Множественное наследование, C++-касты и RTTI

Александр Смаль

CS центр
22 марта 2017
Санкт-Петербург

Множественное наследование

Множественное наследование (multiple inheritance) — возможность наследовать сразу несколько классов.

```
struct Student {
    string name()       const { return name_; }
    string university() const { return university_; }
private:
    string name_, university_;
};
struct FullTimeEmployee {
    string name()    const { return name_; }
    string company() const { return company_; }
private:
    string name_, company_;
};
struct BadStudent: Student, FullTimeEmployee {
    string name() const { return Student::name(); }
};
```

Интерфейсы

```
struct Person {
    string name() const { return name_; }
    string name_;
};
struct IStudent {
    virtual string name()       const = 0;
    virtual string university() const = 0;
    virtual ~IStudent() {}
};
struct IFullTimeEmployee {
    virtual string name()    const = 0;
    virtual string company() const = 0;
    virtual ~IFullTimeEmployee() {}
};
struct BadStudent: Person, IStudent, IFullTimeEmployee {
    string name()       const { return Person::name(); }
    string university() const { return university_; }
    string company()    const { return company_; }
    string university_, company_;
};
```

Переопределение одинаковых функций

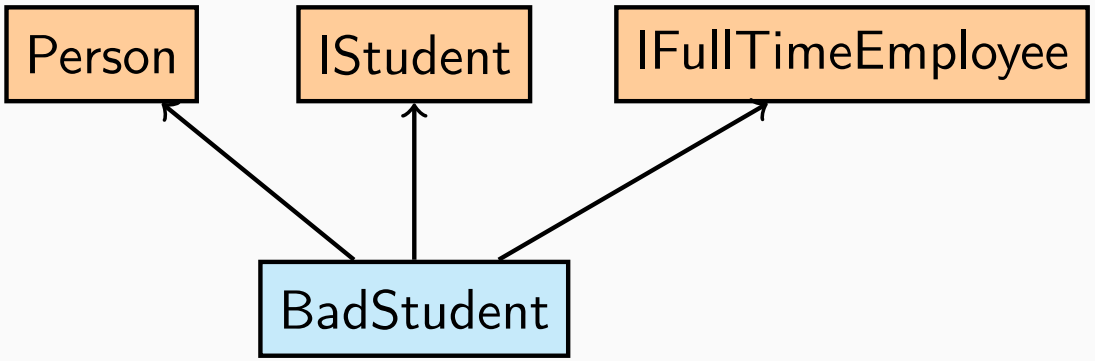
```
struct IStudent {
    virtual string name()       const = 0;
    virtual string university() const = 0;
    virtual ~IStudent() {}
};

struct IPlayer {
    virtual string name()       const = 0;
    virtual ~IPlayer() {}
};

struct TypicalStudent: Person, IStudent, IPlayer {
    ...
    string name()       const { return Person::name(); }
    ...
};
```

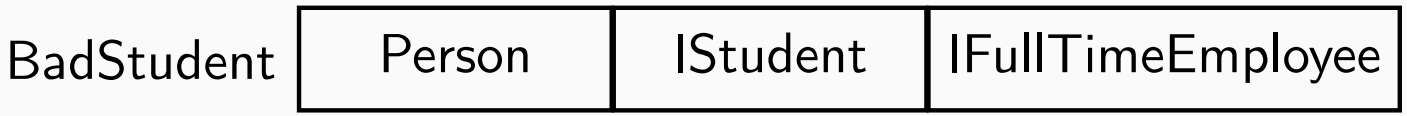
Переопределение одинаковых функций

```
struct IStudent {
    virtual string name()       const = 0;
    virtual string university() const = 0;
    virtual ~IStudent() {}
};
struct IStudentX : IStudent {
    string name() const { return studentName(); }
    virtual string studentName() const = 0;
};
struct IPlayer {
    virtual string name()       const = 0;
    virtual ~IPlayer() {}
};
struct IPlayerX : IPlayer {
    string name() const { return playerName(); }
    virtual string playerName() const = 0;
};
struct TypicalStudent: Person, IStudentX, IPlayerX {
    string studentName() const { return Person::name(); }
    string playerName() const  { return "DarkEvil666"; }
};
```

Представление в памяти

Во многих языках множественное наследование заменяется возможностью реализовывать интерфейсы.





