C/C++: Лекция 4

Воробьев Д.В

25.09.2020

ООП

Наследование продолжение

Dimond problem

Было

```
struct A {
    int x = 10;
};

struct B : A {};

struct C : A {};

struct D : B, C {};

int main() {
    D d;
    // CE
    std::cout << d.x;
    return 0;
}
```

Стало

```
struct A {
    int x = 10;
};

struct B : virtual A {};

struct C : virtual A {};

struct D : B, C {};

int main() {
    D d;
    // A::x в ед-ом экземпляре
    std::cout << d.x;
    return 0;
}
```

Виртуальное наследование это не полиморфизм

```cpp
#include <type_traits>

struct Base {};

struct Derived : virtual Base {};

int main() {
    // 0
    std::cout << std::is_polymorphic<Derived>::value;
}
```

Полиморфизм

Полиморфизм

Полиморфный класс

Класс определяющий метод virtual

```
// полиморфный
struct A {
    virtual void foo();
};

int main() {
    return 0;
}
```

Полиморфизм

В дочерних ключевое слово virtual указывать не обязательно

```
// полиморфный
struct A {
    virtual void foo() { std::cout << "1"; }
};

struct B : A {
    void foo() {std::cout << "2";}
};

int main() {
    B b;
    // 2
    b.foo();
    return 0;
}
```

При вызове вызывается функция класса первого ближайшего к вызывающему

```cpp
struct A {
    virtual void foo() { std::cout << "1"; }
};

struct B : A {
    void foo() {std::cout << "2";}
};

struct C : B {};

int main() {
    A* ptr = new c;
    // 2
    ptr->foo();
    return 0;
}
```

Полиморфизм

Виртуальные методы проявляют свое свойство только при использовании указателя на базовый класс

Полиморфный класс

```
struct Base {
    virtual void foo() {
        std::cout << "Base";
    }
};

struct Derived : Base {
    virtual void foo() {
        std::cout << "Derived";
    }
};

int main() {
    Derived d;
    // Derived
    d.foo();
    return 0;
}
```

Не полиморфный класс

```
struct Base {
    void foo() {
        std::cout << "Base";
    }
};

struct Derived : Base {
    void foo() {
        std::cout << "Derived";
    }
};

int main() {
    Derived d;
    // Derived
    d.foo();
    return 0;
}
```

Полиморфный класс

```
struct Base {
    virtual void foo() {
        std::cout << "Base";
    }
};

struct Derived : Base {
    virtual void foo() {
        std::cout << "Derived";
    }
};

int main() {
    Base* ptr = new Derived;
    // Derived
    ptr->foo();
    return 0;
}
```

Не полиморфный класс

```
struct Base {
    void foo() {
        std::cout << "Base";
    }
};

struct Derived : Base {
    void foo() {
        std::cout << "Derived";
    }
};

int main() {
    Base* ptr = new Derived;
    // Base
    ptr->foo();
    return 0;
}
```

Виртуальные функциии и аргументы по умолчанию

- Функции, переопределющие виртуальную, (overriders) не перенимают аргументы по умолчанию из функции базы
- При вызове аргументы по умолчанию = аргументы по умолчанию в declaration функции у static type объекта

Виртуальные функции и аргументы по умолчанию

```
struct Base {
    virtual void f(int a=7) {
        std::cout << "Base:" << a;
    }
};

struct Derived : Base {
    void f(int a) {
        std::cout << "Derived:" << a;
    }
};

int main() {
    Derived d;
    Base& b = d;
    // static_type : Base, Base по умолч. a=7 => f(7)
    b.f();
    return 0;
}
```

Virtual method table

VMT

Таблица виртуальных функций - это массив указателей на функции вышестоящих классов

Полиморфный класс хранит указатель на таблицу виртуальных функций

VMT: влияние на размер

Полиморфный класс

```
struct Base {
    virtual void foo();
};

int main() {
    // 8
    std::cout << sizeof(Base);
    return 0;
}
```

Не полиморфный класс

```
struct Base {
    void foo();
};

int main() {
    // 1
    std::cout << sizeof(Base);
    return 0;
}
```

VMT: влияние на размер

Полиморфный класс

```
struct Base1 {
    virtual void foo();
};

struct Base2 {
    virtual void foo();
};

struct Derived : Base1, Base2 {};

int main() {
    // 16 : 2 указателя
    std::cout << sizeof(Derived);
    return 0;
}
```

Не полиморфный класс

```
struct Base1 {
     void foo();
};

struct Base2 {
     void foo();
};

struct Derived : Base1, Base2 {};

int main() {
    // 1
    std::cout << sizeof(Derived);
    return 0;
}
```

VMT: содержимое

```
class Base {
public:
    FunctionPointer *__vptr;
    virtual void function1() {};
    virtual void function2() {};
};

class D1: public Base {
public:
    virtual void function1() {};
};

class D2: public Base {
public:
    virtual void function2() {};
};
```

