虚函数

思考：为什么要引入虚函数//test_virtual.cpp

【例4】没有使用虚函数的例题。

```
#include<iostream.h>
class base            /定义基类base
{    public:
       void who()
        {cout<<"this  is  the  class of     base!"<<endl;}
};
class derive1:public base          //定义派生类derive1
{
       public:
           void  who()
           {cout<<"this is the class of derive1!"<<endl;}
};
```

```
class derive2 : public base          //定义派生类
derive2
{   public:
   void who()
 {cout<<"this is the class of derive2!"<<endl;}
 };
void  main()
{   base obj,*ptr;
    derive1 obj1;
    derive2 obj2;
    ptr = &obj;
    ptr->who();
```

```
    ptr=&obj1;    //指向第一个子类
     ptr->who();
     ptr=&obj2;    //指向第二个子类
     ptr->who();
     obj1.who();  //分别指向子类
     obj2.who();
```

程序运行结果：

```
This is the class of base! (a)
This is the class of base! (b)
This is the class of base! (c)
This is the class of derive1!   (d)
This is the class of derive2!   (e)
```

就是说，通过指针引起的普通成员函数调用，仅仅与指针的类型有关，而与指针正指向什么对象无关。在这种情况下，必须采用显式的方式调用派生类的函数成员。

要调用派生类的函数成员，就需要引入虚函数的概念。

这里，只需将基类的who()函数声明为虚函数即可。

* 虚函数的定义及使用

1.虚函数的定义

虚函数的定义是在基类中进行的。它是在基类中需要定义为虚函数的成员函数的声明中冠以关键字virtual。当基类中的某个成员函数被声明为虚函数后，此虚函数就可以在一个或多个派生类中被重新定义，在派生类中重新定义时，其函数原型，包括返回类型、函数名、参数个数、参数类型以及参数的顺序都必须与基类中的原型完全相同。

一般虚函数的定义语法如下：

virtual<函数类型><函数名>(形参表)

{　　函数体　　}

其中，被关键字virtual说明的函数为虚函数。特别要注意的是，**虚函数的作用是允许在派生类中对基类的虚函数重新定义，显然它只能用于类的继承层次中**。

＊使用虚函数的情况：

（1）首先应考虑成员函数所在的类是否作为一个基类，然后看成员函数在类的继承后有无可能被更改功能，如果希望更改其功能，一般应该把它声明为虚函数。

（2）应考虑对成员函数的调用是通过对象名还是通过基类指针或引用，如果是通过基类指针或引用去访问，则应声明为虚函数。

[例5] 虚函数的作用。//test_virtual.cpp

```
#include<iostream.h>
class Base
{
      public:
           Base(int x,int y)
           {  a=x; b=y; }
           virtual void show()     //定义虚函数show()
           { cout<<"Base----------\n"; cout<<a<<" "<<b<<endl;}
        private:
           int a,b;
  };
class Derived : public Base
{
        public:
           Derived(int x,int y,int z):Base(x,y){c=z; }
           void show()           //重新定义虚函数show()
           { cout<< "Derived---------\n"; cout<<c<<endl;}
       private:
           int c;
};
```

```
void main()
{
        Base mb(60,60),*pc;
        Derived mc(10,20,30);
        pc=&mb;
        pc->show();     //调用基类Base的show()版本
        pc=&mc;
        pc->show();     //调用派生类Derived的show()版本
 }
```

程序运行结果如下:

```
Base---------
60   60
Derived--------
30
```

2. 虚函数与重载的关系

在一个派生类中重新定义基类的虚函数是函数重载的另一种特殊形式，但它不同于一般的函数重载。

一般的函数重载，只要函数名相同即可，函数的返回类型及所带的参数可以不同。但当重载一个虚函数时，也就是说在派生类中重新定义此虚函数时，要求函数名、返回类型、参数个数、参数类型以及参数的顺序都与基类中的原型完全相同，不能有任何的不同。

3. 多继承中的虚函数

在多继承中由于派生类是由多个基类派生而来的，因此，虚函数的使用就不像单继承那样简单。请看下面的例题。

```
#include<iostream.h>
class base1
 {      public:
            virtual void who() //函数who()为虚函数
              {cout<<"this is the class of base1!"<<endl;}
};
class base2                          //定义基类base2
{    public:
        void who()                   //此函数who()为一般的函数
            { cout<<"this is the class of base2!"<<endl;}
};
```

```
class derive : public base1,public base2
{  public:
     void who()
     {cout<<"this is the class of derive!"<<endl;}
};
main()
{     base1 obj1,*ptr1;
      base2 obj2,*ptr2;
      derive obj3;
      ptr1=&obj1;
      ptr1->who();
```

```
ptr2=&obj2;
      ptr2->who();
     ptr1=&obj3;
     ptr1->who();
     ptr2=&obj3;
     ptr2->who();
}
```

此时，程序执行的结果为

this is the class of base1!

this is the class of base2!

this is the class of derive!

this is the class of base2!

