базовый класс, потом наследник




```
Person(...) { printf(“Person\n”); }

Student(...): Person(...) { printf(“Student\n”); }

Bachelor(...): Student(...) { printf(“Bachelor\n”); }
```

```
Person
Student
Bachelor
```

Наследование методов

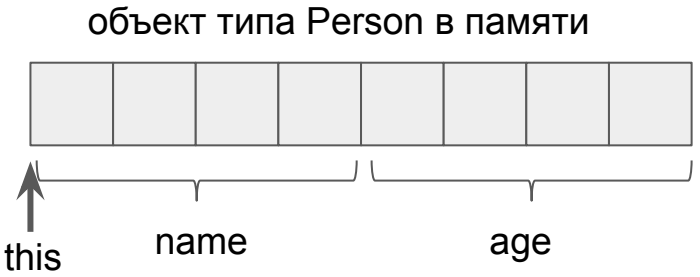
```
class Person {
        char* name;
        int age;
public:
        ...
        int getAge() {
                return this->age;
        }
};
```

```
class Student : public Person {
        float gpa;
public:
        Student(char* n, int a, float g);
};
```

```
Student s(“Dima”, 18, 4.75);
s.getAge();
```

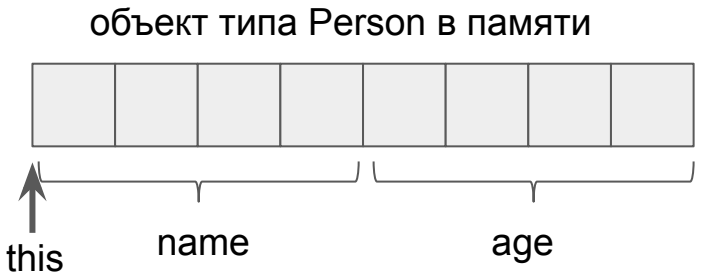
Наследование методов

```
int getAge() {
        return this->age;
}
```



Наследование методов

```
int getAge() {
        return this->age;
}
```

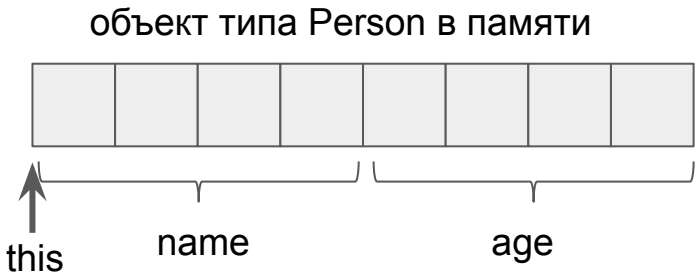


“поле” - это не больше, чем пара свойств:

1) тип
2) смещение относительно начала объекта (this)

Наследование методов

```
int getAge() {
        return this->age;
}
```




```
typedef char byte;

int getAge() {
        byte* ptr= (byte*) this;
        byte* ptrToAge = ptr + 4;
        int value = *((int*)ptrToAge);
        return value;
}
```

“поле” - это не больше, чем пара двух свойств:

1) тип
2) смещение относительно начала объекта (this)

Наследование методов

```
typedef char byte;

int getAge() {
        byte* ptr= (byte*) this;
        byte* ptrToAge = ptr + 4;
        int value = *((int*)ptrToAge);
        return value;
}
```

Этот код применим к любому массиву байтов, для которого мы полагаем, что по смещению 4 лежит “поле” age типа int.

Наследование методов

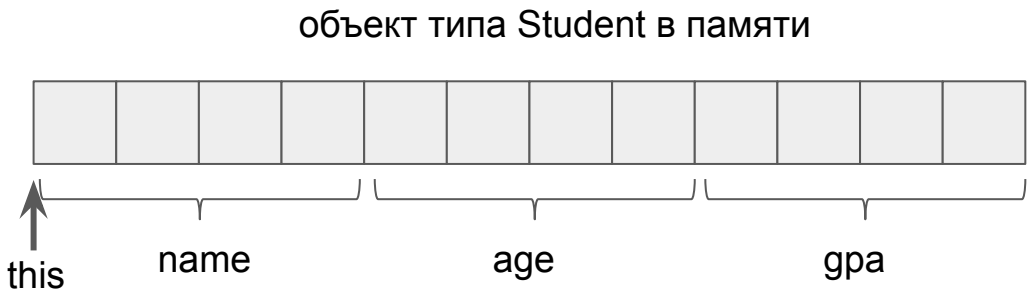
```
typedef char byte;

int getAge() {
        byte* ptr= (byte*) this;
        byte* ptrToAge = ptr + 4;
        int value = *((int*)ptrToAge);
        return value;
}
```

Этот код применим к любому массиву байтов, для которого мы полагаем, что по смещению 4 лежит “поле” age типа int.

