C/C++: Lecture 4

Vorobev D.V

25.09.2020

OOP

Inheritance

Dimond problem: virtual inheritance

Before

```cpp
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;
}
```

After

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

struct B : virtual A {};

struct C : virtual A {};

struct D : B, C {};

int main() {
    D d;
    // there is only one version o
    std::cout << d.x;
    return 0;
}
```

Virtual inheritance is not a dynamic polymorphism

```cpp
#include <type_traits>

struct Base {};

struct Derived : virtual Base {};

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

Polymorphism

Polymorphism

Polymorphic type

A class / struct type that declares or inherits at least one virtual function

```cpp
// polymorphic type
struct A {
    virtual void foo();
};

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

Polymorphism

There is no need to specify keyword virtual in derived classes for overridable function.

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

struct B : A {
    // B::foo is a virtual function
    void foo() {std::cout << "2";}
};
```

If the class does not override the function than the final overrider is called

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

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

struct C : B {};

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

Polymorphism

Virtual member functions demonstrate their behavior only with using the reference or pointer to the Base class.

Polymorphic type

```cpp
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;
}
```

Non-polymorphic type

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

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

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

Polymorphic type

```cpp
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;
}
```

Non-polymorphic type

```cpp
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;
}
```

Virtual member functions and default arguments

- The overriders of virtual functions do not acquire the default arguments from the base class declarations
- The default arguments are decided based on the static type of the object

Virtual member functions and default arguments

```
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, the default arg in the Base: a=7 => f(7)
    b.f();
    return 0;
}
```

Virtual method table

VMT

Virtual method table is an array of pointers to member functions of base classes

Polymorphic class implicitly stores a pointer to the VMT

VMT increases class size

Polymorphic type

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

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

Non-polymorphic type

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

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

VMT: increases class size

Polymorphic type

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

struct Base2 {
    virtual void foo();
};

struct Derived : Base1, Base2 {};

int main() {
    // 16 = 8 + 8: 2 pointers
    std::cout << sizeof(Derived);
    return 0;
}
```

Non-polymorphic type

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

struct Base2 {
     void foo();
};

struct Derived : Base1, Base2 {};

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

VMT: under the hood

```
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: under the hood

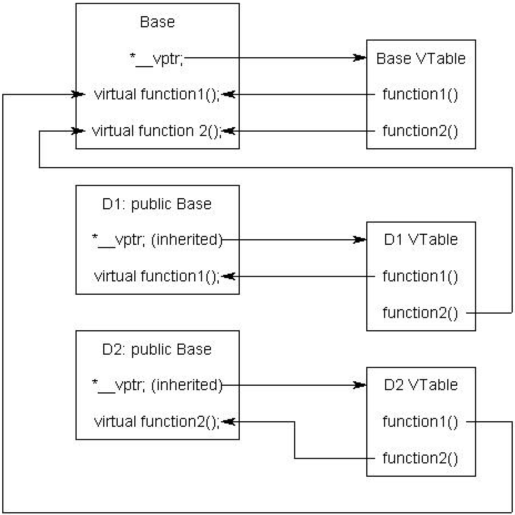

Figure: Vtables content

Abstract class

Pure virtual function

It is a virtual function specified with $= 0$

Abstract class

It is a class that declares with at least one pure virtual function

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

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

Abstract class

Pure virtual function can not be implemented

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

Abstract class

Abstract class cannot be instantiated

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

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

Abstract class

Usage

Abstract class dictates an interface

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

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

Virtual destructor

Problem

```cpp
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;
}
```

Solution

```cpp
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

- This specifier ensures that the function is overriding a virtual function from a base class.
- This specifier gives us a certainty that the function is virtual.

override: sample from open source

Looking at the interface of DGSLEffectFactory we instantly get confidence that Foo is a virtual function.

```
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;
```

The sample code from https://github.com/microsoft/DirectXTK

Explicit type conversion + Inheritance

static_cast

```cpp
class Base {};

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

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

dynamic_cast

Quick facts

- converts pointers and references
- checks correctness of the cast at RunTime
- can be applied only to polymorphic types

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

struct Derived : Base {};

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

Let's consider the following case

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

struct Derived : Base {
    void bar() { std::cout << "Derived: bar"; }
    void foo() override { std::cout << "Derived: foo"; }
};
void baz(const Base& base) {
    // 2. Here we want to call "bar"
}
int main() {
    Derived d;
    // 1. We pass the Derived
    baz(d);
    return 0;
}
```

To implement this, we can use dynamic_cast

```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: overhead

Note

dynamic_cast imposes significant overhead

Advice

Think about redesigning the architecture of your solution

```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: overhead

Avoid cascading of dynamic_casts.
(Scott Meyers: Effective C++ Third Edition)

```
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: production code

```
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);
```

The sample code from https://github.com/microsoft/DirectXTK