Виртуальные методы и виртуальное наследование

Гусев Илья

Московский физико-технический институт

Москва, 2018

Содержание

Зачем нам виртуальные методы?

Игрушечный пример:

```
class Animal{
public:
    string do_sound(){ return "WTF"; }
};
class Dog: public Animal{
public:
    string do_sound(){ return "Woof!"; }
};
class Cat: public Animal{
public:
    string do_sound(){ return "Meow!"; }
};
```

Хочется собрать всех животных и заставить их издавать звуки, присущие конкретному животному. Но мы не можем положить разнородные объекты в массив. Давайте положим их указатели на Animal!

Зачем нам виртуальные методы?

```
#include <vector>
using namespace std;

int main(){
    B dog;
    C cat;
    vector<A*> animals;
    animals.push_back(&B);
    animals.push_back(&C);
    for( A* ptr: animals ) {
        cout<<ptr->do_sound()<<" ";
    }
}
```

Вывод: "WTF WTF".
А как нам например, обрабатывать коллекцию окон в ОС? Использовать сложные иерархии?

Зачем нам виртуальные методы?

```
class Animal{
public:
    virtual string do_sound(){ return "WTF"; }
};
class Dog: public Animal{
public:
    string do_sound(){ return "Woof!"; }
};
class Cat: public Animal{
public:
    string do_sound(){ return "Meow!"; }
};
```

Тот же main.
Вывод: "Woof! Meow!".

Зачем нам виртуальные методы?[2]

Старый код вызывает новый код.

```
class Animal{
public:
    string say() { return do_sound()+do_sound(); }
    virtual string do_sound(){ return "WTF"; }
};
class Dog: public Animal{
public:
    string do_sound(){ return "Woof!"; }
};
class Cat: public Animal{
public:
    string do_sound(){ return "Meow!"; }
};
```

Виртуальные методы

1. Синтаксис - ключевое слово virtual перед методом базового класса.
2. virtual у методов производных классов опционален, но лучше писать для понятности.
3. Чисто виртуальные функции: virtual void foo() = 0;
4. Если функция виртуальная, то какая именно функция определяется в runtime, а не на этапе компиляции.
5. Если у класса есть хотя бы одна виртуальная функция, деструктор тоже следует сделать виртуальным. Почему?
6. Преобразования по иерархии через dynamic_cast (для безопасности).

Виртуальный деструктор

```
class Base {
public:
    ~Base();
};

class Derived : public Base {
public:
    ~Derived();
};

void f()
{
    Base* p = new Derived;
    delete p;  // деструктор Derived не вызывается
}
```

А если в деструкторе происходило освобождение ресурсов (закрытие файлового десриптора, особождение по указателям полей)?

Как это работает

1. Полностью определяется компилятором!
2. Всегда есть v-table.
3. Рассмотрим на примере

Как это работает

Такой класс:

```
class Base {
public:
  virtual int virt0();
  virtual int virt1();
  virtual int virt2();
  virtual int virt3();
};
```

Как это работает

Шаг 1: компилятор создаёт статическую таблицу с указателями на функции.

```
// Псевдокод для статической таблицы в Base.cpp
// Притворимся, что FunctionPtr - это указатель на метод
FunctionPtr Base::__vtable[4] = {
  &Base::virt0, &Base::virt1, &Base::virt2, &Base::virt3
};
```

Шаг 2: компилятор добавляет скрытый указатель (обычно void*) к каждому объекту класса Base. Этот указатель обычно называют v-pointer. Можно представлять, что компилятор переписывает класс примерно так:

```
class Base {
public:
  // ...
  FunctionPtr* __vptr;  // подставляется компилятором, скрыт
  // ...
};
```

Как это работает

Шаг 3: компилятор инициализирует this->__vptr в каждом конструкторе. Можно представлять, что в списко ицниициализации добаляется привязка к v-table:

```
Base::Base()
  : __vptr(&Base::__vtable[0])   // подставляется компилятором
  // ...
{
  // ...
}
```

Как это работает

Для производного класса выполняются шаги 1 и 3 (но не 2). Предположим, что Derived переопределяет virt0 и virt1. Тогда:

```
FunctionPtr Der::__vtable[5] = {
  &Der::virt0, &Der::virt1, &Base::virt2, &Base::virt3
};
```

В итоге этот код:

```
void mycode(Base* p)
{
  p->virt3();
}
```

Трансофрмируется в этот (псевдокод):

```
void mycode(Base* p)
{
  p->__vptr[3](p);
}
```

Пример реального расположения для g++

```
class A{
public:
    virtual void doSomeWork();
};

class B : public A{
public:
    virtual void doSomeWork();
};
```

Пример реального расположения для g++

Команда: g++ -fdump-class-hierarchy example.h

```
Vtable for A
A::_ZTV1A: 3u entries
0     (int (*)(...))0
8     (int (*)(...))(& _ZTI1A)
16    (int (*)(...))A::
    doSomeWork

Class A
   size=8 align=8
   base size=8 base align=8
A (0x7fb76785a4) 0
    vptr=((& A::_ZTV1A) + 16u)
```

```
Vtable for B
B::_ZTV1B: 3u entries
0     (int (*)(...))0
8     (int (*)(...))(& _ZTI1B)
16    (int (*)(...))B::
    doSomeWork

Class B
   size=8 align=8
   base size=8 base align=8
B (0x7fb7678510) 0
    vptr=((& B::_ZTV1B) + 16u)
  A (0x7fb76785a540) 0
      primary-for B (0
          x7fb7678510)
```