Это и обеспечивается при наследовании

Наследование методов

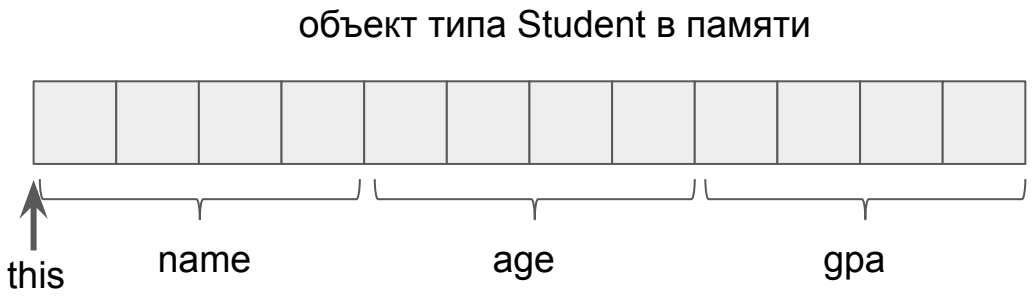
```
typedef char byte;

int getAge() {
        byte* ptr= (byte*) this;
        byte* ptrToAge = ptr + 4;
        int value = *((int*)ptrToAge);
        return value;
}
```



Наследование методов

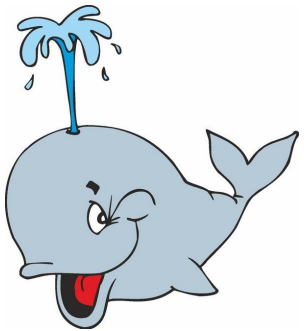
```
typedef char byte;

int getAge() {
        byte* ptr= (byte*) this;
        byte* ptrToAge = ptr + 4;
        int value = *((int*)ptrToAge);
        return value;
}
```

Начало объекта типа Student выглядит, как объект типа Person, поэтому метод getAge не видит подмены и работает



Полиморфизм

Способность функции обрабатывать данных разных типов



Ad hoc полиморфизм

Один интерфейс - много реализаций

В C++: перегрузка и переопределение методов, приведение типов, шаблоны

“Неполноценный” полиморфизм - общий только символ (имя) функции, для разных типов применяется разная реализация

Ad hoc полиморфизм

a + b

Ad hoc полиморфизм

a + b

a - int, b - int

a - float, b - float

a - int, b - float

a - Matrix, b - Matrix

Ad hoc полиморфизм

a + b	a - int, b - int	целочисленное сложение
	a - float, b - float	вещественное сложение
	a - int, b - float	приведение типа и вещественное сложение
	a - Matrix, b - Matrix	вызов метода

Параметрический полиморфизм

Одна реализация для множества типов данных (полиморфные функции)

В C++: наследование методов

Истинный полиморфизм - код полностью переиспользуется

Полиморфизм подтипов

Всё, что работает с элементами базового типа, может работать с элементами подтипа

В C++: указатель на базовый класс может хранить адрес объекта наследника

Синтаксический инструмент реализации полиморфизма

Полиморфизм подтипов

```
class Person {
        ...
        int getAge() {
                return this->age;
        }
};
```

```
class Student : public Person {
        ...
};
```

```
Student s(...);
s.getAge();
```

Полиморфизм подтипов

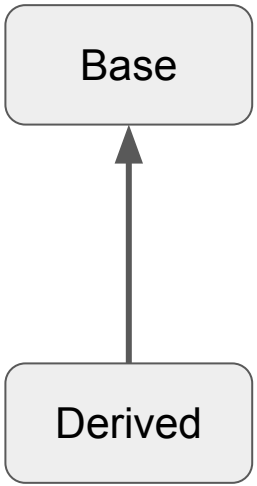
```
class Person {
        ...
        int getAge(/* Person* this */) {
                return this->age;
        }
};
```

```
class Student : public Person {
        ...
};
```

```
Student s(...);
s.getAge(); // Person::getAge(&s), &s имеет тип Student*
```

Полиморфизм подтипов

```
Derived X;
Base* ptr = &X;
```



Переопределение методов

```
class Person {
        char* name;
        int age;
public:
        void print() {
                printf(“%s, %d”,
                        this->name,
                        this->age);
        }
};
```

```
class Student : public Person {
        float gpa;
public:
        void print() {
                printf(“%s, %d, %f”,
                        this->name,
                        this->age,
                        this->gpa);
        }
};
```

Переопределение методов

```
class Person {
        char* name;
        int age;
public:
        void print() {
                printf(“%s, %d”,
                        this->name,
                        this->age);
        }
};
```

```
class Student : public Person {
        float gpa;
public:
        void print() {
                printf(“%s, %d, %f”,
                        this->name,
                        this->age,
                        this->gpa);
        }
};
```

Переопределение методов

```
class Person {
        char* name;
        int age;
public:
        void print() {
                printf(“%s, %d”,
                        this->name,
                        this->age);
        }
};
```

```
class Student : public Person {
        float gpa;
public:
        void print() {
                printf(“%s, %d, %f”,
                        this->name,
                        this->age,
                        this->gpa);
        }
};
```

можно “полечить” с помощью protected, но останется дублирование кода

Переопределение методов

```
class Person {
        char* name;
        int age;
public:
        void print() {
                printf(“%s, %d”,
                        this->name,
                        this->age);
        }
};
```

```
class Student : public Person {
        float gpa;
public:
        void print() {
                this->Person::print();
                printf(“, %f”, this->gpa);
        }
};
```

явный вызов реализации базового класса

Переопределение методов

```
Person p(...);
Student s(...);

p.print();
s.print();
```

по умолчанию вызывается **максимально специфичная реализация** (из своего класса, если есть, иначе из базового, иначе из его базового, и т.д.)