从上面的例子看出，派生类**derive**中的函数**who()**在不同的场合呈现不同的性质。如相对**base1**路径，由于在**base1**中的**who()**函数前有关键字**virtual**，所以它是一个虚函数；若相对于**base2**派生路径，在**base2**中的**who()**函数为一般函数，所以，此时它只是一个重载函数。

若一个派生类，它的多个基类中有公共的基类，在公共基类中定义一个虚函数，则多重派生以后仍可以重新定义虚函数，也就是说，**虚特性是可以传递的**。请看下面的例题。

【例7】多继承中虚特性的传递例题。

```
#include<iostream.h>
class base
{   public:
        virtual void who()          //定义虚函数
            {cout<<"this is the class of base!"<<endl;}
};
```

```
class base1:public base  //定义派生类base1
{    public:
      void who()
         {cout<<"this is the class of base1!"<<endl;}
};
class base2 : public base //定义派生类类base2
{    public:
       void who()
          {cout<<"this is the class of base2!"<<endl;}
};
```

```
class derive:public base1,public base2 //定义派生类derive
{    public:
      void who()
       {cout<<"this is the class of derive!"<<endl;}
};
void  main()
{     base1*ptr1;
       base2*ptr2; derive obj;
       ptr1 = &obj;
       ptr1->who();
       ptr2 = &obj;
       ptr2->who();
}
```

此时，程序执行的结果为

Tis is the class of derive!

This is the class of derive!

进一步探讨虚函数与实函数的区别

假设基类和派生类都只有一个公有的数据成员，其中类A有vfunc1和vfunc2两个虚函数和func1和func2两个实函数。类A公有派生类B，类B改写vfunc1和func1函数，它又作为类C的基类，公有派生类C。类C也改写vfunc1和func1函数。

