Происхождение мутантов

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

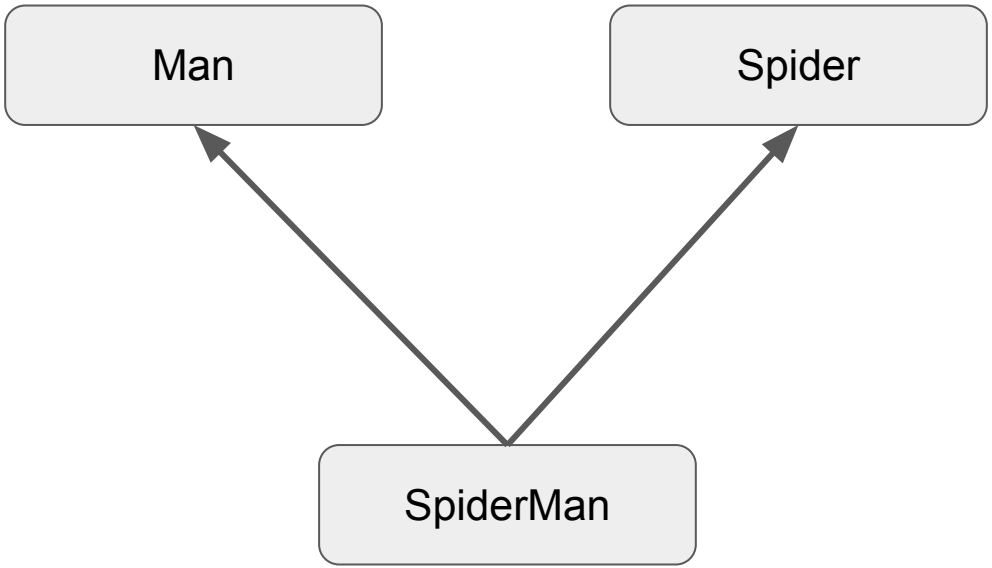
Тип “Человек-Паук” одновременно наследует свойства типов “Человек” и “Паук”


Man
Spider


char* name
void speak()
Man
Spider
bool toxicity
void shootWeb()

char* name

void speak()



bool toxicity

void shootWeb()

Множественное наследование

```
class Man {
        const char* name;
public:
        Man(const char* n): name(n) {
                printf(“Man constr\n”);
        }

        void speak() {
                printf(“My name is %s\n”, name);
        }
};
```

```
class Spider {
        bool toxicity;
public:
        Spider(bool tx): toxicity(tx) {
                printf(“Spider constr\n”);
        }

        void shootWeb() {
                printf(“Piu!\n”);
        }
};
```

Множественное наследование

```
class SpiderMan: public Spider, public Man {
public:
        SpiderMan(): Spider(false), Man(“Piter”) {}
};
```

Множественное наследование

```
class SpiderMan: public Spider, public Man {
public:
        SpiderMan(): Spider(false), Man(“Piter”) {}
};
```

```
SpiderMan sm;
sm.speak();
sm.shootWeb();
```

Spider constr
Man constr
My name is Piter
Piu!

Множественное наследование

```
class SpiderMan: public Spider, public Man {
public:
        SpiderMan(): Spider(false), Man(“Piter”) {}
};
```

```
SpiderMan sm;
sm.speak();
sm.shootWeb();
```

```
Spider constr
Man constr
My name is Piter
Piu!
```

Множественное наследование

```
class SpiderMan: public Spider, public Man {
public:
        SpiderMan(): Man(“Piter”), Spider(false) {}
};
```

```
SpiderMan sm;
sm.speak();
sm.shootWeb();
```

Spider constr
Man constr
My name is Piter
Piu!

Множественное наследование

```
class SpiderMan: public Spider, public Man {
public:
        SpiderMan(): Man(“Piter”), Spider(false) {}
};
```

Порядок вызова конструкторов базовых классов определён порядком наследования (и не может быть изменён в произвольном конструкторе)

Множественное наследование

```
SpiderMan sm;

SpiderMan* p1 = &sm;
Spider* p2 = p1;
Man* p3 = p1;

printf(“%d\n%d\n%d”, p1, p2, p3);
```

Множественное наследование

```
SpiderMan sm;

SpiderMan* p1 = &sm;
Spider* p2 = p1;
Man* p3 = p1;

printf(“%d\n%d\n%d”, p1, p2, p3);
```

```
9436516
9436516
9436520
```

Множественное наследование

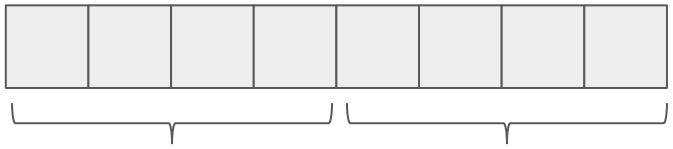
```
SpiderMan sm;

SpiderMan* p1 = &sm;
Spider* p2 = p1;
Man* p3 = p1;

printf("%d\n%d\n%d", p1, p2, p3);
```

9436516
9436516
9436520

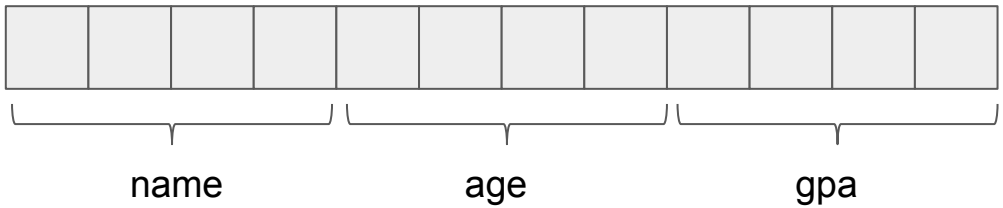
Прошлая лекция

class Student : public Person

объект типа Person в памяти



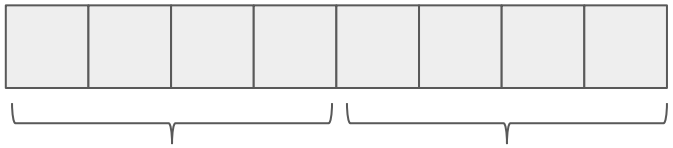
объект типа Student в памяти



Прошлая лекция

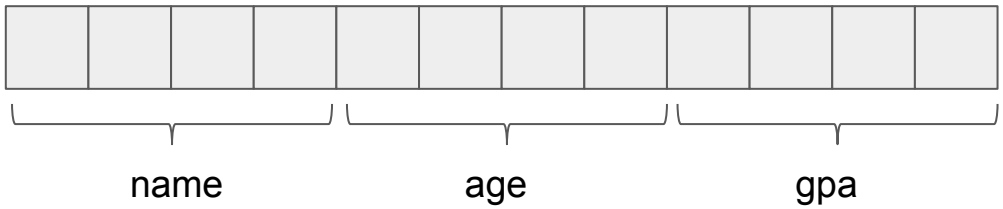
class Student : public Person

объект типа Person в памяти




```
Student* s = new Student();
Person* p = s;
```

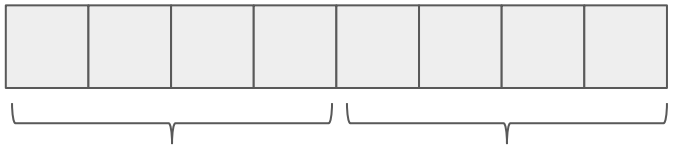
объект типа Student в памяти



Прошлая лекция

class Student : public Person

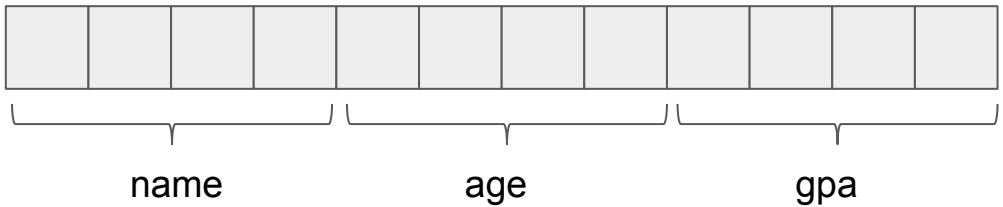
объект типа Person в памяти




```
Student* s = new Student();
Person* p = s;
```