Переопределение методов

```
Person p(...);
Student s(...);

p.print();   // p.Person::print()
s.print();   // s.Student::print()
```

по умолчанию вызывается **максимально специфичная реализация** (из своего класса, если есть, иначе из базового, иначе из его базового, и т.д.)

Переопределение методов

```
Person p(...);
Student s(...);

p.print();   // p.Person::print()
s.print();   // s.Student::print()

s.Person::print();
```

можно вызвать нужную реализацию явно

Переопределение методов

Операция вызова переопределённого метода - ad hoc полиморфизм

Код только выглядит одинаково для Person и Student, в реальности это прямые вызовы разных методов

Переопределение методов

Операция вызова переопределённого метода - ad hoc полиморфизм

Код только выглядит одинаково для Person и Student, в реальности это **прямые** вызовы разных методов

> То есть, компилятор точно знает конкретную реализацию метода, которая вызывается

Не дружит с полиморфизмом подтипов

Переопределение методов

```
Person p(...);
Student s(...);

Person* ptr1 = &p;
Person* ptr2 = &s;

ptr1->print();     // Person::print
ptr2->print();
```

Переопределение методов

```
Person p(...);
Student s(...);

Person* ptr1 = &p;
Person* ptr2 = &s;

ptr1->print();    // Person::print
ptr2->print();    // Person::print !
```

Переопределение методов

```
Person p(...);
Student s(...);

Person* ptr1 = &p;
Person* ptr2 = &s;

ptr1->print();    // Person::print
ptr2->print();    // Person::print !
```

Виртуальные методы

```
Person p(...);
Student s(...);

Person* ptr1 = &p;
Person* ptr2 = &s;

ptr1->print();    // Person::print
ptr2->print();    // Person::print !
```

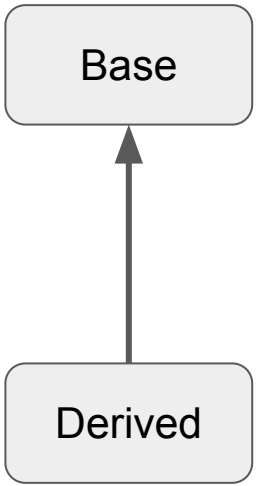
```
class Person {
        char* name;
        int age;
public:
        void print() {
                printf("%s, %d",
                        this->name,
                        this->age);
        }
};
```

Виртуальные методы

```
Person p(...);
Student s(...);

Person* ptr1 = &p;
Person* ptr2 = &s;

ptr1->print();    // Person::print
ptr2->print();    // Student::print !
```

```
class Person {
        char* name;
        int age;
public:
        virtual void print() {
                printf("%s, %d",
                        this->name,
                        this->age);
        }
};
```

Виртуальные методы




```
Derived X;
Base* ptr = &X;
ptr->foo();
```

Если foo объявлен, как виртуальный в классе Base, то вызовется реализация, максимально специфичная для Derived, иначе для Base

Как это работает

```
class Person {
        virtual void f0() { … }
        virtual void f1() { … }
        virtual void f2() { … }
};

class Student: public Person {
        void f0() { … }
        void f2() { … }
};
```

VMT - virtual method table

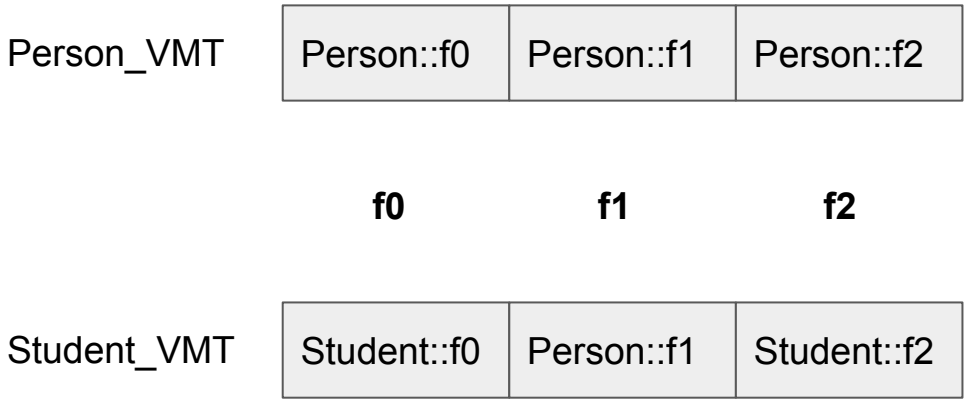
1) Массив указателей на функции

2) Создаётся компилятором в статическом классе памяти для каждого класса, содержащего хотя бы один виртуальный метод

3) Заполняется адресами максимально специфичных реализаций виртуальных методов

VMT - virtual method table

```
class Person {
        virtual void f0() { … }
        virtual void f1() { … }
        virtual void f2() { … }
};

class Student: public Person {
        void f0() { … }
        void f2() { … }
};
```