Важно: помните про преобразование указателей.

Создание и удаление объекта

```
struct A { };

struct B : A { };

struct C : B { };

struct D { };

struct E : A, D { };

struct F : C, D, E { };
```

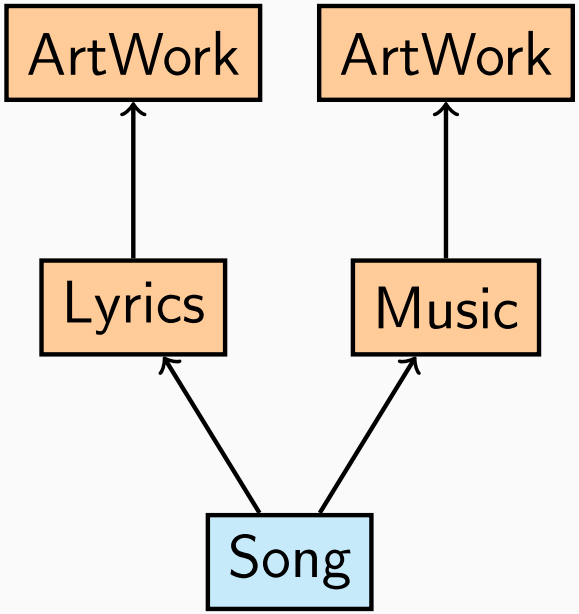
Порядок вызова конструкторов: `A`, `B`, `C`, `D`, `A`, `D`, `E`, `F`.

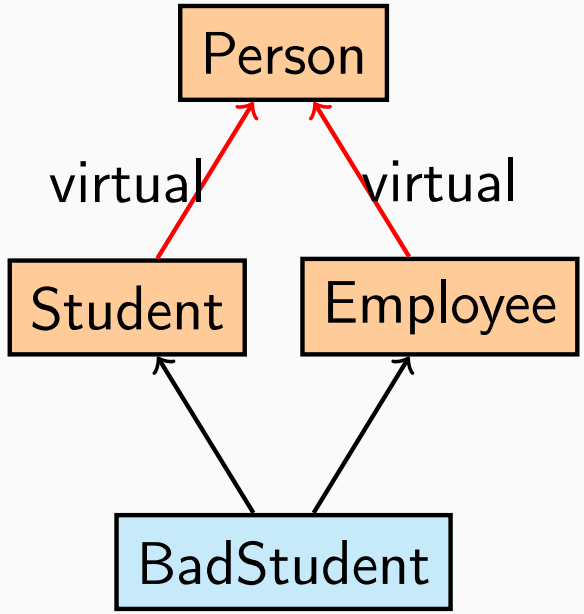
Деструкторы вызываются в обратном порядке.

Проблемы:

1. Дублирование `A` и `D`.
2. Недоступность первого `D`.

Виртуальное наследование






```
struct Person {};
struct Student   : virtual Person {};
struct Employee : virtual Person {};
struct BadStudent : Student, Employee {};
```