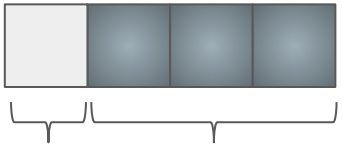
Другой тип, но значение то же самое

объект типа Student в памяти



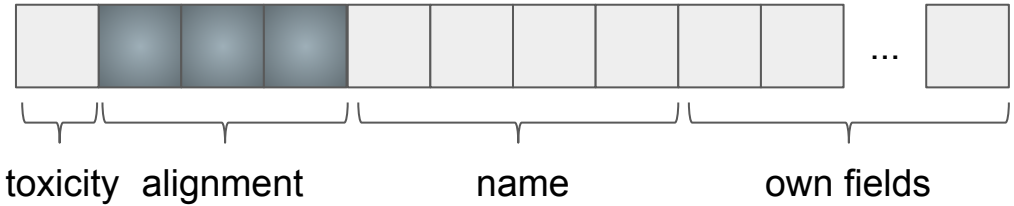
Со множественным наследованием так не выйдет

объект типа Spider в памяти

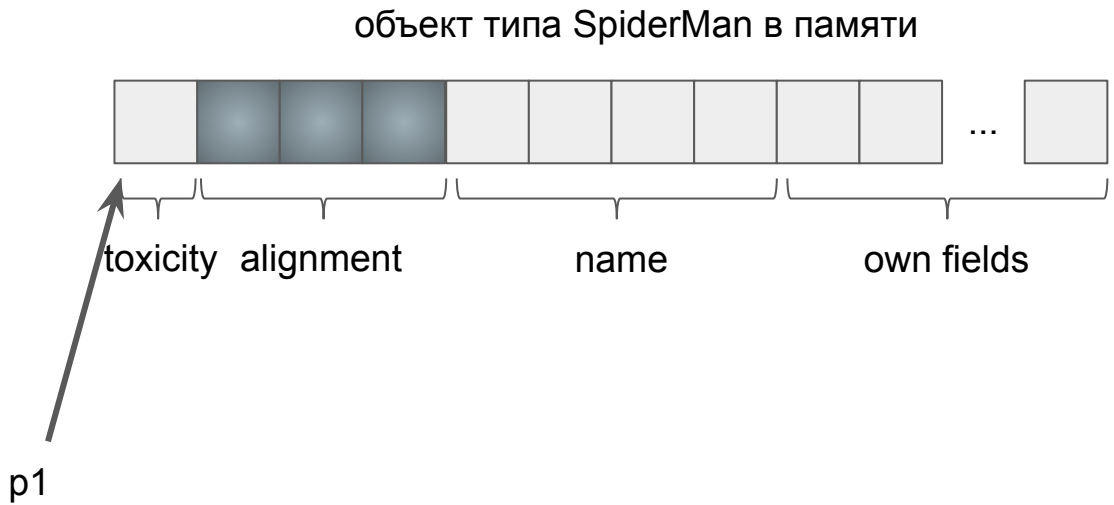


объект типа Man в памяти

объект типа SpiderMan в памяти

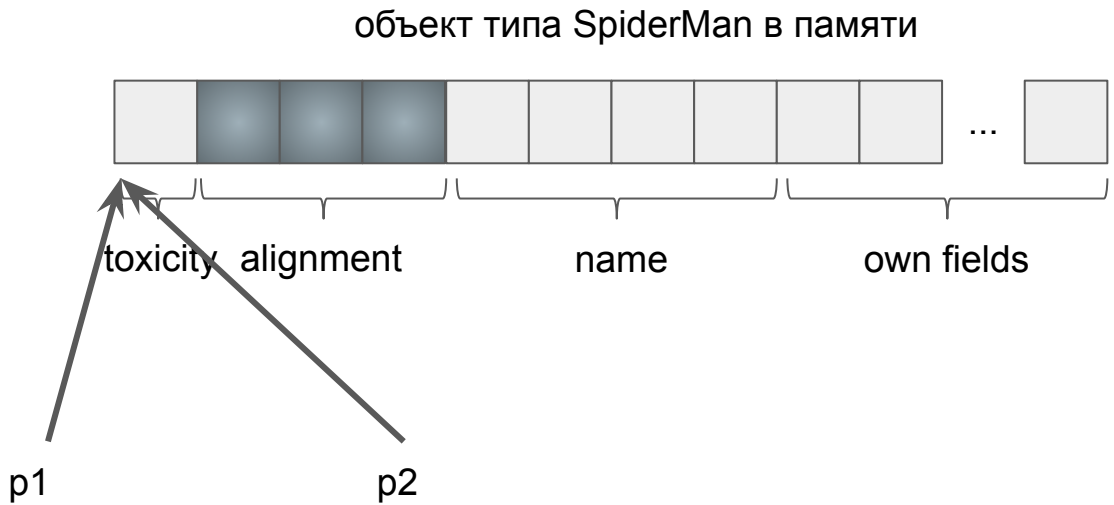


Со множественным наследованием так не выйдет



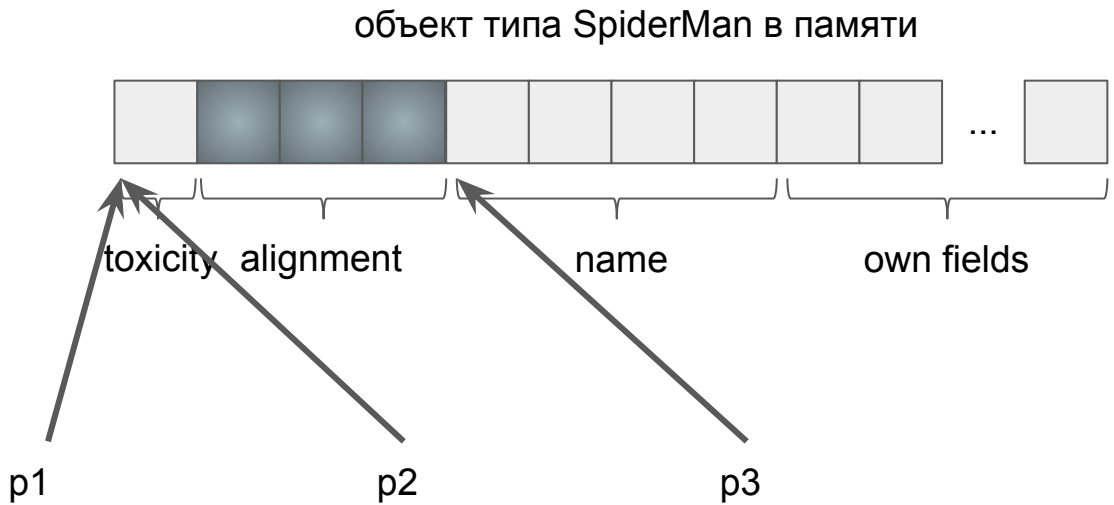

```
SpiderMan sm;

SpiderMan* p1 = &sm;
Spider* p2 = p1;
Man* p3 = p1;
```

Со множественным наследованием так не выйдет



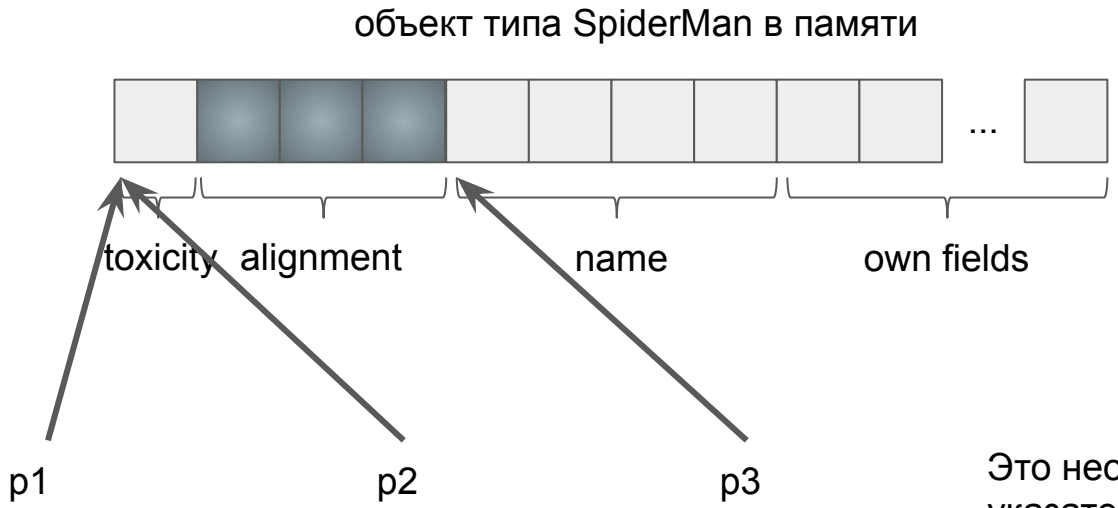

```
SpiderMan sm;

SpiderMan* p1 = &sm;
Spider* p2 = p1;
Man* p3 = p1;
```