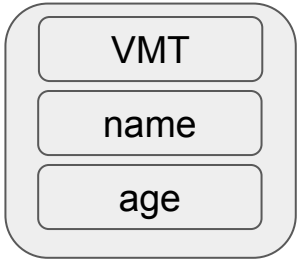
VMT - virtual method table

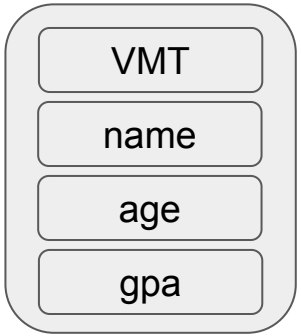
1) В объекте заводится дополнительное поле

2) Адрес VMT типа **T** записывается в это поле при создании объекта типа **T**

VMT - virtual method table

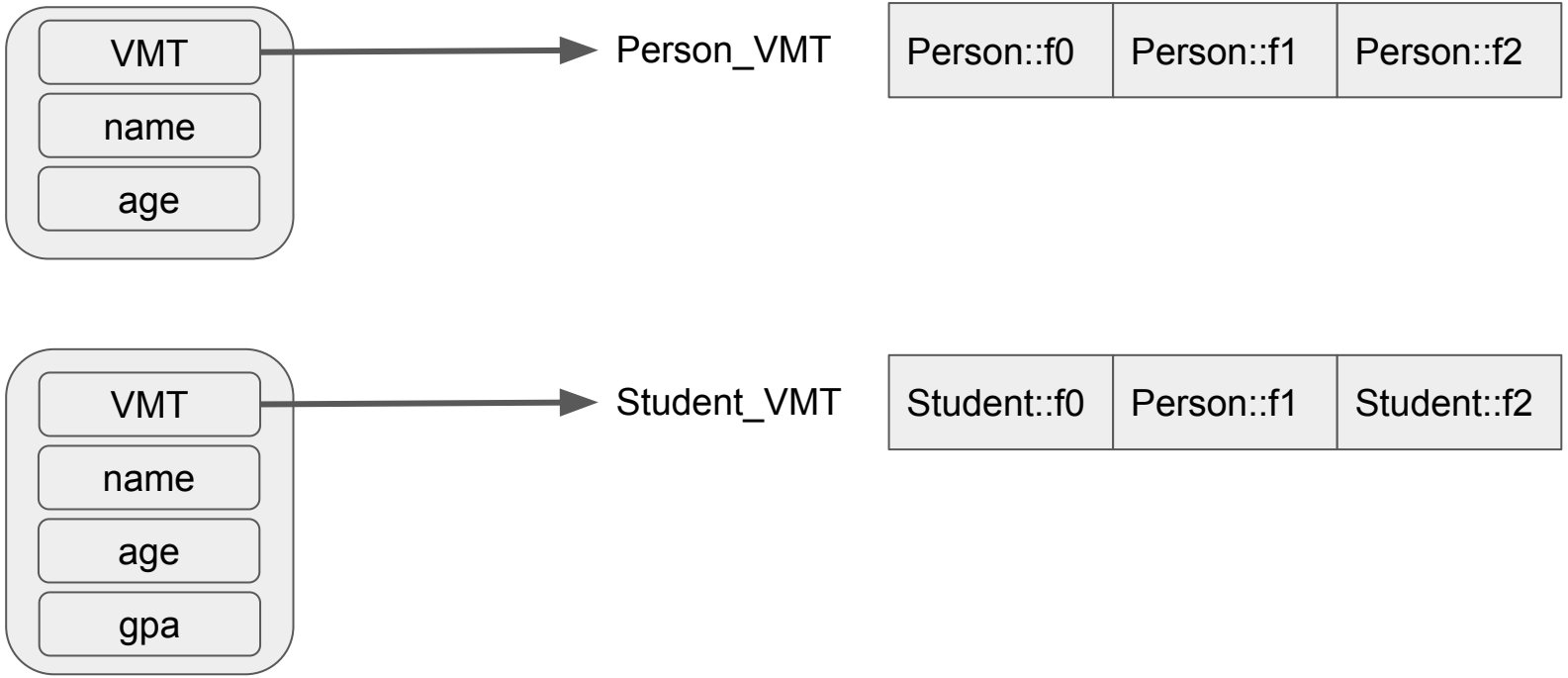
Person_VMT	Person::f0	Person::f1	Person::f2
Student_VMT	Student::f0	Person::f1	Student::f2






```
Person p(...);    // p.VMT = Person_VMT
Student s(...);   // s.VMT = Student_VMT
```