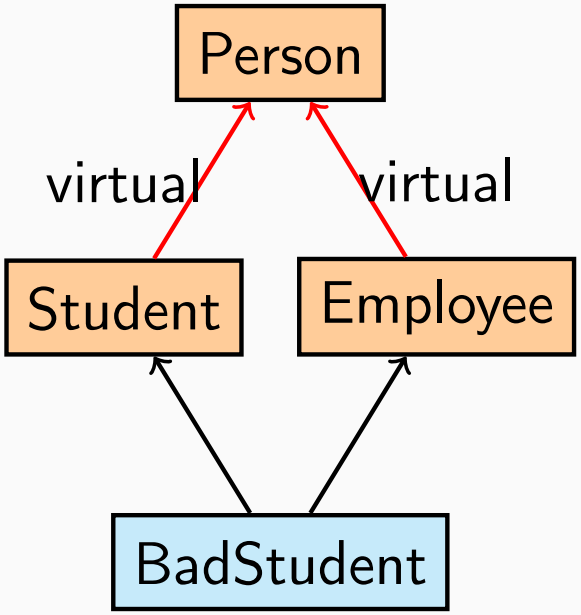
Виртуальное наследование: вопросы

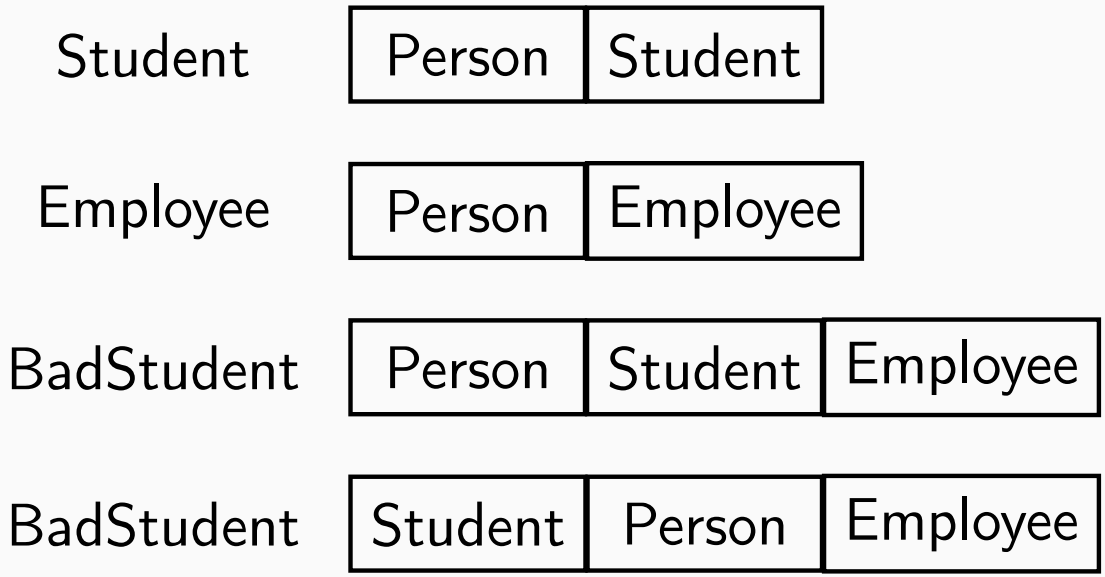
Кто вызывает конструктор базового класса?

```
struct Person {
    explicit Person(string const& name)
        : name_(name) {}
    string name_;
};
struct Student  : virtual Person {
    explicit Student(string const& name) : Person(name) {}
};
struct Employee : virtual Person {
    explicit Employee(string const& name) : Person(name) {}
};
struct BadStudent : Student, Employee {
    explicit BadStudent(string const& name)
        : Person(name), Student(name), Employee(name)
    {}
};
```

Виртуальное наследование: вопросы

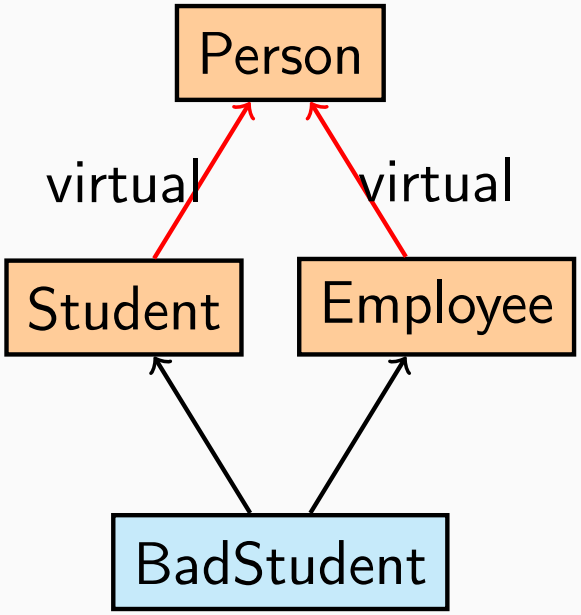
Как устроено расположение в памяти?

