多态

**指的是不同对象接收到同一消息时会产生不同的行为**

**在同一个类或继承体系结构的基类与派生类中，用同名函数来实现各种不同的功能**

通过基指针完成对派生类函数的访问

* 1. 重载多态：包括函数重载和运算符重载；上面的add函数就是函数重载多态的例子。
  2. 模板多态：通过一个模板生成不同的函数或类（第7章介绍）；
  3. **继承多态**：通过基类对象的指针（引用），调用不同派生类对象的重定义同名成员函数，表现出不同的行为。一般情况下，多态即指这种类型

要实现继承多态性，须具备三个必要条件：

* + 要有**继承**；
  + 派生类要**覆盖（重定义）基类的虚函数**，即派生类具有和基类函数据原形完全相同的虚成员函数；
  + 把**基类的指针或引用绑定到派生类对象**上。

用虚函数来表达这类确实存在但又无法实施的抽象概念。

**这一特征对程序功能扩展带来极大的方便**

多态是指当基类的指针（或引用）绑定到派生类对象上，通过此指针（引用）调用基类的成员函数时，实际上调用到的是该函数在派生类中的覆盖函数版本

* 用基类对象的指针或引用作为函数的参数，通过它调用派生类对象中的覆盖函数版本。

void animalSound(Animal &animal) { animal.sound(); }

* animalSound(dog); //调用Dog::sound()
* animalSound(cat); //调用Cat::sound()

**基类提供接口，派生类提供实现**

**要把调用函数名与对应函数（这些函数可能来源于不同的文件或库）关联在一起，这个过程就是绑定（binding），又称联编。**

静态联编：编译时就把调用函数名与具体函数绑定在一起。 **执行速度快**

动态联编：程序运行时才把调用函数名与具体函数绑定在一起。

在C++中的编译时多态性是通过函数重载和运算符重载实现的

在C++中的运行时多态性是通过继承和虚函数实现的。

**基类与派生类的赋值相容**

* + **派生类对象可以赋值给基类对象。**
  + **派生类对象的地址可以赋值给指向基类对象的指针。**
  + **派生类对象可以作为基类对象的引用**

**只能通过基类对象（或基类对象的指针或引用）访问到派生类对象从基类中继承到的成员**

**虚函数的定义形式**

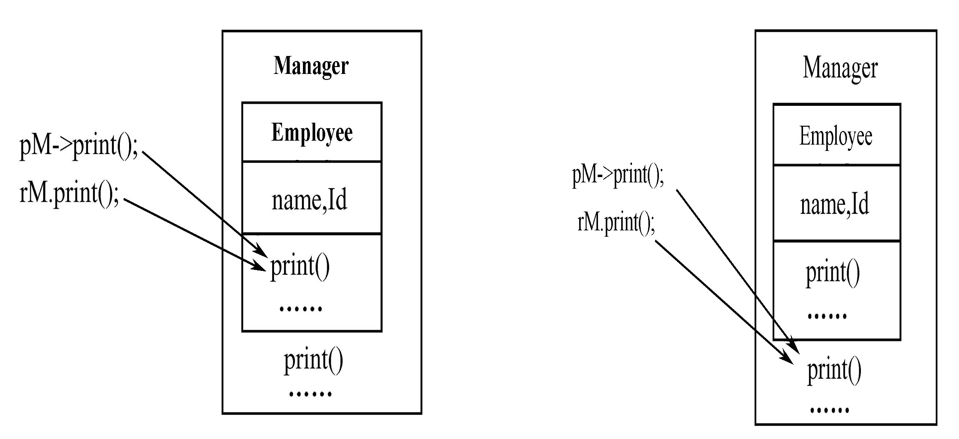
**class x{**

**……**

**virtual f(参数表);**

**}**

**图左为非虚函数，图右为虚函数**



* + 如果基类中的非静态成员函数被定义为虚函数，且派生类覆盖了基类的虚函数，**当通过基类的指针或引用调用派生类对象中的虚函数时，编译器将执行动态绑定，调用到该指针（或引用）实际所指对象所在类中的虚函数版本**。