Со множественным наследованием так не выйдет




```
SpiderMan sm;

SpiderMan* p1 = &sm;
Spider* p2 = p1;
Man* p3 = p1;
```

Со множественным наследованием так не выйдет




```
SpiderMan sm;

SpiderMan* p1 = &sm;
Spider* p2 = p1;
Man* p3 = p1;
```

Это необходимо, ведь p3 должен быть указателем на Man, чтобы из него можно быть читать поля в произвольном коде

А в обратную сторону?

```
Man* manPtr = …;
SpiderMan* smPtr = (SpiderMan*) manPtr;

printf(“%d\n%d”, manPtr, smPtr);
```

А в обратную сторону?

```
Man* manPtr = …;
SpiderMan* smPtr = (SpiderMan*) manPtr;

printf(“%d\n%d”, manPtr, smPtr);
```

17237352
17237348

А в обратную сторону?

```
Man* manPtr = …;
SpiderMan* smPtr = (SpiderMan*) manPtr;

printf(“%d\n%d”, manPtr, smPtr);
```

17237352
17237348

Компилятор знает, какое смещение у Man в составе SpiderMan, предполагает, что указатель корректный, и меняет его значение при касте

А в обратную сторону?

```
Man* manPtr = …;
SpiderMan* smPtr = (SpiderMan*) (void*) manPtr;

printf(“%d\n%d”, manPtr, smPtr);
```

```
17237352
17237352
```

Работает только между известными компилятору типами

Сырые касты указателей

1) В целом “не очень безопасная” операция

2) Со множественным наследованием становится ещё более интересной

Сырые касты указателей

```
Man* manPtr = …;
SpiderMan* smPtr = (SpiderMan*) manPtr;

if (manPtr == smPtr) {
        printf("%d\n%d", manPtr, smPtr);
}
```

Сырые касты указателей

```
Man* manPtr = …;
SpiderMan* smPtr = (SpiderMan*) manPtr;

if (manPtr == smPtr) {
        printf("%d\n%d", manPtr, smPtr);
}
```

```
3164440
3164436
```

Сырые касты указателей

```
Man* manPtr = …;
SpiderMan* smPtr = (SpiderMan*) manPtr;

if (manPtr == smPtr) {
        printf("%d\n%d", manPtr, smPtr);
}
```

```
3164440
3164436
```

Сравнение на равенство - что может быть проще?

Конфликты имён

```
class Man {
        …
public:
        const char* name;
};
```

```
class Spider {
        …
public:
        const char* name;
};
```

```
class SpiderMan: public Spider(“Piter”), public Man(“Piter”) { … };

SpiderMan sm;
printf(sm.name);
```

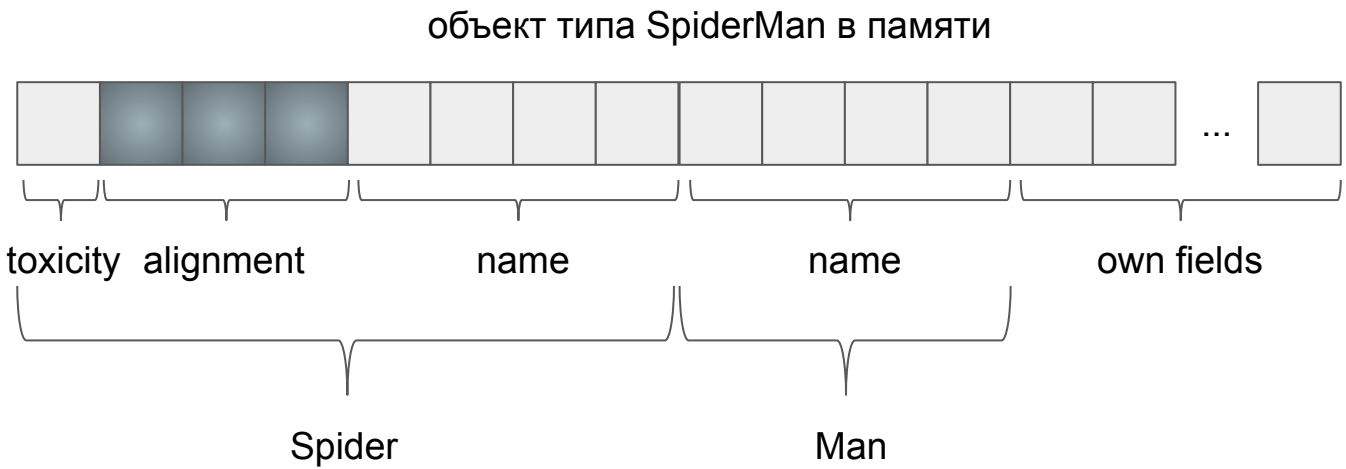
Конфликты имён

```
class Man {
        …
public:
        const char* name;
};
```

```
class Spider {
        …
public:
        const char* name;
};
```

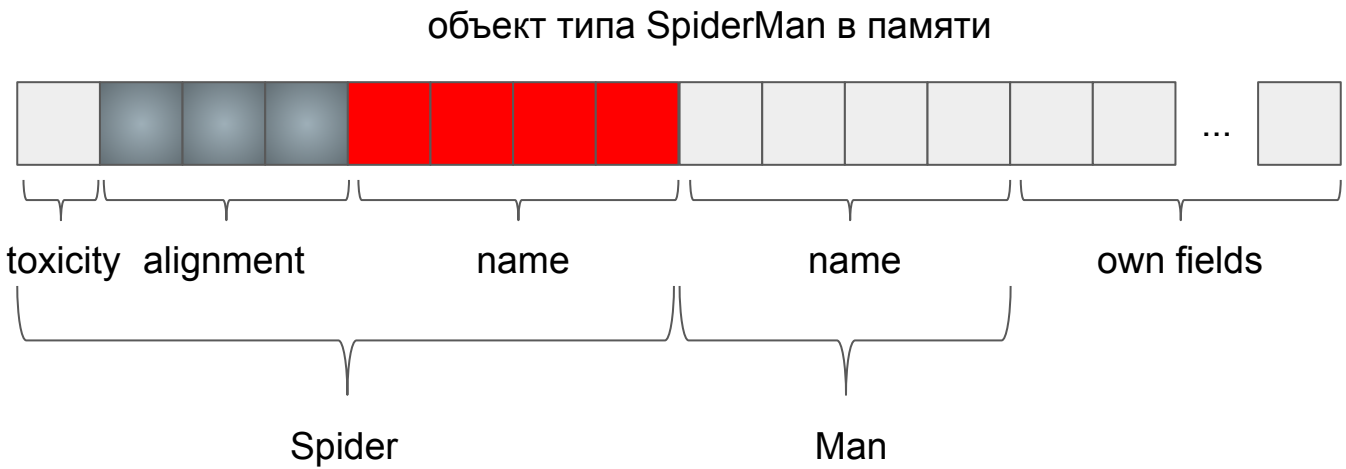
```
class SpiderMan: public Spider(“Piter”), public Man(“Piter”) { … };

SpiderMan sm;
printf(sm.name);
```