VMT: содержимое

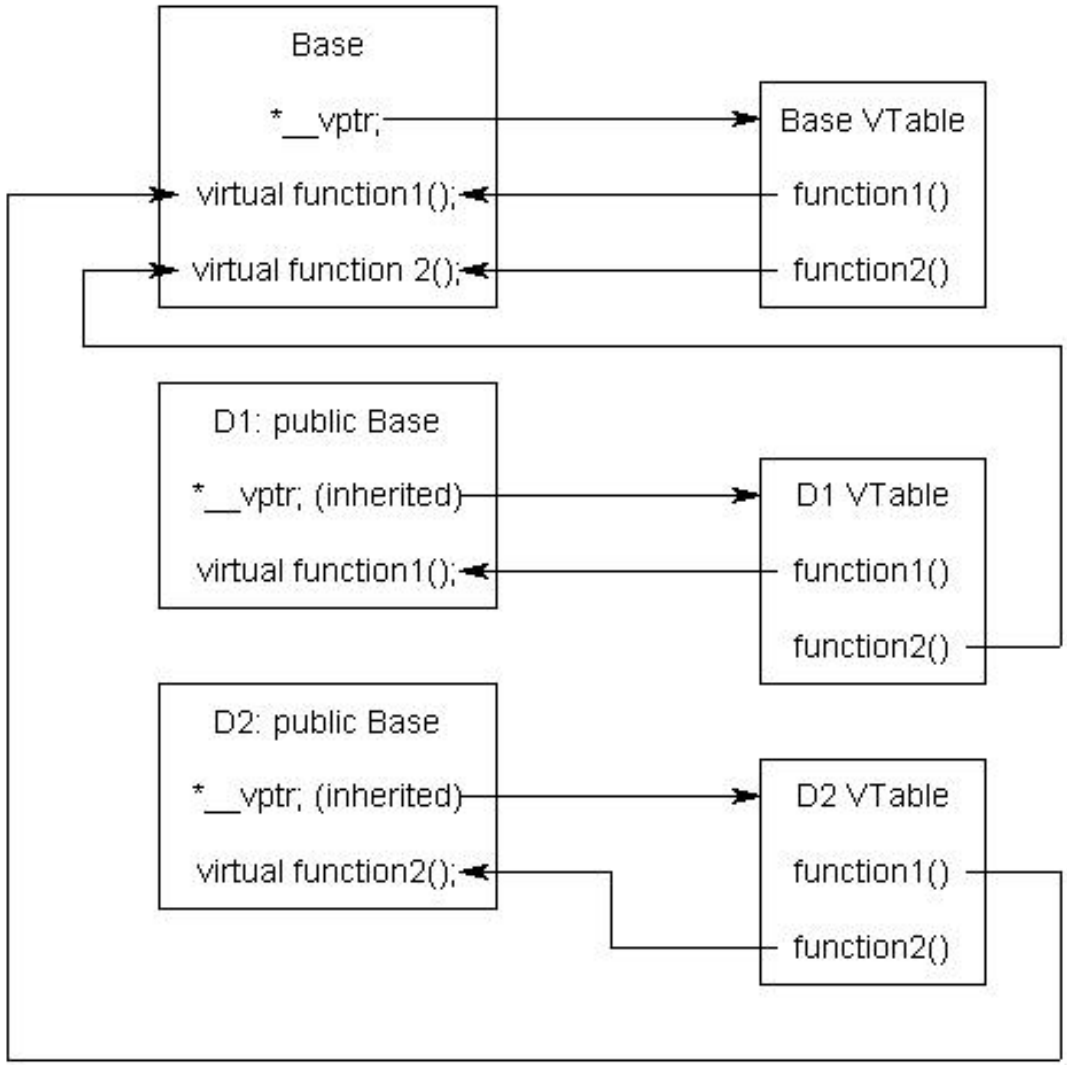

Figure: Расстановка указателей

Абстрактный класс

Чисто виртуальная функция

Функция специфицированная = 0

Абстрактный класс

Класс, определяющий чисто виртуальную функцию

```
struct Base {
    virtual void foo() = 0;
};

int main() {
    return 0;
}
```

Абстрактный класс

Реализацию методов делать нельзя

```
struct Base {
    // CE
    virtual void foo() = 0 {
        std::cout << "implementation";
    }
};
```

Абстрактный класс

Экземпляры создавать нельзя

```
struct Base {
    virtual void foo() = 0;
};

int main() {
    // CE
    Base b;
    return 0;
}
```

Абстрактный класс

Назначение

Надиктовывать интерфейс иерархии

```
struct Base {
    virtual void foo() = 0;
};
struct Derived : Base {}

int main() {
    // CE
    Derived d;
    return 0;
}
```

Виртуальный деструктор

Проблема

```
struct Base {
    int* x;
    Base() { x = new int(1); }
    ~Base() {
        std::cout << "~Base";
        delete x;
    }
};

struct Derived : Base {};

int main() {
    Base* ptr = new Derived;
    //
    delete ptr;
    return 0;
}
```

Решение

```
struct Base {
    int* x;
    Base() { x = new int(1); }
    virtual ~Base() {
        std::cout << "~Base";
        delete x;
    }
};

struct Derived : Base {};

int main() {
    Base* ptr = new Derived;
    //~Base
    delete ptr;
    return 0;
}
```

override

- Демонстриует компилятору, что данный метод будет переопределять виртуальную функцию.
- При чтении кода дает определенность, что данная функция виртуальная.