VMT - virtual method table



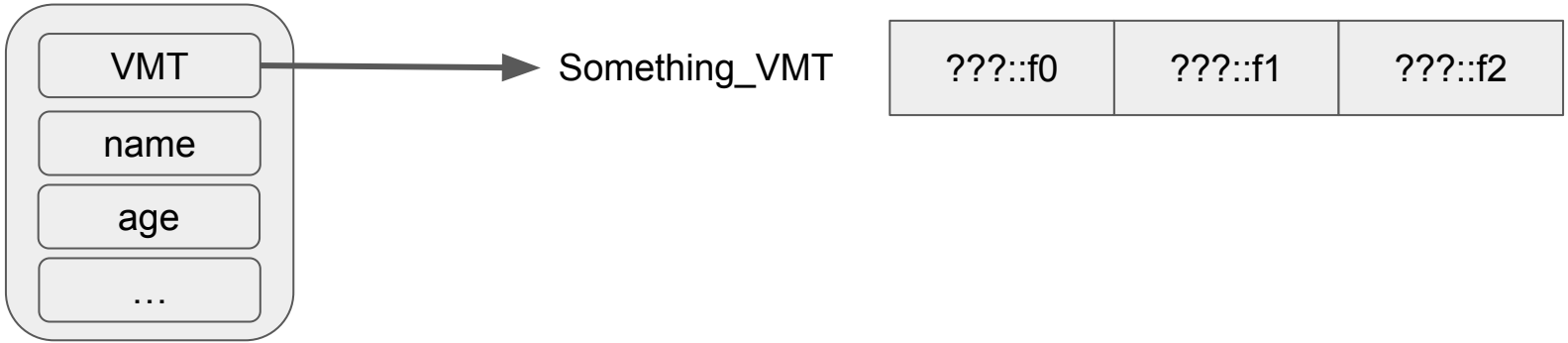
Прямой вызов

```
class Person {
        void f2() { … }
        ...
};

Person* ptr = ...;
ptr->f2();  // call Person::f2 (ptr)
```

Косвенный вызов

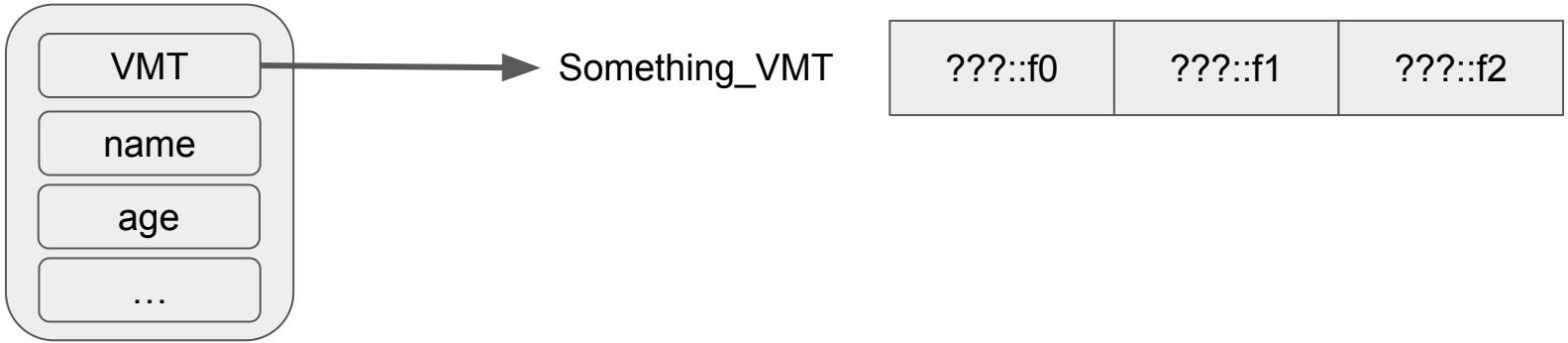
```
class Person {
        virtual void f2() { … }
        ...
};

Person* ptr = ...;
ptr->f2();  // call ptr->VMT[2] (ptr)
```

Косвенный вызов



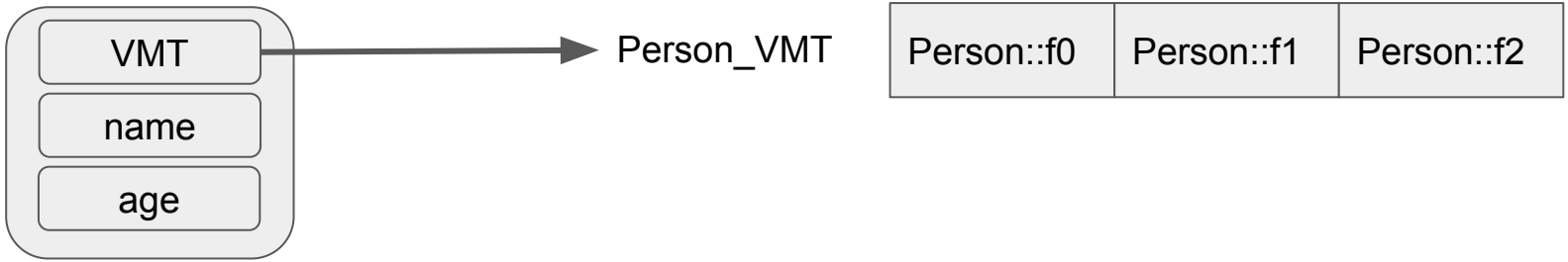

```
Person* ptr = ...;
ptr->f2();  // call ptr->VMT[2] (ptr)
```

