C++:44---关键字virtual、override、final

发布于2021-02-03 14:51:42　　阅读 641

一、虚函数

概念：在函数前面加virtual，就是虚函数

虚函数的一些概念：

只有成员函数才可定义为虚函数，友元/全局/static/构造函数都不可以

虚函数需要在函数名前加上关键字virtual

成员函数如果不是虚函数，其解析过程发生在编译时而非运行时

派生类可以不覆盖（重写）它继承的虚函数

重写（覆盖）的概念与规则

派生类重写（覆盖）基类中的函数，其中函数名，参数列表，返回值类型都必须一致，并且重写（覆盖）的函数是virtual函数

虚函数在子类和父类中的访问权限可以不同

相关规则：

①如果虚函数的返回值类型是基本数据类型：返回值类型必须相同

②如果虚函数的返回值类型是类本身的指针或引用：返回值类型可以不同，但派生类的返回值类型小于基类返回值类型

基类与派生类的虚函数名与参数列表相同，至于参数列表为什么一致是为了避免虚函数被隐藏

函数返回值有以下要求：

```
class A {
public:
int a;
public:
A(int num) :a(num) {};
virtual A& func() {}; //虚函数
};
class B:public A{
public:
int b;
public:
B(int num) :A(num) {};
virtual B& func() {}; //重写了基类的虚函数
};
```

二、为什么要设计虚函数

我们知道派生类会拥有基类定义的函数，但是对于某些函数，我们希望派生类各自定义适合于自己版本的函数，于是基类就将此函数定义为虚函数，让派生类各自实现自己功能版本的函数（但是也可以不实现）

我们通常在类中将这两种成员函数分开来：

一种是基类希望派生类进行覆盖的虚函数

一种是基类希望派生类直接继承而不要改变的函数

三、覆盖（重写）

概念：基类的虚函数，如果派生类有相同的函数，则子类的方法覆盖了父类的方法

覆盖(重写)与隐藏的关系：

覆盖与隐藏都是子类出现与父类相同的函数名，但是有很多的不同

隐藏可以适用于成员变量和函数，但是覆盖只能用于函数

覆盖（重写）在多态中有很重要的作用

四、virtual、override关键字

virtual：

放在函数的返回值前面，用于表示该成员函数为虚函数

父类虚函数前必须写；子类虚函数前可以省略（不困省不省略，该函数在子类中也是虚函数类型）

virtual只能出现在类内部的声明语句之前而不能用于类外部的函数定义

override：

父类的虚函数不可使用

放在子类虚函数的参数列表后（如果函数有尾指返回类型，那么要放在尾指返回类型后），用来说明此函数为覆盖(重写)父类的虚函数。如果类方法在类外进行定义，那么override不能加

不一定强制要求子类声明这个关键字，但是建议使用（见下面的五）

这是C++11标准填入的

override设计的最初原因：

有些情况下，我们的父类定义了一个虚函数，但是子类没有覆盖（重写）这个虚函数，而子类中却出现了一个与基类虚函数名相同、但是参数不同的函数，这仍是合法的。编译器会将派生类中新定义的这个函数与基类中原有的虚函数相互独立，这时，派生类的函数没有覆盖掉基类的虚函数版本，虽然程序没有出错，但是却违反了最初的原则

因此C++11标准添加了一个override关键字放在派生类的虚函数后，如果编译器发现派生类重写的虚函数与基类的虚函数不一样（参数或其他不一样的地方），那么编译器将报错

```
class A{
virtual void f1(int) const;
virtual void f2();
void f3();
};
calss B:public A{
```

```
void f1(int)const override; //正确
void f2(int)override;       //错误，参数不一致
void f3()override;          //错误，f3不是虚函数
void f4()override;          //错误，B没有名为f4的函数
};
```

五、禁止覆盖（final关键字）

如果我们定义的一个虚函数不想被派生类覆盖（重写），那么可以在虚函数之后添加一个final关键字，声明这个虚函数不可以被派生类所覆盖（重写）

如果函数有尾指返回类型，那么要放在尾指返回类型后

演示案例

```
class A
{
virtual void func1()final {};
};
class B:public A
{
virtual void func1()override {}; //报错，func1被A声明为final类型
};
class A
{
virtual void func1() {};
};
class B:public A
{
virtual void func1()override final {}; //正确
};
class C :public B
{
virtual void func1()override {}; //报错，func1被B声明为final类型
};
```

六、虚函数的默认实参

和其他函数一样，虚函数也可以拥有默认实参，使用规则如下：

如果派生类调用虚函数没有覆盖默认实参，那么使用的参数是基类虚函数的默认实参；如果覆盖了虚函数的默认实参，那么就使用自己传入的参数

派生类可以改写基类虚函数的默认实参，但是不建议，因为这样就违反了默认实参的最初目的

建议：如果虚函数使用了默认实参，那么基类和派生类中定义的默认实参最好一致

```
class A
{
virtual void func1(int a, int b = 10) {};
};
class B:public A
{
virtual void func1(int a,int b=10)override {}; //没有改变
};
class C :public B
{
virtual void func1(int a, int b = 20)override {}; //改变了默认实参，不建议
};
class D :public C
{
virtual void func1(int a, int b)override {}; //删去了默认实参，那么在调用fun1时，必须传入
};
```