Конфликты имён



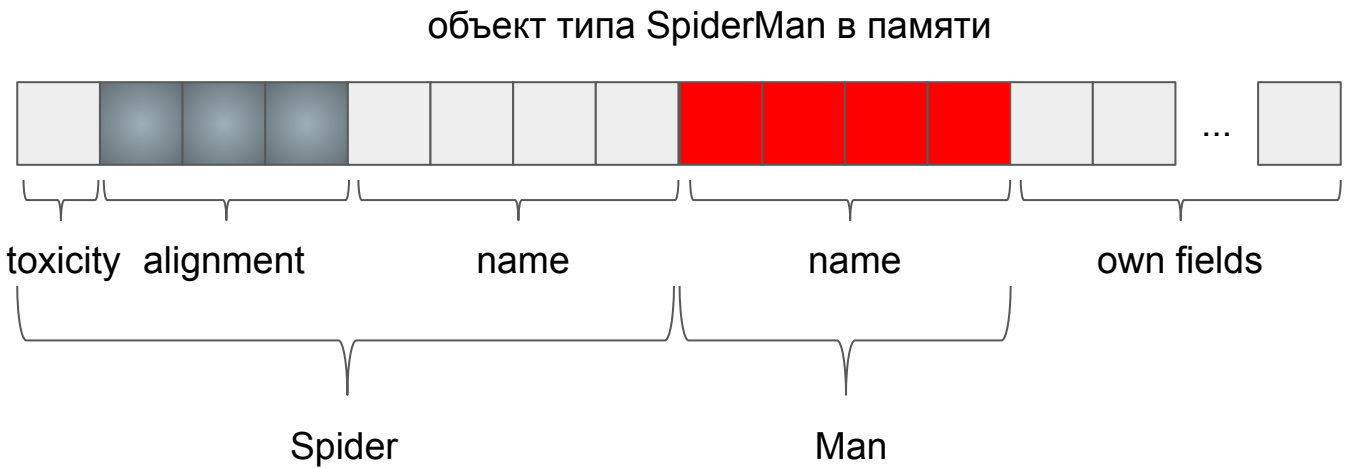

```
SpiderMan sm;
printf(sm.name);
```

Конфликты имён



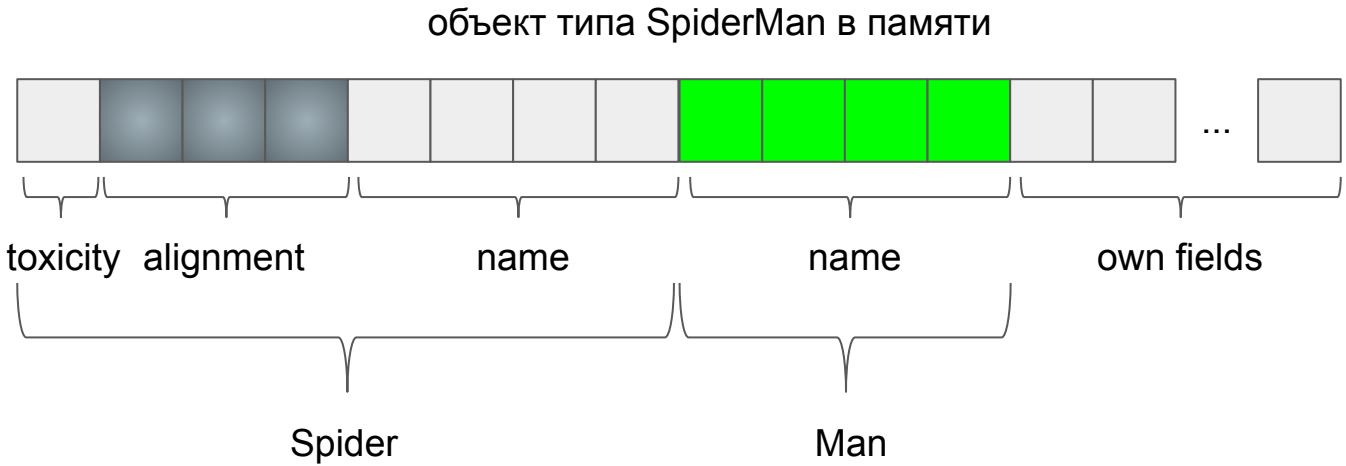

```
SpiderMan sm;
printf(sm.name);
```

Конфликты имён



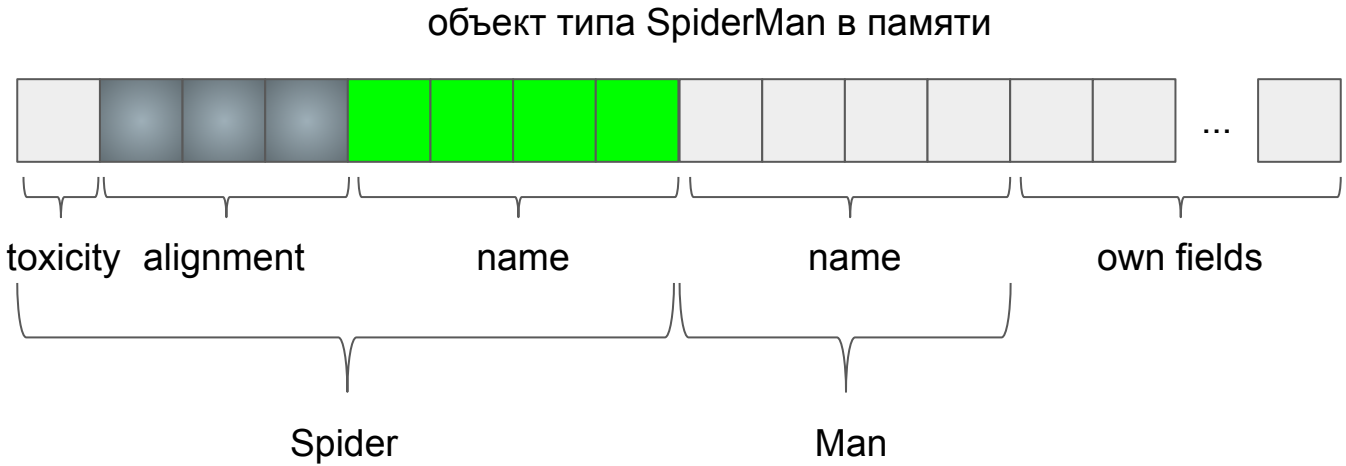

```
SpiderMan sm;
printf(sm.name);
```

Конфликты имён




```
SpiderMan sm;
printf(sm.Man::name);
```

Конфликты имён




```
SpiderMan sm;
printf(sm.Spider::name);
```

Но они же одинаковые!

```
class Man {
        …
public:
        const char* name;
};
```

```
class Spider {
        …
public:
        const char* name;
};
```

```
class SpiderMan: public Spider(“Piter”), public Man(“Piter”) { … };

SpiderMan sm;
printf(sm.name);
```

Формальность превыше всего

1) Компилятор не умеет угадывать, что вы там себе придумали о своей программе

Формальность превыше всего

1) Компилятор не умеет угадывать, что вы там себе придумали о своей программе (жаль, конечно)

Формальность превыше всего

1) Компилятор не умеет угадывать, что вы там себе придумали о своей программе (жаль, конечно)

2) Раз поля разные - значит они разные!

3) Ведь в другом примере они действительно могут быть разные

Конфликты имён

```
class Man {
        …
public:
        Man(...): legsAmount(2) ...
        int legsAmount;
};
```

```
class Spider {
        …
public:
        Spider(...): legsAmount(8) ...
        int legsAmount;
};
```

```
class SpiderMan: public Spider, public Man { … };

SpiderMan sm;
printf(“%d”, sm.legsAmount);
```

Конфликты имён

1) На самом деле хорошо, потому что указывает на потенциальные ошибки

2) Пример с одинаковыми полями name - ошибка проектирования (решение будет через N слайдов)

Переопределение полей

```
class Man {
        …
public:
        Man(...): legsAmount(2) ...
        int legsAmount;
};
```

```
class Spider {
        …
public:
        Spider(...): legsAmount(8) ...
        int legsAmount;
};
```

```
class SpiderMan: public Spider, public Man {
        …
public:
        SpiderMan(...): Spider(...), Man(...) ...
};
```

```
SpiderMan sm;
printf(“%d”, sm.legsAmount);
```

Переопределение полей

```
class Man {
        …
public:
        Man(...): legsAmount(2) ...
        int legsAmount;
};
```

```
class Spider {
        …
public:
        Spider(...): legsAmount(8) ...
        int legsAmount;
};
```

```
class SpiderMan: public Spider, public Man {
        …
public:
        SpiderMan(...): Spider(...), Man(...), legsAmount(2) ...
        int legsAmount;
};

SpiderMan sm;
printf(“%d”, sm.legsAmount);
```

Переопределение полей

В этом примере

1) Не экономит память (все три поля присутствуют в объекте)

2) Немного улучшает семантику (понятно, какое поле главное)

Переопределение методов

```
class Man {
        …
public:
        int legsAmount() { return 2; }
};
```

```
class Spider {
        …
public:
        int legsAmount() { return 8; }
};
```

```
class SpiderMan: public Spider, public Man {
        …
public:
};
```

```
SpiderMan sm;
printf(“%d”, sm.legsAmount());
```