Косвенный вызов



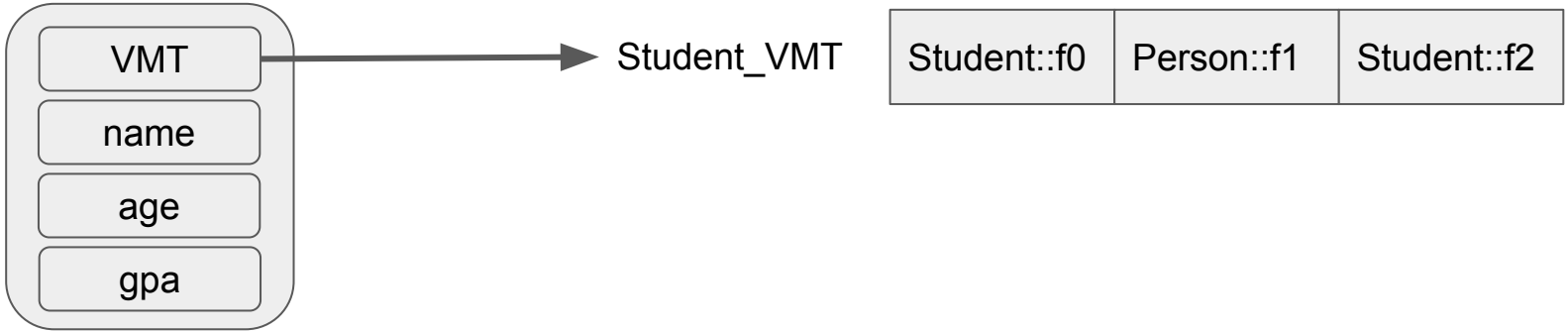

```
Person* ptr = ...;
ptr->f2();  // call ???::f2 (ptr)
```

Косвенный вызов




```
Person p;
Person* ptr = &p;
ptr->f2();  // call Person::f2 (ptr)
```

Косвенный вызов




```
Student s;
Person* ptr = &s;
ptr->f2();  // call Student::f2 (ptr)
```

Виртуальные методы

1) Код вызова является полноценным параметрическим полиморфизмом

2) Дружат с полиморфизмом подтипов

3) Содержат в себе накладные расходы
   a) Дополнительные поля в объектах - больше прокачка памяти
   b) Косвенный вызов дороже прямого и хуже оптимизируется

Деструкторы при наследовании

```
class Base {
        ~Base() {
                printf(“base part removed\n”);
        }
};

class Derived : public Base {
        ...
        ~Derived() {
                delete …;
                printf(“derived part removed\n”);
        }
}
```

```
Base* ptr = new Derived(...);
…
delete ptr;
```

Деструкторы при наследовании

```
class Base {
        ~Base() {
                printf(“base part removed\n”);
        }
};

class Derived : public Base {
        ...
        ~Derived() {
                delete …;
                printf(“derived part removed\n”);
        }
}
```

```
Base* ptr = new Derived(...);
…
delete ptr; // ptr->~Base()
```

```
>> base part removed
```

Деструкторы при наследовании

```
class Base {
        virtual ~Base() {
                printf(“base part removed\n”);
        }
};

class Derived : public Base {
        ...
        ~Derived() {
                delete …;
                printf(“derived part removed\n”);
        }
}
```

```
Base* ptr = new Derived(...);
…
delete ptr; // ptr->~Derived()
```

```
>> derived part removed
>> base part removed
```

Деструкторы при наследовании

```
class Base {
        virtual void foo() { printf(“Base::foo\n”); }
        virtual ~Base() {
                printf(“From ~Base()\n”);
                this->foo();
        }
};

class Derived : public Base {
        virtual void foo() { printf(“Derived::foo\n”); }
        virtual ~Derived() {
                printf(“From ~Derived()\n”);
                this->foo();
        }
}
```

```
Base* ptr = new Derived(...);
…
delete ptr;
```

Деструкторы при наследовании

```
class Base {
        virtual void foo() { printf(“Base::foo\n”); }
        virtual ~Base() {
                printf(“From ~Base()\n”);
                this->foo();
        }
};

class Derived : public Base {
        virtual void foo() { printf(“Derived::foo\n”); }
        virtual ~Derived() {
                printf(“From ~Derived()\n”);
                this->foo();
        }
}
```

```
Base* ptr = new Derived(...);
…
delete ptr;
```

```
>> From ~Derived()
>> Derived::foo()
>> From ~Base()
>> Base::foo()
```

Неявная семантика

```
class Foo : public Bar {
        ~Foo() {
                ...
        }
}
```

Неявная семантика

```
class Foo : public Bar {
        ~Foo() {
                // this->VMT = Foo_VMT;
                …
                // this->~Bar();
        }
}
```

Неявная семантика

```
class Foo : public Bar {
        ~Foo() {
                // this->VMT = Foo_VMT;
                …
                // this->~Bar();
        }
}
```

```
class Bar : public Baz {
        ~Bar() {
                // this->VMT = Bar_VMT;
                …
                // this->~Baz();
        }
}
```

Деструкторы при наследовании

Логика такого поведения в следующем:

1) Та часть объекта, которая относилась к классу-наследнику, уже уничтожена
2) Переопределённые методы могут пытаться использовать уничтоженные части