**通过指向派生类对象的基类指针访问函数成员时，**

* + **非虚函数由指针类型决定调用的版本**
  + **虚函数由指针指向的实际对象决定调用的版本**

用override限定类D的outData函数，编译器就知道它是基类虚函数的覆盖函数版本。当该函数的参数表与基类的不同时，**编译器就会检测到该错误，不允许通过编译**。

class B {

public:

virtual void outData(**int** a)

{ cout << a; };

};

class D : public B {

public:

void outData(**double** b) **override**{ cout << b; } //错误

};

override**只能用在派生类中限定从基类继承到的虚函数**

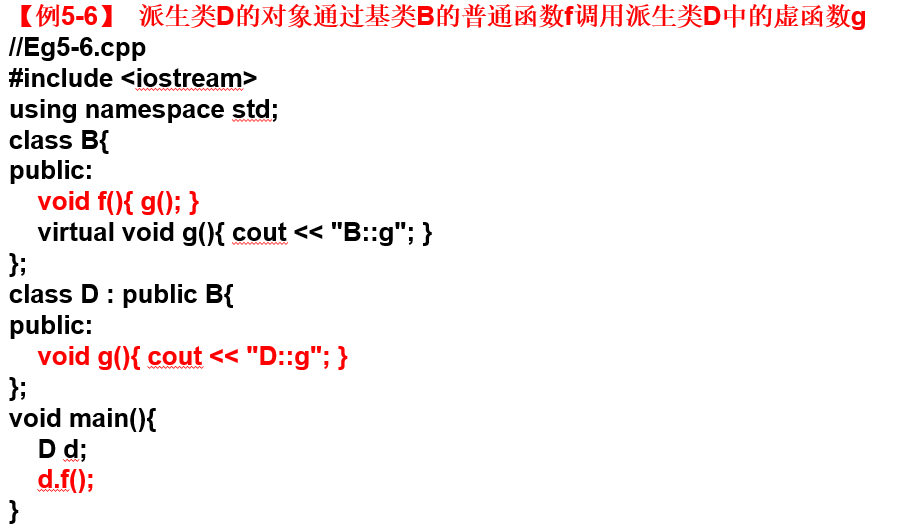
final用于限定只想让派生类继承，而不允许被覆盖的虚成员函数。

**final只能限定虚函数**，被限定为final的成员函数，则任何派生类对该函数的覆盖定义都是错误的

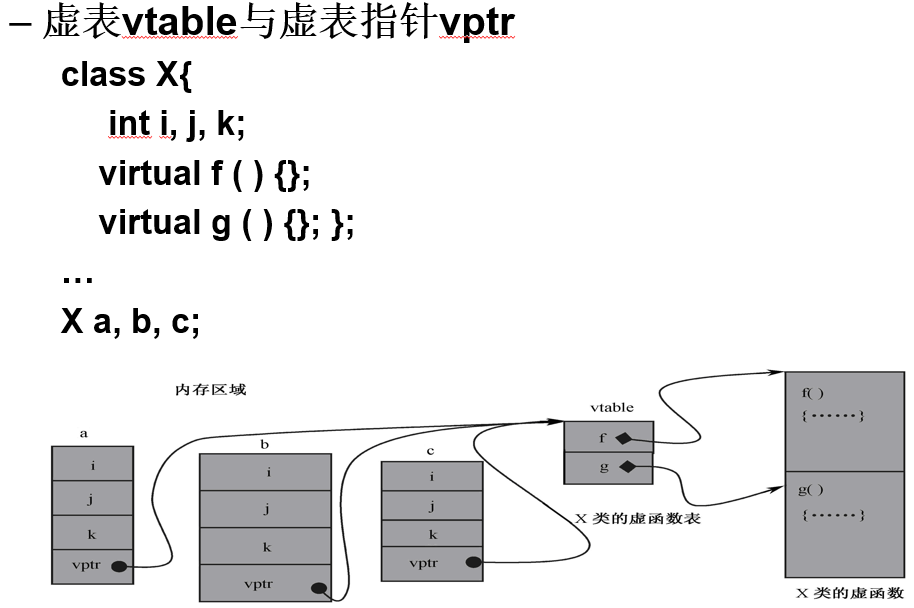
**一旦将某个成员函数声明为虚函数后，它在继承体系中就永远为虚函数了**

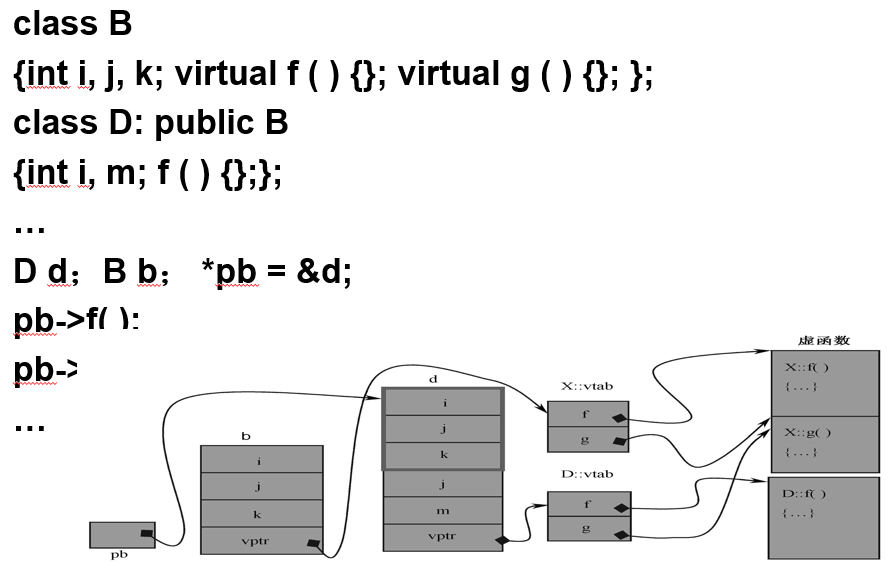
**派生类中的虚函数要保持其虚特征，必须与基类虚函数的函数原型完全相同，**否则就是普通的重载函数，与基类的虚函数无关。

**构造函数和静态成员函数不能定义为虚函数。内联函数不能是虚函数。**



**虚函数的实现技术**





**纯虚函数（不知道如何实现它的某些成员函数）**来表示这类函数。具有纯虚函数的类就称为**抽象类**

class X{

……