Переопределение методов

```
class Man {
        …
public:
        int legsAmount() { return 2; }
};
```

```
class Spider {
        …
public:
        int legsAmount() { return 8; }
};
```

```
class SpiderMan: public Spider, public Man {
        …
public:
};
```

```
SpiderMan sm;
printf(“%d”, sm.Man::legsAmount());
```

Переопределение методов

```
class Man {
        …
public:
        int legsAmount() { return 2; }
};
```

```
class Spider {
        …
public:
        int legsAmount() { return 8; }
};
```

```
class SpiderMan: public Spider, public Man {
        …
public:
};
```

```
SpiderMan sm;
printf(“%d”, sm.Spider::legsAmount());
```

Переопределение методов

```
class Man {
        …
public:
        int legsAmount() { return 2; }
};
```

```
class Spider {
        …
public:
        int legsAmount() { return 8; }
};
```

```
class SpiderMan: public Spider, public Man {
        …
public:
        int legsAmount() { return 2; }
};

SpiderMan sm;
printf(“%d”, sm.legsAmount());
```

Переопределение методов

1) Позволяет решать конфликты имён и реализаций

2) Так же как и в одиночном наследовании, не дружит с полиморфизмом подтипов (т.к. вызов прямой, т.е. ad hoc полиморфный)

Переопределение методов

```
class SpiderMan: public Spider, public Man {
        …
public:
        int legsAmount() { return 2; }
};


SpiderMan sm;
Spider* ptr = &sm;

ptr->legsAmount(); // Spider::legsAmount()
```

Виртуальные методы

```
class Man {
        …
public:
        virtual int legsAmount() { return 2; }
};
```

```
class Spider {
        …
public:
        virtual int legsAmount() { return 8; }
};
```

```
class SpiderMan: public Spider, public Man {
        …
public:
        int legsAmount() { return 2; }
};

SpiderMan sm;
Spider* ptr = &sm;
printf("%d", ptr->legsAmount());      // SpiderMan::legsAmount()
```

Виртуальные методы

```
class Man {
        …
public:
        int legsAmount() { return 2; }
};
```

```
class Spider {
        …
public:
        virtual int legsAmount() { return 8; }
};
```

```
class SpiderMan: public Spider, public Man {
        …
public:
        int legsAmount() { return 2; }
};

SpiderMan sm;
Spider* ptr = &sm;
printf(“%d”, ptr->legsAmount());      // SpiderMan::legsAmount()
```

Виртуальные методы

```
class Man {
        …
public:
        virtual int legsAmount() { return 2; }
};
```

```
class Spider {
        …
public:
        int legsAmount() { return 8; }
};
```

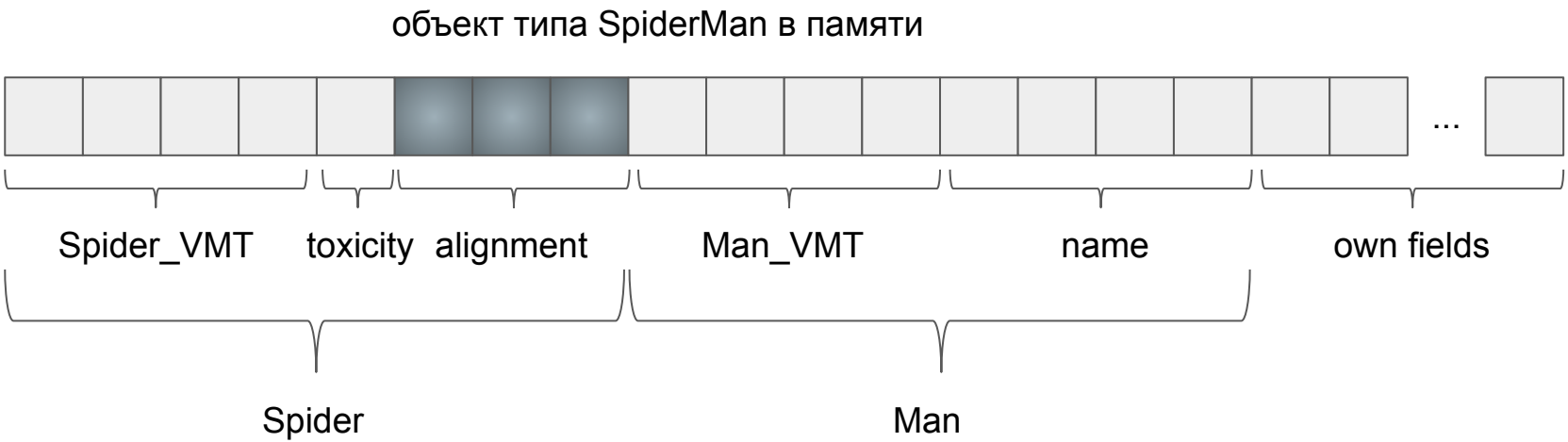
```
class SpiderMan: public Spider, public Man {
        …
public:
        int legsAmount() { return 2; }
};

SpiderMan sm;
Spider* ptr = &sm;
printf("%d", ptr->legsAmount());      // Spider::legsAmount()
```

Виртуальные методы

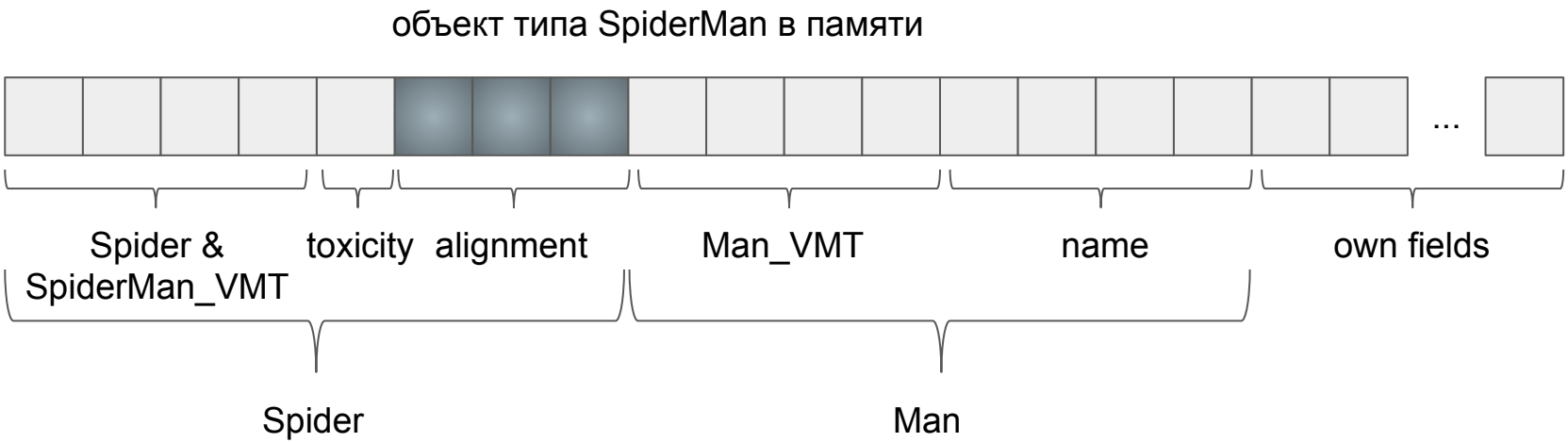
1) При конфликтах имён работают только для тех подтипов, где объявлены, как виртуальные

2) Вызовы косвенные, дороже, зато параметрический полиморфизм

Виртуальные методы



Виртуальные методы



VMT при множественном наследовании

1) Присутствуют по экземпляру на каждый базовый класс

2) В таблице для первого базового класса перечислены адреса его виртуальных методов и виртуальных методов класса наследника

3) В таблицах для остальных базовых классов перечислены адреса их виртуальных методов

VMT при множественном наследовании

1) Так надо

2) VMT в середине объекта нужны, чтобы скастованный к соответствующему базовому типу указатель вёл себя, как надо

3) Расширенная VMT в начале нужна на случай, если от SpiderMan будут наследоваться

VMT при множественном наследовании

Множественное наследование могло бы быть реализовано и по другому

VMT при множественном наследовании

Множественное наследование могло бы быть реализовано и по другому

Но простых (относительно одиночного наследование) реализаций нет

Виртуальные методы

```
class Man {
        …
public:
        void foo() {
                bar();
        }

        virtual void bar() {
                ...
        }
};
```

```
class SpiderMan: public Spider, public Man {
        …
public:
        virtual void bar() {
                ...
        }
};
```

```
SpiderMan sm;
sm.foo();
sm.bar();
```

Виртуальные методы

```
class Man {
        …
public:
        void foo(/* Man* this */) {
                bar(); // bar(this)
        }

        virtual void bar() {
                ...
        }
};
```

```
class SpiderMan: public Spider, public Man {
        …
public:
        virtual void bar(/* SpiderMan* this */) {
                ...
        }
};
```

```
SpiderMan sm;
sm.foo(); // foo((Man*)&sm)
sm.bar(); // bar(&sm)
```

Виртуальные методы

```
class Man {
        …
public:
        void foo(/* Man* this */) {
                bar(); // bar(this)
        }

        virtual void bar() {
                ...
        }
};
```

```
class SpiderMan: public Spider, public Man {
        …
public:
        virtual void bar(/* SpiderMan* this */) {
                ...
        }
};
```

```
SpiderMan sm;
sm.foo(); // foo((Man*)&sm)
sm.bar(); // bar(&sm)
```

Виртуальные вызовы

1) SpiderMan::bar ждёт указателя на SpiderMan
2) Man::foo, вызывая виртуально ???::bar, передаёт указатель на Man
3) Явный вызов SpiderMan::bar передаёт указатель на SpiderMan

И если в одиночном наследовании указатель на Man и на SpiderMan - это одно и то же, то в множественном они имеют разные значения!

