C++期末不挂科

CH4 - 面向对象2 - 继承与多态

本章内容

- 继承
- 虚函数
- 多态

理解以下名词：

- 继承
- 父类(基类)和子类(派生类)
- 公有、私有与保护继承
- 虚函数与重写
- (运行时)多态
- 动态联编和静态联编
- 纯虚函数、抽象类与接口

思考并回答以下问题：

- 为什么类需要支持继承
- 为什么类需要支持(运行时)多态

熟悉以下题型：

- 三种继承方式对三种访问控制属性成员的作用
- 在子类中访问父类的同名成员
- 在父类中声明虚函数并在子类中重写
- 通过虚函数重写 + 指向子类的父类指针实现运行时多态

继承

虚函数

多态

同一对象的多重身份

- 回顾面向对象思想：首先识别系统中的实体，然后提炼属性/行为，然后…
- 问题来了：同一个实体可能有多重身份，这些身份之间往往有**层次递进**关系

“张华考上了北京大学；李萍进了中等技术学校；我在百货公司当售货员：我们都有光明的前途。”

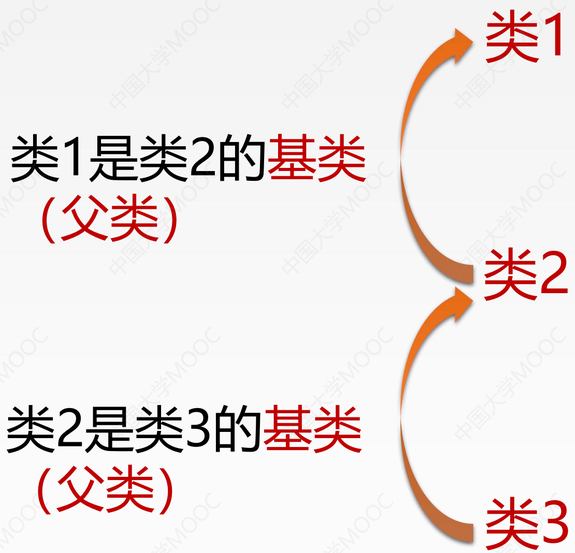
父类和子类

- 父类和子类：不同身份之间的**层次递进**关系
- 张华首先是一个人，具体一点是一个学生，再具体一点是大学生...
- 人 ← 学生 ← 大学生 ← ...

类的派生与继承 Inherit

- 人和学生之间是**层次递进**关系，而非并列关系：**若一个实体是学生，则一定是人**
- 那么在定义学生类时，无需把属于人的那一部分属性和方法再定义一遍
- 直接让学生**继承自**人：**获得**人的属性和方法
- 目的?
- 还是因为：**代码复用**

类的派生与继承 Inherit



派生

渐进明细
不断**补充**

人(性别,姓名,前途...)

学生(学校,学院...)

大学生(专业...)

张华*(实例化对象)*

继承

获得父类成员

面向对象的四个特征之3：继承

类的派生与继承 Inherit

```
class A {
public:
        int nA;
        A() {
                nA = 1;
        }
        void funcA() {
                cout << "funcA\n";
        }
};
```

```
class B : public A{
public:
        int nB;
        B() {
                nB = 2;
        }
        void funcB() {
                cout << "funcB\n";
        }
};
```

```
class C : public B {
public:
        int nC;
        C() {
                nC = 3;
        }
        void funcC() {
                cout << "funcC\n";
        }
};
```

- C c;
- 则对象 c 中既有继承得来的属性nA、 nB和方法funcA、funcB
- 也有自己专属的属性nC和方法funcC

public protected private
公有继承、保护继承、私有继承

- 继承方式：决定父类成员在子类中的访问控制属性
  - 父类的private成员不会被子类继承
  - 公有继承不改变控制属性，保护继承和私有继承指示父类成员在子类中的相应控制属性

	父类的public成员	父类的protected成员	父类的private成员
public继承	public	**protected**	不可见
protected继承	**protected**	protected	不可见
private继承	private	private	不可见

子类中的访问控制属性

父子同名成员**并存**

```
class Father {
public:
        int n = 1;
        void func() {
                cout << "This is Father";
        }
};
```

```
class Son : public Father {
public:
        int n = 2;
        void func() {
                cout << "This is Son";
        }
        void set() {
                Father::n = -1;
                n = -2;
        }
};
```

```
Son son;

son.func();
son.Father::func();

son.set();
cout << son.Father::n;
cout << son.n;
```

- 子类中同时有两个n和两个func()
- 直接使用默认指子类成员
- 如果需要使用父类的成员，需要使用父类名字空间显式指明

继承

虚函数

多态

virtual
“虚”