七、动态绑定

概念：当某个虚函数通过指针或引用调用时，编译器产生的代码直到运行时才能确定到该调用哪个版本的函数（根据该指针所绑定的对象）

必须清楚动态绑定只有当我们通过指针或引用调用“虚函数”时才会发生，如果通过对象进行的函数调用，那么在编译阶段就确定该调用哪个版本的函数了（见下面的演示案例）

当然，如果派生类没有重写基类的虚函数，那么通过基类指针指向于派生类时，调用虚函数还是调用的基类的虚函数（因为派生类没有重写）

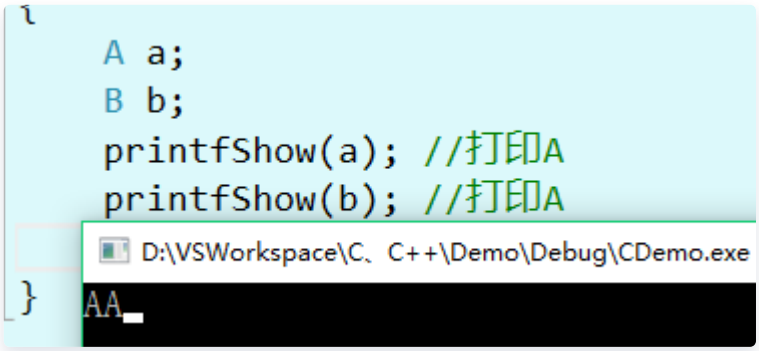
动态绑定与“派生类对象转换为基类对象”是相似的，原理相同

演示案例

```
class A
{
public:
void show()const{
cout << "A";
};
};
class B :public A //B继承于A
{
public:
void show()const{
cout << "B";
};
};
void printfShow(A const& data)
{
data.show();
```

```
}
int main()
{
A a;
B b;
printfShow(a);
printfShow(b);
return 0;
}
```

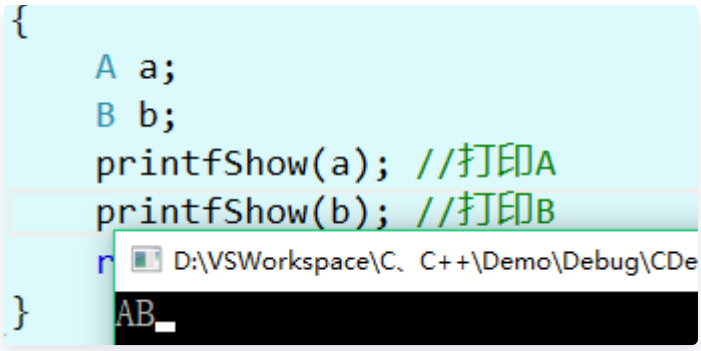
上面的程序中，B继承于A，并且B隐藏了A的show()函数。当我们运行程序时，可以看到程序打印的是“AA”。所以可以得出，非虚函数的调用与对象无关，而是取决于类的类型（这个在程序的编译阶段就已经确定了），此处函数的参数类型为A，所有打印的永远是A里面的show()函数



现在我们修改程序，将基类A的show函数改为虚函数形式

```
class A
{
public:
virtual void show()const{
cout << "A";
};
};
```

现在再来运行程序，可以看到程序打印的是“AB”。这就是动态绑定产生的效果，对于虚函数的调用是在程序运行时才决定的



八、回避虚函数的机制

上面我们介绍过，我们通过指针调用虚函数，会产生动态绑定，只有当程序运行时才回去确定到底该调用哪个版本的函数

某些情况下，我们希望对虚函数的调用不要进行动态绑定，而是强迫其执行虚函数的某个特定版本。这种方式的调用是在编译时解析的。方法是通过域运算符来实现

通常，只有成员函数（或友元）中的代码才需要使用作用域运算符来回避虚函数的机制

什么时候需要用到这种回避虚函数的机制：

> 通常，基类定义的虚函数要完成继承层次中所有的类都要完成的共同的任务，而各个派生类在虚函数中各自添加自己的功能。此时，派生类希望使用基类的虚函数来完成大家共同的任务，那么就通过域运算符来调用基类的虚函数

```
#include <iostream>
using namespace::std;
class A
{
public:
virtual void func1() { cout << "A" << endl; };
};
class B:public A
{
public:
virtual void func1()override { cout << "B" << endl; };
};
int main()
{
A *p;
B b;
p = &b;
p->A::func1();   //正确，打印A
//p->B::func1(); //错误的用法
p->func1();      //正确，打印B
return 0;
}
```