Виртуальные вызовы

1) SpiderMan::bar ждёт указателя на SpiderMan
2) Man::foo, вызывая виртуально ???::bar, передаёт указатель на Man
3) Явный вызов SpiderMan::bar передаёт указатель на SpiderMan

И если в одиночном наследовании указатель на Man и на SpiderMan - это одно и то же, то в множественном они имеют разные значения!

Кто-то точно сломается

Виртуальные методы

```
class Man {
        …
public:
        void foo(/* Man* this */) {
                bar(); // bar(this)
        }

        virtual void bar() {
                ...
        }
};
```

```
class SpiderMan: public Spider, public Man {
        …
public:
        virtual void bar(/* SpiderMan* this */) {
                ...
        }
};
```

```
SpiderMan sm;
sm.foo(); // foo((Man*)&sm)
sm.bar(); // bar(&sm)
```

Виртуальные методы

```
class Man {
        …
public:
        void foo(/* Man* this */) {
                bar(); // bar(this)
        }

        virtual void bar() {
                ...
        }
};
```

```
class SpiderMan: public Spider, public Man {
        …
public:
        virtual void bar(/* Man* this */) {
                // this = (SpiderMan*) this;
                ...
        }
};
```

```
SpiderMan sm;
sm.foo(); // foo((Man*)&sm)
sm.bar(); // bar((Man*)&sm)
```

Виртуальные методы

1) Во всех реализациях виртуального метода указатель this имеет тип указателя на класс, описавший виртуальный метод (базовый)

2) В каждой реализации этот указатель первым делом кастуется к типу указателя на класс-наследник

Виртуальные методы

1) Во всех реализациях виртуального метода указатель this имеет тип указателя на класс, описавший виртуальный метод (базовый)

2) В каждой реализации этот указатель первым делом кастуется к типу указателя на класс-наследник

для одиночного наследования эта операция бесплатна
для множественного - одно вычитание

Виртуальные методы

```
class Man {
        …
public:
        virtual void bar() { … }
};
```

```
class SpiderMan: public Spider, public Man {
        …
public:
        virtual void bar(/* Man* this */) {
                // this = (SpiderMan*) this;
                ...
        }
};
```

Виртуальные методы

```
class Man {
        …
public:
        virtual void bar() { … }
};
```

```
class SpiderMan: public Spider, public Man {
        …
public:
        virtual void bar(/* Man* this */) {
                // this = (SpiderMan*) this;
                ...
        }
};
```

Виртуальные методы

```
class Man {
        …
public:
        virtual void bar() { … }
};
```

```
class Spider {
        …
public:
        virtual void bar() { … }
};
```

```
class SpiderMan: public Spider, public Man {
        …
public:
        virtual void bar(/* Man* this */) {
                // this = (SpiderMan*) this;
                ...
        }
};
```

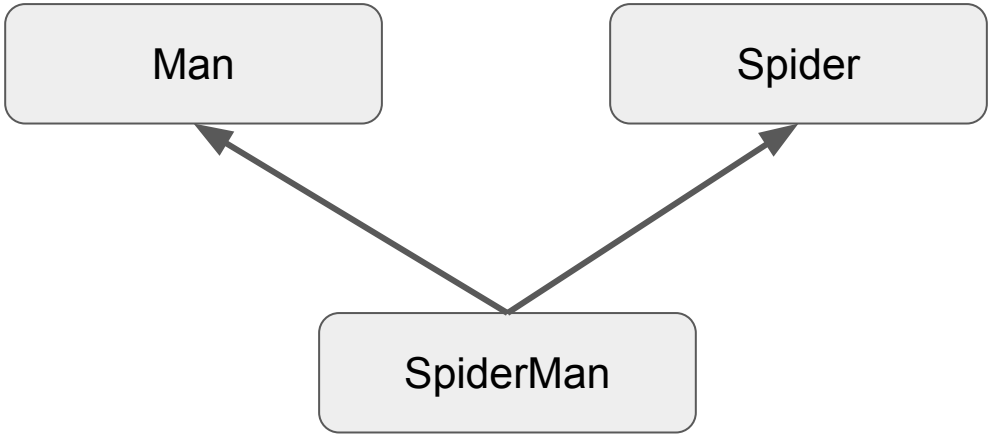
Виртуальные методы

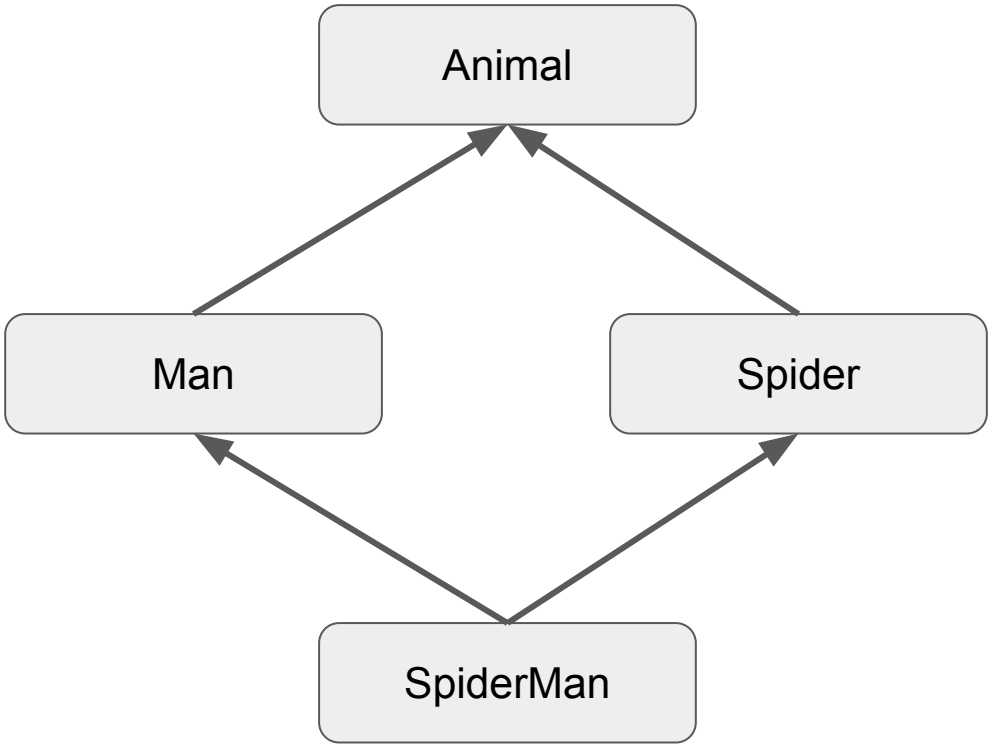
```
class Man {
        …
public:
        virtual void bar() { … }
};
```

```
class Spider {
        …
public:
        virtual void bar() { … }
};
```

```
class SpiderMan: public Spider, public Man {
        …
public:
        virtual void bar(/* Man* this */) {
                // this = (SpiderMan*) this;
                ...
        }
};
```

Упражнение


Man
Spider
SpiderMan


Animal
Man
Spider
SpiderMan

Ромбовидное наследование

```
class Animal {
public:
        const char* name;
        virtual void speak() = 0;
};
```

```
class Man: public Animal {
public:
        virtual void speak() {
                cout << “My name is ” << name;
        };
};
```

```
class Spider: public Animal {
public:
        virtual void speak() {
                cout << “Shhhh”;
        };
};
```