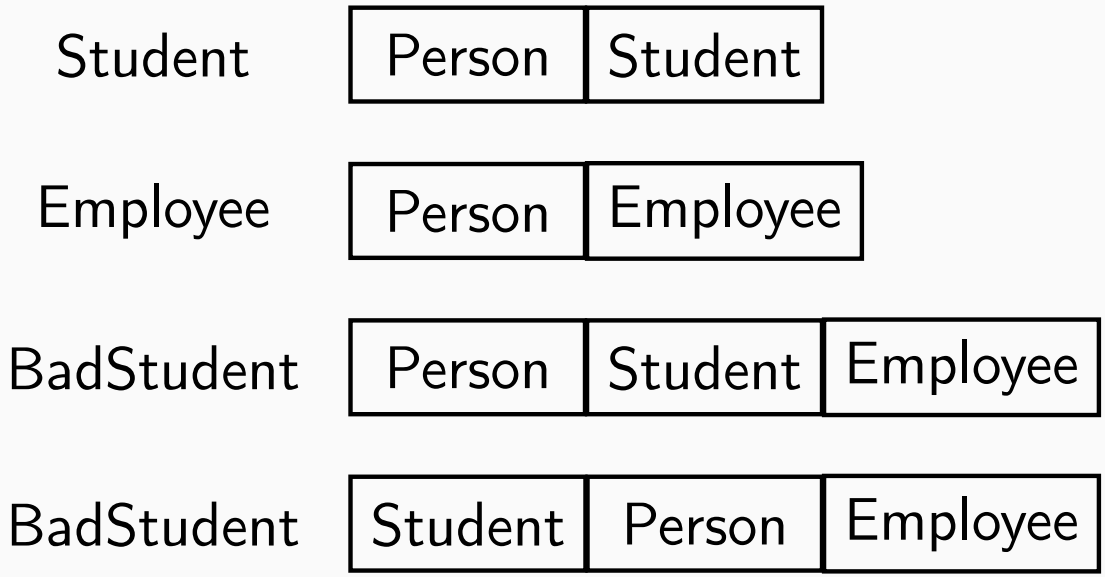




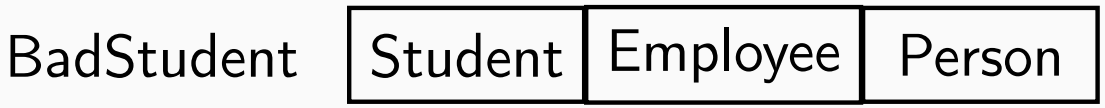
Виртуальное наследование: вопросы

Как устроено расположение в памяти?





На самом деле как-то так.



Виртуальное наследование: вопросы

Доступ через таблицу виртуальных методов

```
struct Person {
    string name;
};
struct Student   : virtual Person { };
struct Employee : virtual Person { };
struct BadStudent : Student, Employee { };

int main {
    BadStudent bs;
    string name = bs.name;
    // на самом деле
    string name = bs.__getPerson()->name;
}
```

Заключение

1. Не используйте множественное наследование для наследования реализации.
2. Используйте интерфейсы.
3. Хорошо подумайте перед тем, как использовать виртуальное наследование.
4. Помните о неприятностях, связанных с множественным наследованием.
5. Помните о неприятностях, связанных с виртуальным наследование.

C-style cast

Стандартный способ приведения типов в C.

```
int a = 10;
int b = 3;

double d = ((double)a) / b + 3.5;
d = int(d);

double * m = (double *) malloc(sizeof(double) * 100);
m[0] = 10.5;

char * mc = (char *)m;
mc[4] = 23;
```

В C преобразует арифметические типы и указатели.

Что делает в C++?

Преобразования в C++: static_cast

Служит для преобразований связанных типов:

- Стандартные преобразования.

```
double d = static_cast<double>(10) / 3 + 3.5;
d = static_cast<int>(d);
```

- Явное (пользовательское) приведение типа:

```
T t = static_cast<T>(e); // T t(e);
```

- Обратные варианты стандартных преобразований:
  - целочисленные типы в перечисляемые,
  - `Base *` в `Derived *` (downcast),
  - `T Base::*` в T Derived::*,
  - `void *` в любой T *
- Преобразование к `void`.

```
static_cast<void>(5);
```

Преобразования в C++: const_cast

Служит для снятия/добавления константности:

```
void f(double const& d) {
    const_cast<double &>(d) = 10;
}
```

Использование const_cast — признак плохого дизайна.