下图给出3个类建立的vptr和vtable之间的关系图解以及实函数与虚函数的区别。

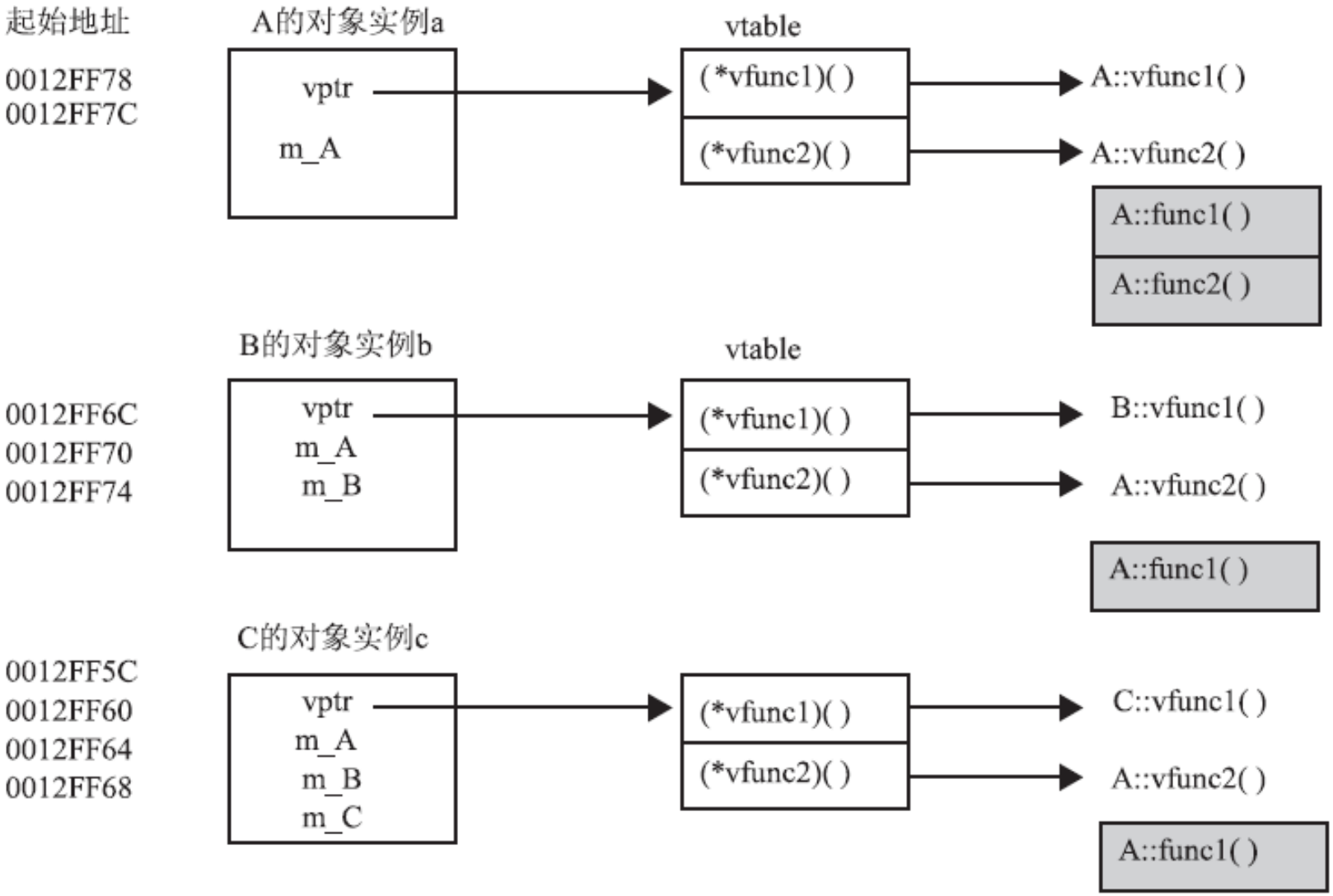

图 9-4　实函数和虚函数的图解示意图

首先给vptr分配地址，它所占字节数决定对象中最长数据成员的长度。因为3个类的数据成员都是整型，所以VC为vptr分配4个字节。如果有double型的数据，则要分配8个字节。

从图中可见，对象的起始地址是vptr。它指向vtable，vtable为每个虚函数建立一个指针函数，如果只是继承基类的虚函数，则它们调用基类的虚函数，这就是b和c的vtable表中(*vfunc2)( )项所描述的情况。如果派生类改写了基类的虚函数，则调用自己的虚函数，这就是b和c的vtable表中(*vfunc1)( )项所描述的情况。

实函数不是通过地址调用，用带底纹的方框表示，它们由对象的名字支配规律决定。

【例7】是程序实现。

【例7】实函数和虚函数调用过程。

```
#include <iostream>
using namespace std;
class A
{
   public:
    int m_A;
    A(int a){m_A=a;}
    void func1(){cout<<"A::func1( )"<<endl;}
    void func2(){cout<<"A::func2( )"<<endl;}
    virtual void vfunc1(){cout<<"A::vfunc1( )"<<endl;}
    virtual void vfunc2(){cout<<"A::vfunc2( )"<<endl;}
};
```

```
class B:public A
{
  public:
    int m_B;
    B(int a, int b):A(a),m_B(b){}
    void func1(){cout<<"B::func1( )"<<endl;}
    void vfunc1(){cout<<"B::vfunc1( )"<<endl;}
};
class C:public B
{
  public:
    int m_C;
    C(int a, int b, int c):B(a,b),m_C(c){}
    void func1(){cout<<"C::func1( )"<<endl;}
    void vfunc1(){cout<<"C::vfunc1( )"<<endl;}
};
```

```
void main()
{    cout<<sizeof(A)<<“,”<<sizeof(B)<<“.“
 <<sizeof(C)<<endl;   //输出类的长度（字节数）
    A a(11);
    B b(21,22);
    C c(31,32,33);
  //输出类的首地址及数据成员地址，验证首地
址是vptr地址
 cout<<&a<<","<<&(a.m_A)<<endl;
 cout<<&b<<","<<&b.m_A<<","<<&b.m_B<<endl;
 cout<<&c<<","<<&c.m_A<<",“
        <<&c.m_B<<","<<&c.m_C<<endl;
```

```
//使用基类指针
 A* pa=&a;          //pa指向基类A
  pa->vfunc1();      //调用A::vfunc1( )
  pa->vfunc2();      //调用A::vfunc2( )
  pa->func1();       //调用A::func1( )
  pa->func2();       //调用A::vfunc2( )
  cout<<endl;
  pa=&b;           // pa指向派生类B
  pa->vfunc1();      //调用B::vfunc1( )
  pa->vfunc2();      //调用A::vfunc2( )
  pa->func1();       //静态联编，只能调用A::func1( )
  pa->func2();       //静态联编，只能调用A::func2( )
  cout<<endl;
```

```
pa=&c;          // pa指向派生类C
  pa->vfunc1();      //调用C::vfunc1( )
  pa->vfunc2();      //调用A::vfunc2( )
  pa->func1();       //静态联编，只能调用A::func1( )
  pa->func2();       //静态联编，只能调用A::func1( )
   cout<<endl;
   //使用类B的指针，类B是类C的直接基类
  B* pb=&b;        // pb指向基类B
  pb->vfunc1();      //调用B::vfunc1( )
  pb->vfunc2();      //调用A::vfunc2( )
  pb->func1();       //静态联编，调用B::func1()
  pb->func2();       //静态联编，只能调用A::func2()
```

```
 cout<<endl;
   pb=&c;            // pb指向派生类C
   pb->vfunc1();       //调用C::vfunc1()
   pb->vfunc2();       //调用A::vfunc2()
   pb->func1();       //静态联编，只能调用B::func1()
   pb->func2();       //静态联编，只能调用A::func2()
   cout<<endl;
   //使用类C的指针
   C* pc=&c;         // pc指向派生类C
   pc->vfunc1();       //调用C::vfunc1( )
   pc->vfunc2();       //调用A::vfunc2( )
   pc->func1();       //静态联编，调用C::func1()
   pc->func2();       //静态联编，只能调用A::func2( )
}
```