Ромбовидное наследование

```
class SpiderMan: public Spider, public Man {
};
```

```
SpiderMan sm;
sm.speak();
sm.name;
(Animal*) &sm;
```

Ромбовидное наследование

```
class SpiderMan: public Spider, public Man {
};
```

```
SpiderMan sm;
sm.speak();
sm.name;
(Animal*) &sm;
```

Ромбовидное наследование

```
class SpiderMan: public Spider, public Man {
};
```

```
SpiderMan sm;
sm.Spider::speak();
sm.name;
(Animal*) &sm;
```

Ромбовидное наследование

```
class SpiderMan: public Spider, public Man {
};
```

```
SpiderMan sm;
sm.Man::speak();
sm.name;
(Animal*) &sm;
```

Ромбовидное наследование

```
class SpiderMan: public Spider, public Man {
};
```

```
SpiderMan sm;
sm.speak();
sm.Spider::name;
(Animal*) &sm;
```

Ромбовидное наследование

```
class SpiderMan: public Spider, public Man {
};
```

```
SpiderMan sm;
sm.speak();
sm.Man::name;
(Animal*) &sm;
```

Ромбовидное наследование

```
class SpiderMan: public Spider, public Man {
};
```

```
SpiderMan sm;
sm.speak();
sm.name;
(Animal*) (Spider*) &sm;
```

Ромбовидное наследование

```
class SpiderMan: public Spider, public Man {
};
```

```
SpiderMan sm;
sm.speak();
sm.name;
(Animal*) (Man*) &sm;
```

Ромбовидное наследование

```
class SpiderMan: public Spider, public Man {
};
```

```
class Animal {
public:
        const char* name;
        virtual void speak() = 0;
};
```

Но ведь поле name определено 1 раз?

```
SpiderMan sm;
sm.speak();
sm.Man::name;
(Animal*) &sm;
```

Ромбовидное наследование

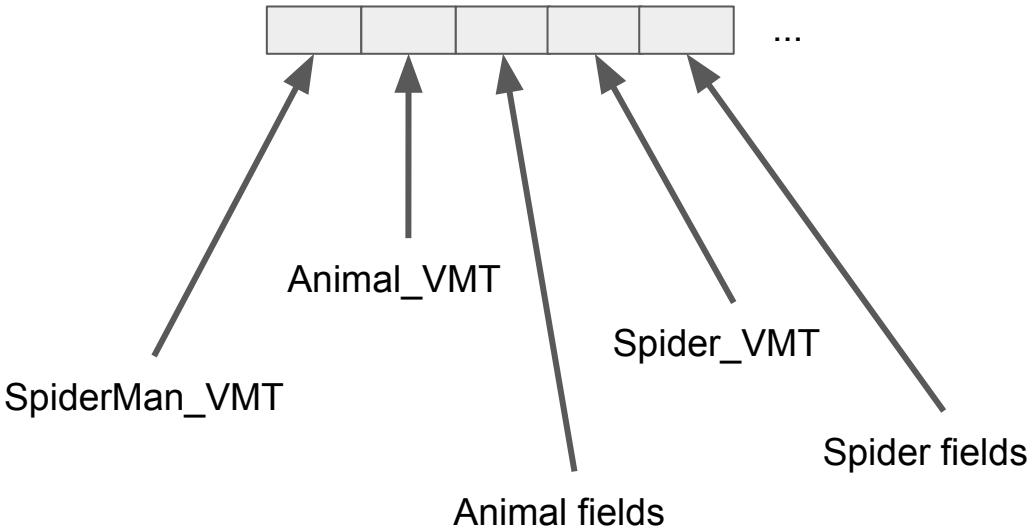
```
class SpiderMan: public Spider, public Man {
};
```

```
class Animal {
public:
	const char* name;
	virtual void speak() = 0;
};
```

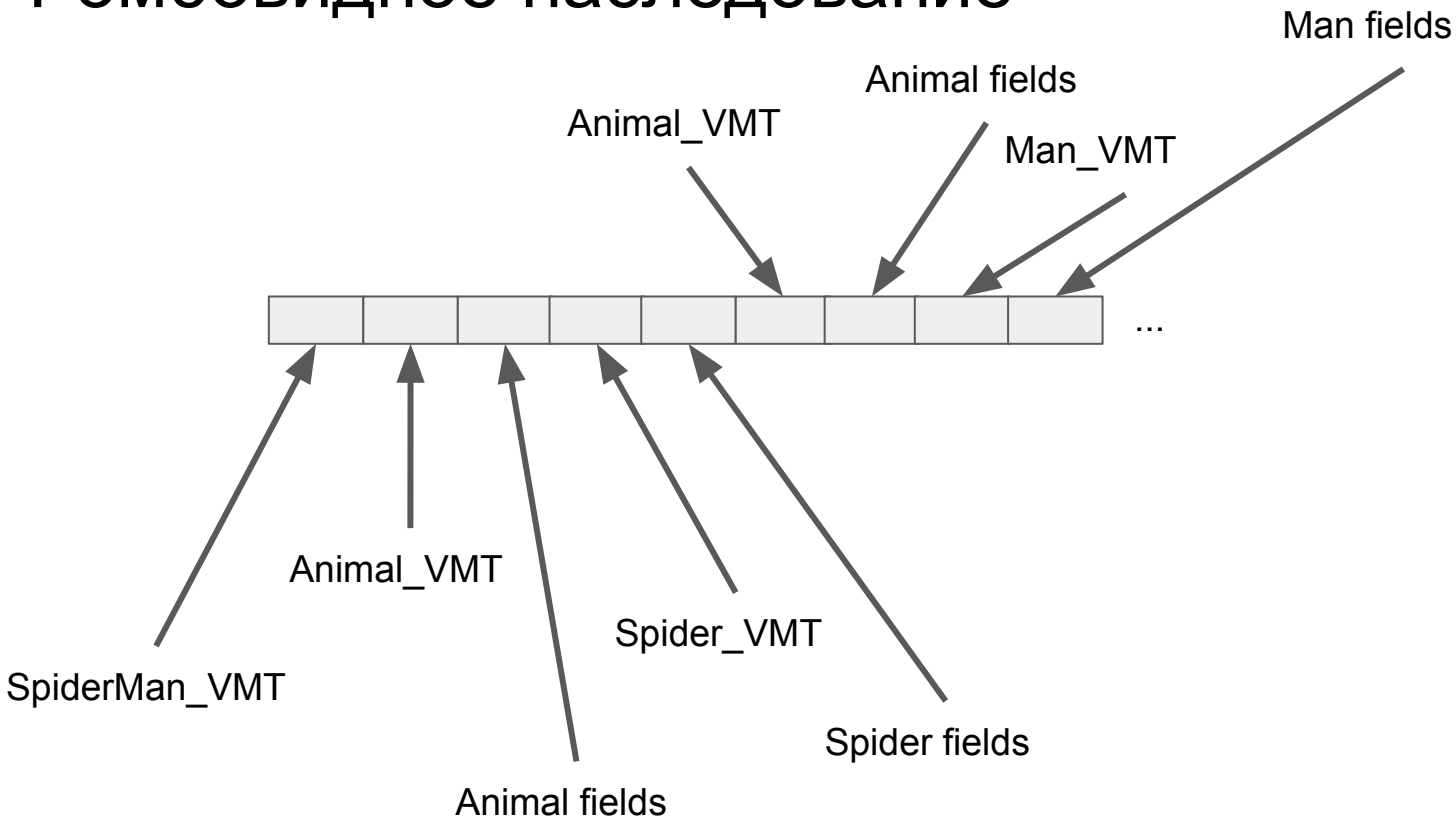
И почему нельзя кастануть сразу?

```
SpiderMan sm;
sm.speak();
sm.name;
(Animal*) (Spider*) &sm;
```