Деструкторы при наследовании

1) По умолчанию - не виртуальные

   Если от класса ожидается наследник, заведите виртуальный деструктор, пусть даже пустой

2) Порядок вызова - от наследника к базовому

3) Перед работой переписывают поле VMT к своему типу

   Потому что часть, относящаяся к наследнику, уже прошла через деструктор

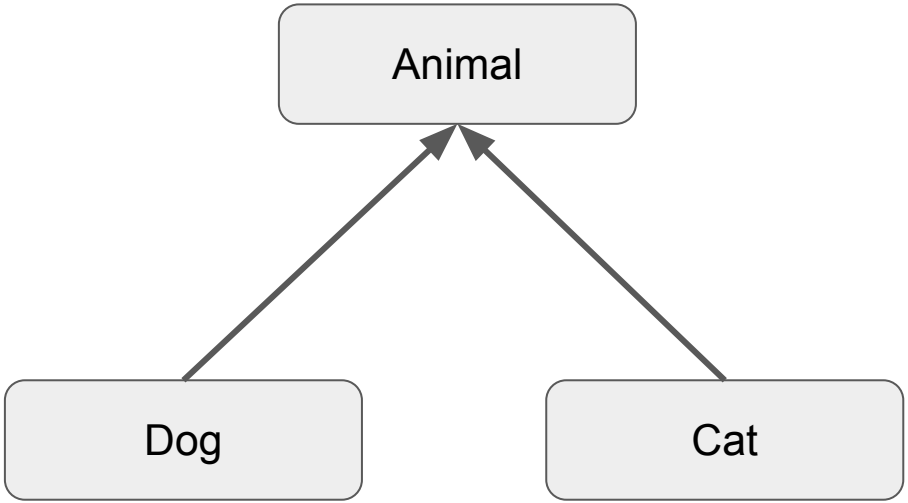
Конструкторы при наследовании

1) Порядок вызова - от базового к наследнику

2) Перед работой переписывают поле VMT к своему типу

Потому что часть, относящаяся к наследнику, ещё не построена

Абстрактные методы

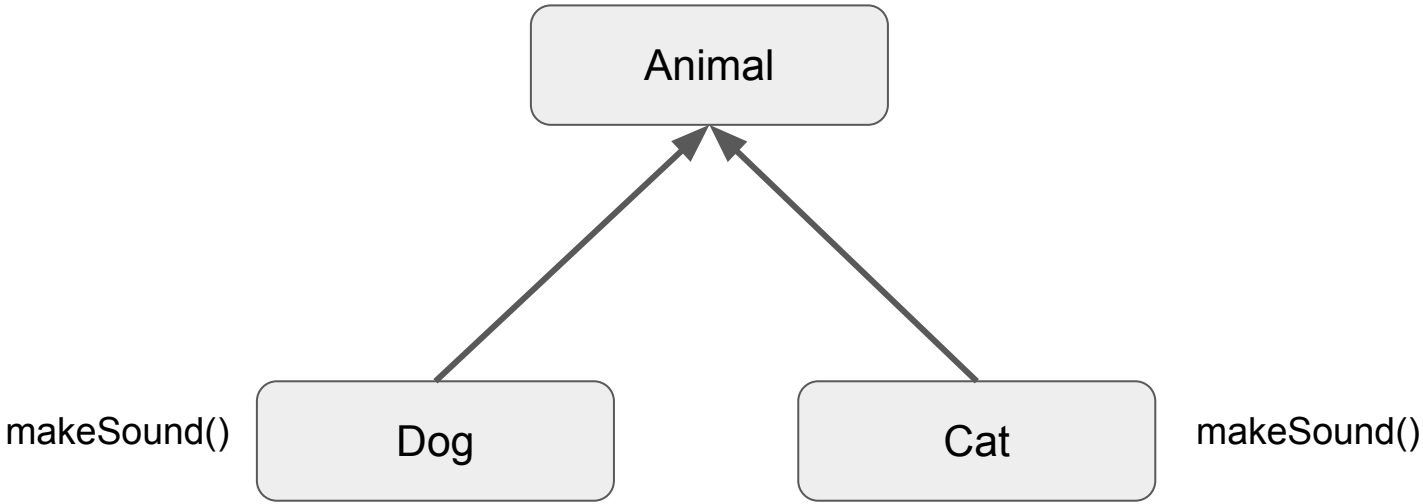


Абстрактные методы

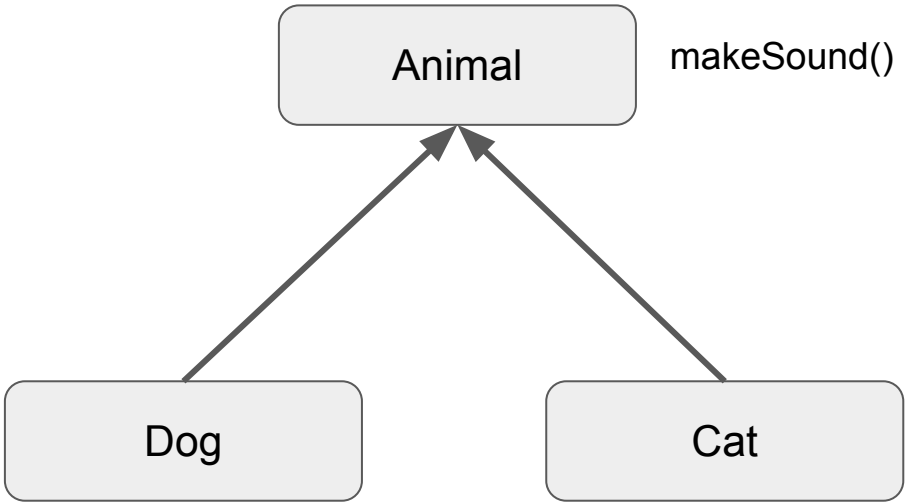
```
class Dog : public Animal {
        …
        void makeSound() {
                printf(“wow!\n”);
        }
};
```

```
class Cat : public Animal {
        …
        void makeSound() {
                printf(“meow!\n”);
        }
};
```

Абстрактные методы



Абстрактные методы



Абстрактные методы

```
class Dog : public Animal {
        …
        void makeSound() {
                printf(“wow!\n”);
        }
};
```

```
class Cat : public Animal {
        …
        void makeSound() {
                printf(“meow!\n”);
        }
};
```

```
class Animal {
        …
        virtual void makeSound()
};
```

Абстрактные методы

```
class Dog : public Animal {
        …
        void makeSound() {
                printf(“wow!\n”);
        }
};
```

```
class Cat : public Animal {
        …
        void makeSound() {
                printf(“meow!\n”);
        }
};
```

```
class Animal {
        …
        virtual void makeSound() {
                ???
        }
};
```

Абстрактные методы

```
class Dog : public Animal {
        …
        void makeSound() {
                printf(“wow!\n”);
        }
};
```

```
class Cat : public Animal {
        …
        void makeSound() {
                printf(“meow!\n”);
        }
};
```

```
class Animal {
        …
        virtual void makeSound() {
                assert(false);
        }
};
```

Абстрактные методы

```
class Dog : public Animal {
        …
        void makeSound() {
                printf(“wow!\n”);
        }
};
```

```
class Cat : public Animal {
        …
        void makeSound() {
                printf(“meow!\n”);
        }
};
```

```
class Animal {
        …
        virtual void makeSound() {
                printf(“do not call me!”);
        }
};
```

Абстрактные методы

```
class Dog : public Animal {
        …
        void makeSound() {
                printf(“wow!\n”);