**virtual returnType funcName (param) = 0;**

}

**在声明时被初始化为0**

**不能建立抽象类的对象，只能作为其他类的基类**

* 1. 如果派生类只是简单地继承了抽象类的纯虚函数，而**没有覆盖基类的纯虚函数，则派生类也是一个抽象类。**

**抽象类的主要用途——作接口**

**RTTI**

运行时类型信息（Run-time Type Information，RTTI）提供了在程序运行时刻**确定对象类型**的方法

dynamic\_cast是一个**强制类型转换操作符**，主要用于多态基类的指针或引用与派生类指针或引用之间的转换，它是在程序运行时刻执行的

dynamic\_cast<type \*>(e) //指针转换，e是指针

dynamic\_cast<type &>(e) //引用转换，e必须是左值

dynamic\_cast<type &&>(e) //右值转换，e不能是左值

**type必须是类类型**

* + **当用基类对象的指针操作派生类对象时，只能通过该指针访问派生类中的虚函数（这些虚函数也是基类中的虚函数）。而那些在基类中没有被定义为虚函数，或派生类新增加的函数，通过基类指针是无法访问的。**

**typeid**操作符在程序运行时**判定一个对象的真实数据类型，t**ypeid定义于头文件typeinfo中，它的用法如下：

**typeid(exp)**

typeid操作符返回一个**type\_info类对象的引用**

该类有一个成员函数name，可以用它来获得表达式exp的类型名称

**cout<<"1: "<<typeid(a).name()<<endl;**

**cout<<"2: "<<typeid(p).name()<<endl;**

**先用typeid识别出类型，再进行强制类型转换就会成功**；