Обычное наследование - контрольные вопросы

1. Чем struct отличается от class по стандарту?
2. А по смыслу?
3. Зачем нужно наследование?
4. Типы наследования.
5. Зачем нужно приватное наследование, примеры использвоания
6. Композиция vs наследование
7. friend - что это?
8. Концепция интерфейсов.
9. ABC

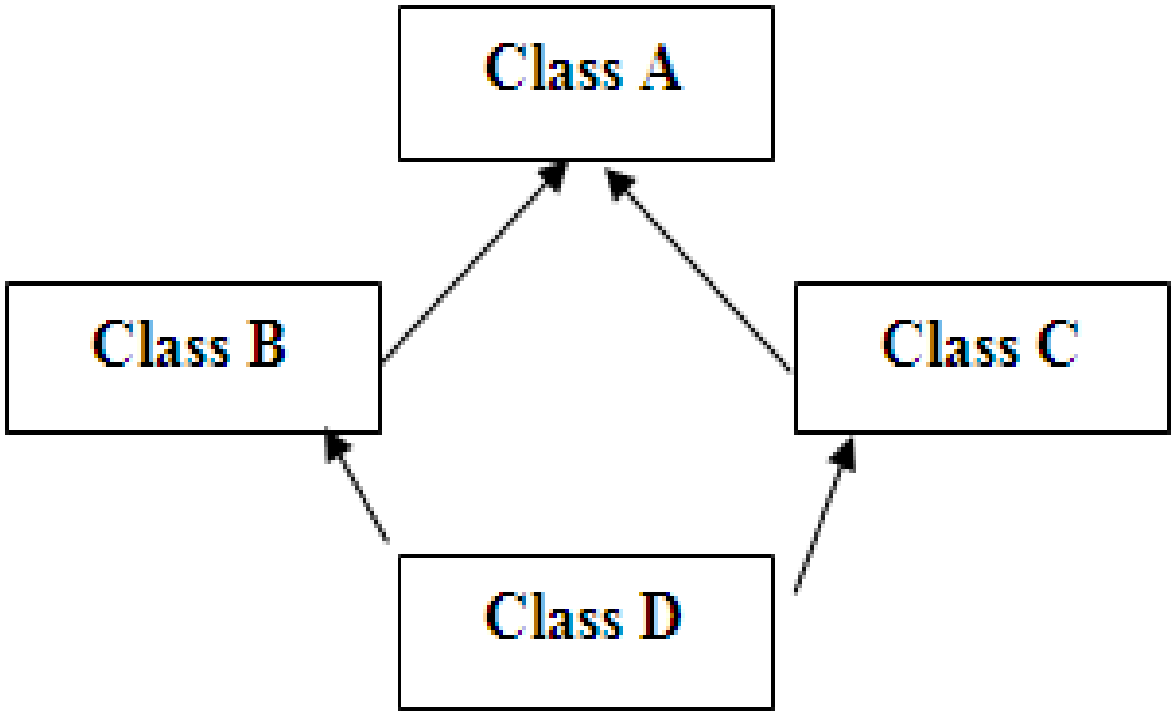
Обычное наследование

B наследуется от A

```
struct A {
    int a;

    void foo(){}
};

struct B: A {
    int b;

    void foo(){}
};

sizeof(int) == 4;
sizeof(A) == ?;
sizeof(B) == ?;

B objectB;
```

Как вызвать foo() из A?

Множественное наследование



Множественное наследование

```
struct A {
    int a = 4;
    void foo(){}
};

struct B: public A {
    int b = 5;
    void foo(){}
};

struct C: public A {
    int c = 6;
    void foo() {}
};

struct D: public B, public C {
    int d = 7;
    void foo() {}
};
```

Множественное наследование

```
sizeof(D) == ?

int GetShiftedInt(int& a, int n) {
    return *((int*)(&a) - n);
}

int main(){
    D objectD;
    int& d = objectD.d;
    cout<<d<<" "<<GetShiftedInt(d, 1)<<" "<<GetShiftedInt(d, 2)<<
    " "<<GetShiftedInt(d, 3)<<" "<<GetShiftedInt(d, 4)<<endl;
}
```

Множественное виртуальное наследование

```
struct A {
    int a = 4;
    void foo(){}
};

struct B: public virtual A {
    int b = 5;
    void foo(){}
};

struct C: public virtual A {
    int c = 6;
    void foo() {}
};

struct D: public B, public C {
    int d = 7;
    void foo() {}
};
```

Множественное виртуальное наследование

```
sizeof(D) == ?
```

Виртуальное наследование - особенности

1. Ни в коем случае не использовать C-style cast, вместо этого - dynamic_cast
2. Внимательнее с порядком вызова конструкторов. В целом - сверху вниз, слева направо в графе наследования.

Полезные ссылки I

C++ Super-FAQ
https://isocpp.org/wiki/faq/virtual-functions

SO - Virtual Table layout in memory?
https://stackoverflow.com/questions/1342126/virtual-table-layout

g++ object model
http://spockwangs.github.io/2011/01/31/cpp-object-model.html

Цикл статей по устройству vtable в clang
https://shaharmike.com/cpp/vtable-part1/