Преобразования в C++: const_cast

Служит для снятия/добавления константности:

```
void f(double const& d) {
    const_cast<double &>(d) = 10;
}
```

Использование const_cast — признак плохого дизайна.

Кроме некоторых исключений:

```
T & operator[](size_t i) {
    return  const_cast<T &>(
            const_cast<Vector const &>(*this)[i]);
}

T const & operator[](size_t i) const {
    assert(i < size_);
    return data_[i]
}
```

Преобразования в C++: reinterpret_cast

Служит для преобразований несвязанных указателей.

```
void send(char const * data, size_t length);
char * recv(size_t * length);

double * m = static_cast<double *>(malloc(sizeof(double) * 100));
...
char * mc = reinterpret_cast<char *>(m);
send(mc, sizeof(double) * 100);

// other side
size_t l = 0;
double * r = reinterpret_cast<double *>(recv(&l));
l /= sizeof(double);
```

Границы применимости C-style cast

C-style cast может вызвать любое из преобразований:
`static_cast`, `reinterpret_cast`, `const_cast`.
Можно использовать:

- преобразование встроенных типов,
- преобразование указателей на явные типы.

Не стоит использовать:

- в шаблонах,
- для преобразования пользовательских типов и указателей на них.

Когда C-style cast приводит к ошибке

```
struct A;
struct B;
struct C;

C * f(B * b) {
    return (C *)b;     // reinterpret_cast
    // return static_cast<C *>(b); doesn't compile
}

struct A {
    int a;
};

struct B {};

struct C : A, B
{};
```

Run-time type information

В C++ есть механизм получения информации о типах времени выполнения.
Состоит из двух компонент:

1. `type_info` и `typeid`
2. `dynamic_cast`

`type_info`

- Класс объявленный в `<typeinfo>`.
- Методы: `==`, `!=`, `name`, `before` (т.е. не копируется).
- Можно получить `type_info` при помощи оператора `typeid`.
- `typeid` от нулевого указателя бросает `bad_typeid`.

Использование type_info

```
struct A {
    virtual ~A() { }
};

struct B : A { };

int main() {
    B b;
    A* ap = &b;
    A& ar = b;
    cout << typeid(*ap).name() << endl; // B
    cout << typeid(ar) .name() << endl; // B
    cout << typeid(ap) .name() << endl; // A *
    cout << typeid(A *).name() << endl; // A *
    cout << (typeid(ar) == typeid(B)) << endl; // 1
}
```

Преобразования в C++: dynamic_cast

Позволяет делать преобразования с проверкой типа времени выполнения.

```
A * a = (rand() % 2) ? new B() : new C();
if(B * b = dynamic_cast<B *>(a))
    ...
else if (C * c = dynamic_cast<C *>(a))
    ...
```

Особенности:

- Не заменяется преобразованием в стиле C.
- Требует наличие виртуальных функций (полиморфность).
- При приведении к ссылке кидает исключение bad_cast.
- При приведении к указателю может вернуть 0.

Что возвращает dynamic_cast<void *>(a)?

Почему следует избегать RTTI?

Пример: double dispatch

```
struct Triangle; struct Rectangle; struct Circle;
struct Shape {
    virtual ~Shape() {}
    virtual bool intersect( Rectangle * r ) = 0;
    virtual bool intersect( Triangle  * t ) = 0;
    virtual bool intersect( Circle    * c ) = 0;
    virtual bool intersect( Shape     * s ) = 0;
};
struct Triangle : Shape {
    bool intersect( Rectangle * r ) { ... }
    bool intersect( Triangle  * t ) { ... }
    bool intersect( Circle    * c ) { ... }
    bool intersect( Shape     * s ) {
        return s->intersect(this);
    }
};
bool intersect(Shape * a, Shape * b) { return a->intersect(b); }
```