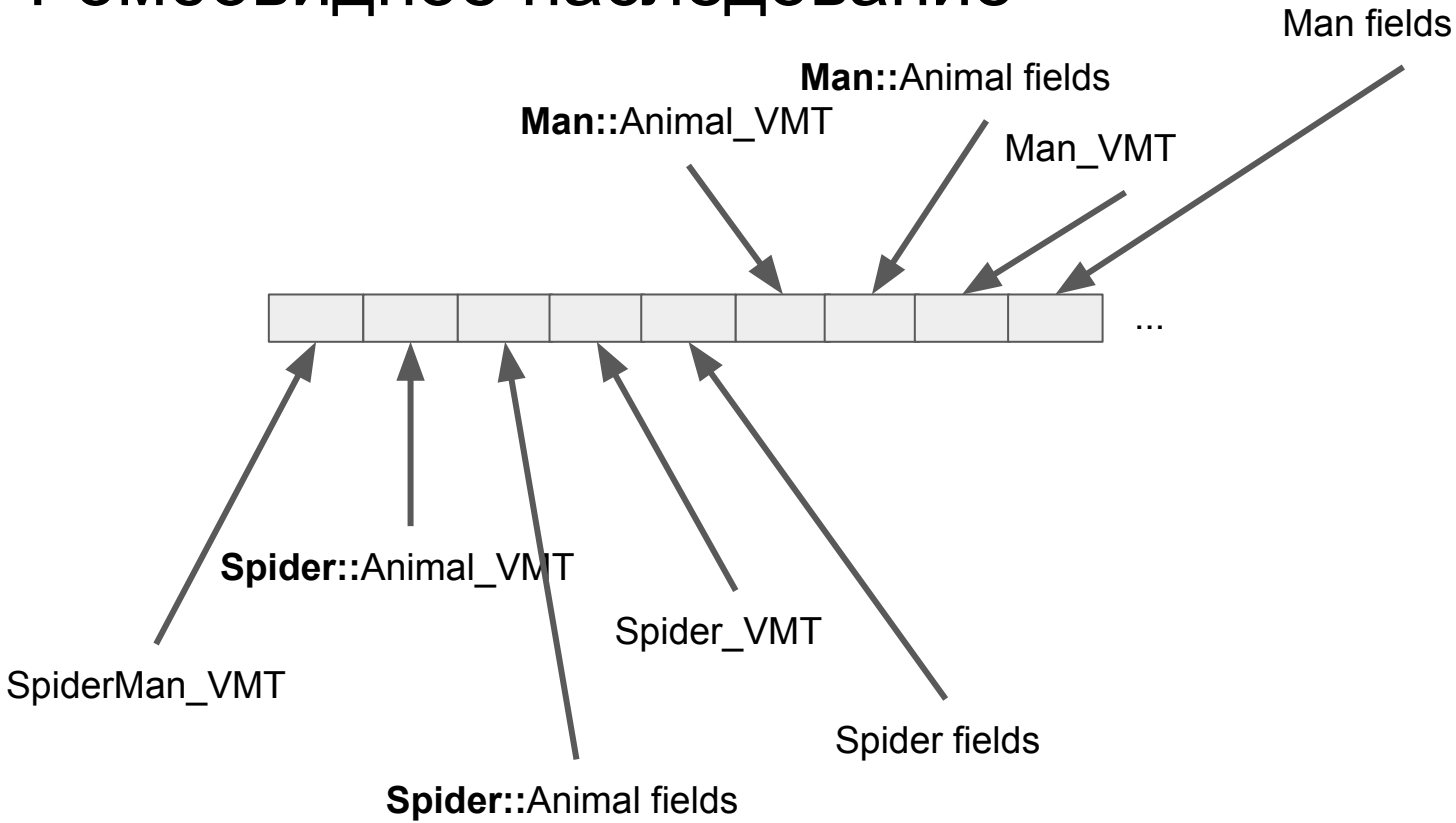
Ромбовидное наследование



Ромбовидное наследование



Ромбовидное наследование



Ромбовидное наследование

1) При ромбовидном наследовании подобъект общего предка дублируется

2) В таком случае каждый базовый подобъект является полноценным внутри объекта класса наследника

3) Это может соответствовать (а может и нет) семантике классов

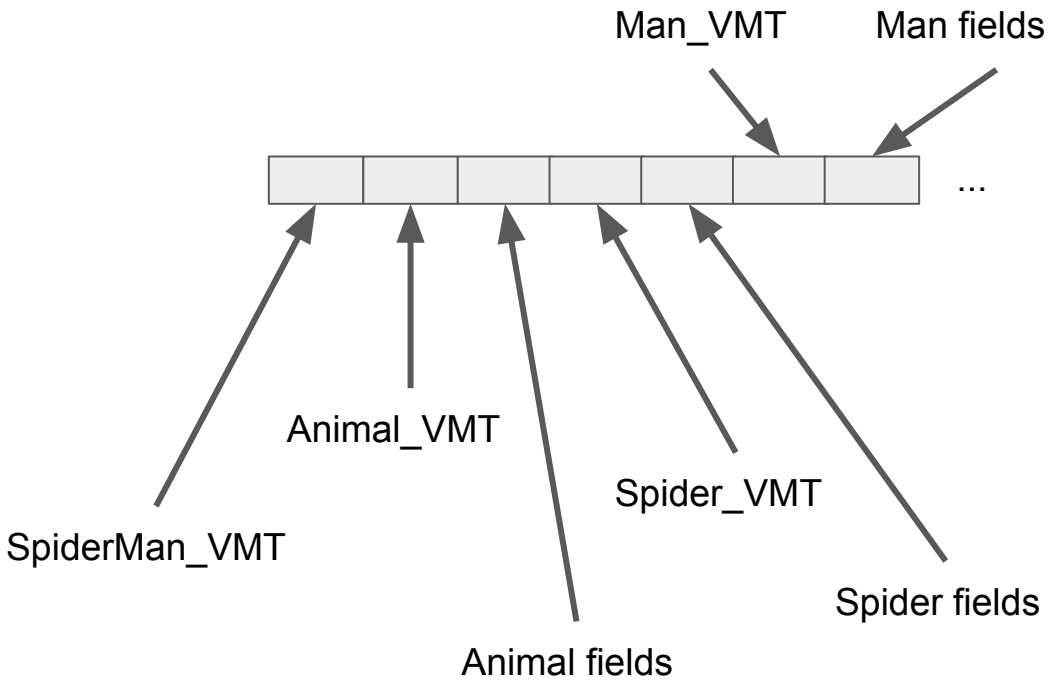
Виртуальное наследование

```
class Animal {
public:
        const char* name;
        virtual void speak() = 0;
};
```

```
class Man: virtual public Animal {
public:
        virtual void speak() {
                cout << “My name is ” << name;
        };
};
```

```
class Spider: virtual public Animal {
public:
        virtual void speak() {
                cout << “Shhhh”;
        };
};
```

Ромбовидное наследование



Виртуальное ромбовидное наследование

1) Подобъект общего предка разделяется между всеми

2) Виртуальные вызовы усложняются ещё сильнее

3) Это может соответствовать (а может и нет) семантике классов

За что мне всё это?

За что мне всё это?

Есть предубеждение: множественное наследование - это зло

За что мне всё это?

Есть предубеждение: множественное наследование - это зло

Это действительно так.

За что мне всё это?

Есть предубеждение: множественное наследование - это зло

Это действительно так. Для компиляторщиков.

За что мне всё это?

Есть предубеждение: множественное наследование - это зло

Это действительно так. Для компиляторщиков.

Для программиста множественное наследование - это мощный инструмент, вы должны уметь им пользоваться

За что мне всё это?

С большой силой появляется большая ответственность

Ошибка проектирования (пример с полем name)

Правильное проектирование:

1) Сделать класс Nameable
2) Виртуально наследовать его в Spider и Man
3) В ромбовидном наследовании у SpiderMan поле name будет присутствовать в одном экземпляре

Пример ромбовидного невиртуального наследования

У классов Student и Teacher можно выделить общего предка, в котором собрана логика выплаты стипендии/зарплаты

Тогда у класса Aspirant, наследующего и Student и Teacher, этот предок должен быть представлен дважды

Убедитесь, что вынесли с этой лекции

Множественное наследование

Каст указателя при множественном наследовании

Конфликты имён, способы разрешения

Переопределение полей и методов

Раскладка объектов в памяти при множественном наследовании

Виртуальные вызовы при множественном наследовании

Ромбовидное наследование, виртуальное ромбовидное наследование

Проверочные вопросы

1) Сколько в одном объекте может быть полей с одинаковыми именами?
2) Как к ним обращаться?
3) Что происходит с указателем при касте от класса-наследника к базовому классу при множественном наследовании?
4) Что происходит при касте в обратную сторону?
5) Чем отличается ромбовидное наследование от виртуального ромбовидного?
6) Что лучше?

* - выполните упражнение на слайде №75

Q & A