对象a有一个整型数据，应分配4个字节，vptr也是4个字节，总共8个字节。对象b和c依次增加一个整型数据成员，内存分配也顺增4个字节。输出结果如下：

8,12，16
0012FF78,0012FF7C　　//vptr, m_A
0012FF6C,0012FF70,0012FF74　　//vptr, m_A, m_B
0012FF5C,0012FF60,0012FF64,0012FF68　　//vptr, m_A,
// m_B, m_C
// A* pa=&a;
A::vfunc1()
A::vfunc2()
A::func1()

```
A::func2()
// pa=&b;
B::vfunc1()
A::vfunc2()
A::func1()
A::func2()
//pa=&c;
C::vfunc1()
A::vfunc2()
A::func1()
A::func2()
// B* pb=&b;
B::vfunc1()
A::vfunc2()
B::func1()
```

```
A::func2()
// pb=&b;
C::vfunc1()
A::vfunc2()
B::func1()
A::func2()
//C* pc=&c;
C::vfunc1()
A::vfunc2()
C::func1()
A::func2()
```

* 构造函数和析构函数调用虚函数

在构造函数和析构函数中调用虚函数时，采用静态联编，即它们所调用的虚函数是自己的类或基类中定义的函数，但不是任何在派生类中重定义的虚函数。下面给出一个具体的例子。

【例】在构造函数和析构函数中调用虚函数。

```
#include <iostream>
using namespace std;
class A {
  public:
     A(){}
     virtual void func( )
     { cout << "Constructing A " <<  endl; }
     ~A( ) { }
     virtual void fund( )
    { cout << "Destructor A " <<  endl; }
};
```

```
class B : public A {
    public:
       B( ) { func( ); }
       void fun( ) { cout<<"Come here and go...";
func( ); }
       ~B( ) { fund( ); }
  };
  class C : public B {
    public:
       C( ) { }
       void func( ) { cout <<"Class C" << endl; }
       ~C( ) {fund();}
       void fund( ){ cout << "Destructor C " <<  endl; }
  };
```

```
void main( )
{
      C c;
      c.fun( );
 }
```

输出结果如下：

```
Constructing A              // 建立对象c调用B( )产生
Come here and go...Class C    // c.fun( )输出
Destructor C                // 析构对象c时，由~C()产生
Destructor A                // 析构对象c时调用~B()产生
```

在建立C类的对象c时，它所包含的基类子对象在派生类中定义的成员建立之前被建立。

在对象撤消时，该对象所包含的在派生类中定义的成员要先于基类子对象之前撤消。

函数func是虚函数，构造对象c时，类A构造函数是空函数，没有输出。执行类B的构造函数时调用func，但B没有定义func，所以调用基类A定义的虚函数func。类C的构造函数为空函数，所以构造对象c时，只有一句输出信息。