- 回顾：继承来源于同一对象可能有多重身份，且这些身份有层级递进关系
- 实践中会有这类情况：
  - 父类中的某些行为需要在子类中被更加具体地细化
  - 父类中的某些行为不可确定，必须在子类中实现
- 于是**虚函数**的概念产生：
  - 父类的虚函数可以在子类中被**重写**(override)，即重新实现，但**参数和返回值必须保持一致**！
- 含有虚函数的类叫做虚类

```
class Human {
public:
        virtual void say() {
                cout << "I’m human";
        }
};
```

func() 被 override

```
class Student : public Human{
public:
        void say() {
                cout << "I’m a student";
        }
};
```

纯虚函数和抽象类

父类中的某些行为不可确定
必须在更精确的子类中定义

- 某些类是**抽象**的，不是具体的，不可独立存在：
  - 我可以是男人或女人，但不可能仅是“人类对象”
  - 可以有矩形对象、三角形对象… 但不能有“图形对象”
- **纯虚函数**：不实现，仅**声明**为纯虚函数，留待子类里重写定义
- 含有纯虚函数的类叫**抽象类**，仅有纯虚函数的类叫**接口**
- 抽象类和接口不可实例化

```
class Shape {
public:
        virtual float getS() = 0;
        virtual float getC() = 0;
};
```

override

```
class Circle : public Shape{
private :
        float radius;
public:
        float getS() { return 3.14 * radius * radius; }
        float getC() { return 2 * 3.14 * radius; }
};
```

```
Shape s;
```

继承

虚函数

多态

多态 Polymorphism

- 已多次强调：同一对象可以有多重层级递进身份
- 有这种情况：**同一对象**在**不同的场合**中，被外界所关注的是**不同的身份**
- 但他的本质和应有的行为并不会因外界眼光而改变

面向对象的四个特征之4：多态

张华（一个实体）

- 生物学家认为该实体是***人类***
- 市教育局认为该实体是***学生***
- 清华大学认为该实体是***大学生***
- …

多态 Polymorphism

- 理解多态：
  - 一个对象就是内存中的一个实体，它只能属于一个确定的类：最精确的子类
  - 它可能在不同处被视为不同身份，但它本质行为方式应与外界如何看待它无关！
- 问题：如何保证一个对象执行其最本质身份的行为？
- 利用**虚函数重写 + 指针！！！**
- **指向子类对象的父类指针！！！**

多态的实现例子1

```
class Human {
public:
   virtual void say() {
      cout << "I'm human\n";
   }
};
```

override →

```
class Student : public Human{
public:
   virtual void say() {
      cout << "I'm a student\n";
   }
};
```

override →

```
class CollegeStudent : public Student{
public:
   void say() {
      cout << "I'm a college student\n";
   }
};
```

```
CollegeStudent a;
Human* p1 = (Human*) &a;
Student* p2 = (Student*) &a;
CollegeStudent* p3 =  &a;
p1->say();
p2->say();
p3->say();
```

→ **指向子类对象的父类指针**

→ **通过指针调用的是对象本质子类的方法**

```
I'm a college student
I'm a college student
I'm a college student
```

多态的实现例子2

```
class Human {
public:
    virtual void toilet()  = 0;
};
```

```
class Man : public Human {
public:
    void toilet() {
        cout << "我去男厕所";
    }
};
```

```
class Woman : public Human {
public:
    void toilet() {
        cout << "我去女厕所";
    }
};
```

```
class Non : public Human {
public:
    void toilet() {
        cout << "我去无性别厕所";
    }
};
```

```
Man man1, man2, man3;
Woman woman1, woman2;
Non non1, non2;
// ...很多很多对象，通过函数func让他们上厕所
func(&man1);
func(&woman2);
func(&non2);
```

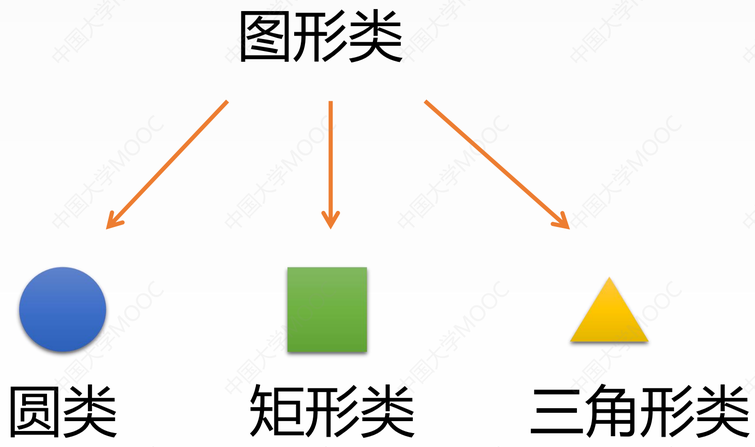
```
void func(Human *human) {
    human->toilet();
}
```

human可能指向三种不同的实际对象

事实上func并不关心实际是什么，反正都当成Human，能toilet就行

多态的意义：代码复用

- 通过“虚函数 + 指向子类对象的父类指针”，可以把不同的子类**统一视为**其共同父类
- 于是无需针对不同的子类写相同逻辑，统一视作其共同父类，利用指针操作即可
- 本质是虚函数将**能做什么**和**怎么做**分离，父类指定要做什么，子类来实现具体做法