override

Находясь в DGSLEffectFactory понимаем, что CreateEffect виртуальная

```
class DGSLEffectFactory : public IEffectFactory
{
public:
    explicit DGSLEffectFactory(_In_ ID3D11Device* device);
    DGSLEffectFactory(DGSLEffectFactory&& moveFrom) noexcept;
    DGSLEffectFactory& operator= (DGSLEffectFactory&& moveFrom) noexcept;

    DGSLEffectFactory(DGSLEffectFactory const&) = delete;
    DGSLEffectFactory& operator= (DGSLEffectFactory const&) = delete;

    ~DGSLEffectFactory() override;

    // IEffectFactory methods.
    std::shared_ptr<IEffect> __cdecl CreateEffect(_In_ const EffectInfo& info, _In_opt_ ID3D11DeviceContext* deviceContext) override;
```

Операторы приведения типов в применении к наследованию

static_cast

```
class Base {};

class Derived : public Base {
    public:
        int x = 10;
};

int main() {
    Base b;
    // UB
    static_cast<Derived&>(b).x;
    return 0;
}
```

reinterpret_cast: доступ к недоступному

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

class Derived : private Base {};

int main() {
    Derived d;
    // =)
    reinterpret_cast<Base&>(d).foo();
    return 0;
}
```

reinterpret_cast: доступ к недоступному

```
class Base {
    int x = 10;
};

class Derived : Base {};

int main() {
    Derived* b = new Derived;
    int* x = reinterpret_cast<int*>(b);
    std::cout << *x;
    return 0;
}
```

dynamic_cast

Факт

- оператор приведения типов
- делает проверку на корректность в RunTime
- класс должен быть полиморфным

```cpp
struct Base {
   virtual void foo() {}
};

struct Derived : Base {};

int main() {
    Base* b = new Base;
    // 0
    std::cout << dynamic_cast<Derived*>(b);
    return 0;
}
```

```cpp
struct Base {
    void foo() { std::cout << "Base: foo"; }
};

struct Derived : Base {
    void bar() { std::cout << "Derived: bar"; }
    virtual void foo() { std::cout << "Derived: foo"; }
};

void baz(const Base& base) {
    // 2. хотим вызвать bar Derived
}

int main() {
    Derived d;
    // 1. передали Derived
    baz(d);
    return 0;
}
```

```cpp
struct Base {
    virtual void foo() { std::cout << "Base: foo"; }
};

struct Derived : Base {
    void bar() { std::cout << "Derived: bar"; }
    void foo() { std::cout << "Derived: foo"; }
};

void baz(Base& base) {
    Derived* d = dynamic_cast<Derived*>(&base);
    if( d != nullptr ) {
        d->bar();
    } else {
        // Base logic
    }
}

int main() {
    Derived d;
    baz(d);
    return 0;
```

dynamic_cast: стоимость

Предупреждение

dynamic_cast выполняется долго

Совет

Постарайтесь перепроектировать систему

```cpp
struct Base {
    void foo() const { std::cout << "Base: foo"; }
};

struct Derived : Base {
    void bar() const { std::cout << "Derived: bar"; }
    virtual void foo() const { std::cout << "Derived: foo"; }
};

void baz(const Base& base) {
    // Base logic
}

// перегрузили baz
void baz(const Derived& d) {
    d.bar();
}

int main() {
    Derived d;
    baz(d);
    return 0;
```

dynamic_cast: стоимость

Избегайте серии dynamic_cast.

```
class Base { ... };
class Derived1 : public Base {...};
class Derived2 : public Base {...};
class Derived3 : public Base {...};
...
for (...) {
    if (Derived1* p1  = dynamic_cast<Derived1*>(iter->get())) {
        ...
    }
    else if (Derived2* p2 = dynamic_cast<Derived2*>(iter->get())) {
        ...
    }
    else if (Derived3* p3 = dynamic_cast<Derived3*>(iter->get())) {
        ...
    }
}
```

dynamic_cast

```
struct Base {
    virtual void foo() { std::cout << "Base"; }
};

struct Derived : Base {
    void foo() { std::cout << "Derived"; }
};

void foo(Base& base) {
    // Derived
    dynamic_cast<Derived&>(base).foo();
}

int main() {
    Derived d;
    foo(d);
    return 0;
}
```

dynamic_cast: в production

```
class IEffectFactory {
public:
    virtual std::shared_ptr<IEffect> __cdecl CreateEffect(...) = 0;
    virtual void __cdecl CreateTexture(...) = 0;
};

class DGSLEffectFactory : public IEffectFactory {
public:
    std::shared_ptr<IEffect> __cdecl CreateEffect(...) override;
    void __cdecl CreateTexture(...) override;

    virtual std::shared_ptr<IEffect> __cdecl CreateDGSLEffect(...);
 }
auto fxFactoryDGSL = dynamic_cast<DGSLEffectFactory*>(&fxFactory);

m.effect = fxFactoryDGSL->CreateDGSLEffect(info, nullptr);
```