        }
};
```

```
class Cat : public Animal {
        …
        void makeSound() {
                printf(“meow!\n”);
        }
};
```

```
class Animal {
        …
        virtual void makeSound() {
        }
};
```

Абстрактные методы

Свойство *говорить* типа Animal **определено**, но не может быть **реализовано**

Абстрактные методы

```
class Dog : public Animal {
        …
        void makeSound() {
                printf(“wow!\n”);
        }
};
```

```
class Cat : public Animal {
        …
        void makeSound() {
                printf(“meow!\n”);
        }
};
```

```
class Animal {
        …
        virtual void makeSound() = 0;
};
```

Абстрактные методы

1) Всегда виртуальный метод (обратное неверно)

2) Синоним (распространённый в C++) - чистый виртуальный метод

3) Класс с хотя бы одним чистым виртуальным методом становится **абстрактным**

Абстрактный класс

```
class Animal {
        …
        virtual void makeSound() = 0;
};
```

```
Animal x;    // нельзя создать объект абстрактного класса
```

Абстрактный класс

```
class Animal {
        …
        virtual void makeSound() = 0;
};
```

```
Animal x;    // нельзя создать объект абстрактного класса

Animal* ptr; // зато можно создать указатель на него
```

Абстрактный класс

```
class Animal {
        …
        virtual void makeSound() = 0;
};
```

```
Animal x;    // нельзя создать объект абстрактного класса

Animal* p1 = new Dog();
Animal* p2 = new Cat();
Animal* p3 = new Animal();
```

Убедитесь, что вынесли с этой лекции

Отношение наследования

Наследование полей и методов

Полиморфизм, классификация его видов

Переопределение методов

Виртуальные вызовы

Поведение конструкторов и деструкторов при наследовании

Абстрактные методы и классы

Проверочные вопросы

1) Чем отличается агрегирование от наследования?
2) В каком порядке вызываются конструкторы при наследовании?
3) В чём плюсы и минусы виртуальных вызовов?
4) В чём разница между ad hoc и параметрическим полиморфизмом?
5) Что такое VMT?
6) Какая неявная семантика в деструкторе при наследовании?
7) Можно ли адрес объекта базового класса записать в указатель на класс-наследник?
8) В чём разница между виртуальными и абстрактными методами?

Проверочные вопросы*

1) Реализуйте механизм виртуальных вызовов инструментами языка C
2) Спровоцируйте вызов абстрактного метода
3) Создайте объект абстрактного класса (работающий с точностью до вызовов абстрактных методов)

4) Вопрос на засыпку темы предыдущей лекции - найдите ошибку, преследующую многие слайды этой лекции, например на слайде №23

Q & A