- 问题：写一个函数，求一个图形的面积和周长之比
- 没有多态：需要为每种图形都实现一个函数
- 有多态：只需实现一个函数
- 它**不关心**这个图形**具体是什么**，反正能求面积和周长即可

静态联编与动态联编

- 上述利用**虚函数重写+指针**实现的多态特指**运行时多态**，与之相对的是**编译时多态**

静态联编 = 编译时多态 = 函数重载 = overload

动态联编 = 运行时多态 = 虚函数重写 = override

- 联编(bind)：确定具体要调用多个同名函数中的哪一个
- 静态联编：在编译时就确定了要调用的是哪个函数（根据多个重载函数的参数列表确定）
- 动态联编：直到运行时才知道实际调用的是哪个函数（根据指针指向对象的实际身份）

☆ Key 1：继承：子类直接获得父类的成员，实现代码复用

☆ Key 2：但又不是随意获得，要视**继承方式**和**访问控制属性**而定

1. 在以下成员中，能**通过**派生类对象访问的是：A

A. 公有继承的基类公有成员

B. 公有继承的基类保护成员

C. 保护继承的基类公有成员

D. 保护继承的基类保护成员

	父类public成员	父类protected成员	父类private成员
public继承	public	**protected**	不可见
protected继承	**protected**	protected	不可见
private继承	private	private	不可见

分析：

此处问的是“通过派生类对象访问”，意即在外部以a.xxx的方式访问。

如果题目问“派生类能访问的”，则意指在类定义内部访问，则除了基类私有成员以外的都可访问。

⭐ Key 1：继承：子类直接获得父类的成员，实现代码复用

⭐ Key 2：但又不是随意获得，要视**继承方式**和**访问控制属性**而定

2.有如下类声明，则类MyDERIVED中的**保护成员**的个数是：C

A.1　　B.2　　C. 3　　D.4

```
class MyBASE {
private:
        int k;
public:
        void set(int n){ k=n;}
        int get( )const{ return k;}
};
```

```
class MyDERIVED: protected MyBASE {
protected:
        int j;
public:
        void set(int m, int n){ MyBASE::set(m); j=n;}
        int get( ) const { return MyBASE::get( )+j; }
};
```

⭐ Key 3：声明虚函数的关键字是virtual，表明子类可以重写(override)它

⭐ Key 4：override需要保持参数和返回值一致，否则就是一个新函数而不是重写

⭐ Key 5：含有纯虚函数(= 0)的叫抽象类，不可实例化

1.下列关于虚函数的说法，正确的是：B

A. 虚函数是一个static类型的成员函数

B. 基类中采用virtual声明一个虚函数后，派生类中定义相同原型的函数时可以不加virtual声明

C. 虚函数是一个非成员函数

D. 派生类中的虚函数与基类中相同原型的虚函数具有不同的参数个数或类型

分析：

子类重写基类的虚函数时，可以继续通过virtual关键字申明为虚函数，表明允许被它的子类(孙子)继续重写，

如果没有这个需要或者不允许继续重写则不加。

⭐ Key 3：声明虚函数的关键字是virtual，表明子类可以重写override这个成员函数

⭐ Key 4：override需要保持参数和返回值一致，否则就是一个新函数而不是重写

⭐ Key 5：含有纯虚函数(= 0)的叫抽象类，不可实例化

2. 下列叙述中不正确的是：C

A. 含纯虚函数的类称为抽象类

B. 不能直接由抽象类建立对象

C. 抽象类不能作为派生类的基类

D. 纯虚函数没有其函数的实现部分

分析：

“建立对象”指实例化，因为抽象类中的纯虚函数没有定义(留给子类来实现)，所以抽象类不能实例化。

⭐ Key 6：**多态**：将不同类型的子类对象统一**视作**基类对象，实现代码复用

⭐ Key 7：多态 = 虚函数重写 + 通过指向子类对象的父类指针调用

1. 类声明如下，给定代码段的输出内容为：D

A. F1F1F1　　B. S2S1S2

C. S2F1S2　　D. S1S1S1

```
class Father {
public:
        int n = 1;
        virtual void func() { cout << "F" << get_n(); }
        int get_n() const { return n; }
};
class Son : public Father {
public:
        int n = 2;
        void func() { cout << "S" << get_n(); }
};
```

```
Son obj;
Father* p1 = &obj;
Son* p2 = &obj;
obj.func();
p1->func();
p2->func();
```

中国大学MOOC
搜索：C++不挂科