执行语句“c.fun();”时，类C自己没有函数fun，转去执行它的直接基类B的fun，输出“Come here and go...”。这个函数fun接着调用func，此时是执行基类B的func还是派生类的func？显然，c是派生类C的对象，类C有自己的func。按照虚函数调用规则，它不会去调用基类A的func，而应该执行自己的func，输出“Class C”。

析构时应先调用C的析构函数，输出“Destructor C”。接着调用类B的析构函数，这个析构函数调用虚函数fund。这个虚函数分别在类B的基类A和派生类C中定义，它只能调用它的基类中的虚函数fund，输出“Destructor A”。基类A中的析构函数没有输出信息，程序结束运行。

目前推荐的C++标准不支持虚构造函数。由于析构函数不允许有参数，因此一个类只能有一个虚析构函数。虚析构函数使用virtual 说明。只要基类的析构函数被说明为虚函数，则派生类的析构函数，无论是否使用virtual进行说明，都自动地成为虚函数。

delete运算符和析构函数一起工作（new 和构造函数一起工作），当使用delete删除一个对象时，delete隐含着对析构函数的一次调用，如果析构函数为虚函数，则这个调用采用动态联编。一般说来，如果一个类中定义了虚函数，析构函数也应说明为虚函数，尤其是在析构函数要完成一些有意义的任务时，例如释放内存等。

如果基类的析构函数为虚函数，则在派生类未定义析构函数时，编译器所生成的析构函数也为虚函数。

* 虚函数的限制

如果我们将所有的成员函数都设置为虚函数，当然是很有益的。它除了会增加一些额外的资源开销，没有什么坏处。但设置虚函数须注意以下几点。

①只有成员函数才能声明为虚函数。因为虚函数仅适用于有继承关系的类对象，所以普通函数不能声明为虚函数。

②虚函数必须是非静态成员函数。这是因为静态成员函数不受限于某个对象。

③内联函数不能声明为虚函数。因为内联函数不能在运行中动态确定其位置。

④构造函数不能声明为虚函数。多态是指不同的对象对同一消息有不同的行为特性。虚函数作为运行过程中多态的基础，主要是针对对象的，而构造函数是在对象产生之前运行的，因此，虚构造函数是没有意义的。

⑤析构函数可以声明为虚函数。析构函数的功能是在该类对象消亡之前进行一些必要的清理工作。析构函数没有类型，也没有参数，和普通成员函数相比，虚析构函数情况略为简单些。(动态对象的释放)

虚析构函数的声明语法如下：

virtual~类名

例如：

```
class   B
    {
        public:
                //...
            virtual  ~B();
    };
```

抽象类

* 纯虚函数

一个抽象类至少带有一个纯虚函数。纯虚函数是一个在基类中说明的虚函数，**它在该基类中没有定义具体的操作内容，要求各派生类根据实际需要定义自己的实现内容**。纯虚函数的声明形式如下：

virtual <函数类型> <函数名> (参数表) = 0;

纯虚函数与一般虚函数在书写形式上的不同在于其后面加了“=0”，表明在基类中不用定义该函数，它的实现部分——函数体留给派生类去做。

```
#include<iostream.h>
 const   double  PI = 3.14159;
 class   Shapes //抽象基类Shapes声明
{  protected:
        int   x  ,  y;
     public:
        void   setvalue(int  xx ,int  yy = 0){x = xx ;y = yy;}
      virtual   void   display() = 0;      //纯虚函数成员
 };
```

```
class   Rectangle : public Shapes                    //派生类Rectangle声明
{
    public:                                                    //虚成员函数
        void   display(){cout<<"The area of rectangle is:"<<x*y<<endl;}
};
class    Circle :  public   Shapes                    //派生类Circle声明
 {
    public:                                                    //虚成员函数
        void    display(){cout<<"The area of  circle is:"<<PI*x*x<<endl;}
};
```

```
void   main()
{     Shapes*  ptr[2];                          //声明抽象基类指针
      Rectangle   rect1;
      Circle     cir1;
ptr[0]=&rect1;                          //指针指向Rectangle类对象
ptr[0]->setvalue(5,8);
ptr[0]->display();
ptr[1]=&cir1;                           //指针指向Circle类对象
ptr[1]->setvalue(10);
ptr[1]->display();
}
```

本例的程序运行结果为

The area of rectangle is:40

The area of circle is : 314.159

另外，程序中派生类的虚成员函数display()并没有用关键字virtual显式说明，因为它们与基类的纯虚函数具有相同的名称及参数和返回值，由系统自动判断确定其为虚成员函数。