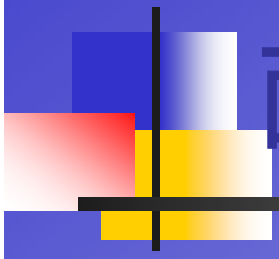
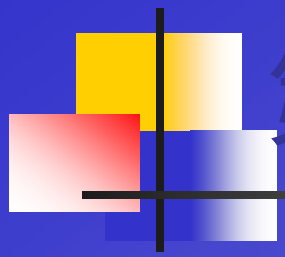


面向对象的程序设计

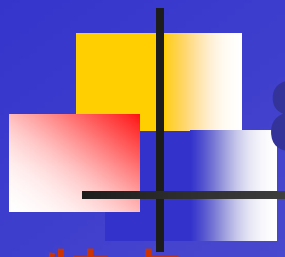
华中科技大学计算机学院

李瑞轩



第8章 多继承类

- 8.1 多继承类
- 8.2 虚基类
- 8.3 派生类成员
- 8.4 构造与析构
- 8.5 类的存储空间



8.1 多继承类

特点：

① 多继承派生类有多个基类或虚基类。

② 派生类继承所有基类(包括间接基类)的数据成员和成员函数

③ 派生类可以定义新的数据成员和函数成员，以便描述新类特有的或不同的属性和功能

④ 单继承是多继承的一种特例，多继承派生类具有更强的类型表达能力。

⑤ 多继承机制是C++语言所特有的（Java、SmallTalk没有）。因此，C++具有更强的描述对象方面的功能。其他面向对象语言需要描述多继承类的对象时，常常通过对象成员或委托代理实现多继承。委托代理在多数情况下能够满足需要，但当对象成员和基类的类型相同或存在共同的基类时，就可能对同一个物理对象重复进行初始化。 【例8.1】

【例8.1】定义具有水平滚动条和垂直滚动条的窗口类。

```
class Window{
   //...
public:
   Window(int top, int left, int bottom, int right);
   ~Window( );
};
class HScrollbar{
   //...
public:
   HScrollbar (int top, int left, int bottom, int right);
   ~ HScrollbar ( );
};
class VScrollbar{
   //...
public:
   VScrollbar (int top, int left, int bottom, int right);
   ~ VScrollbar( );
};
```

```
class ScrollableWind: public Window{
    HScrollbar  hScrollBar;  //委托hScrollBar代理水平滚动
    VScrollbar  vScrollBar;  //委托vScrollBar代理垂直滚动
    //...
public:
    ScrollableWind(int top, int left, int bottom, int right);
    ~ScrollableWind( );
};
ScrollableWind::ScrollableWind (int t, int l, int b, int r):Window(t, l, b, r),
                    hScrollbar(t, r+1, b-1, r), vScrollbar(b-1,l-1,b,r+1)
{
    //...
}
```

如果Window、hScrollBar和hScrollBar分别初始化显示端口，则派生类ScrollableWind的对象就会多次初始化显示端口，从而导致显示屏因多次初始化显示端口出现多次闪烁。→使用虚基类定义ScrollableWind

用多继承方式定义派生类ScrollableWind

```
class ScrollableWind:public Window,public HScrollbar,public VScrollbar{
   //...
public:
   ScrollableWind(int top, int left, int bottom, int right);
   ~ScrollableWind( );
};
ScrollableWind::ScrollableWind (int t, int l, int b, int r): Window(t, l, b, r),
                  HScrollbar(t, r+1, b-1, r),VScrollbar(b-1,l-1,b,r+1)
{
   //...
}
```

1. 多继承派生类的定义：

   class Derived:<派生方式> 基类1，<派生方式> 基类2，…{

   <类体>

   };

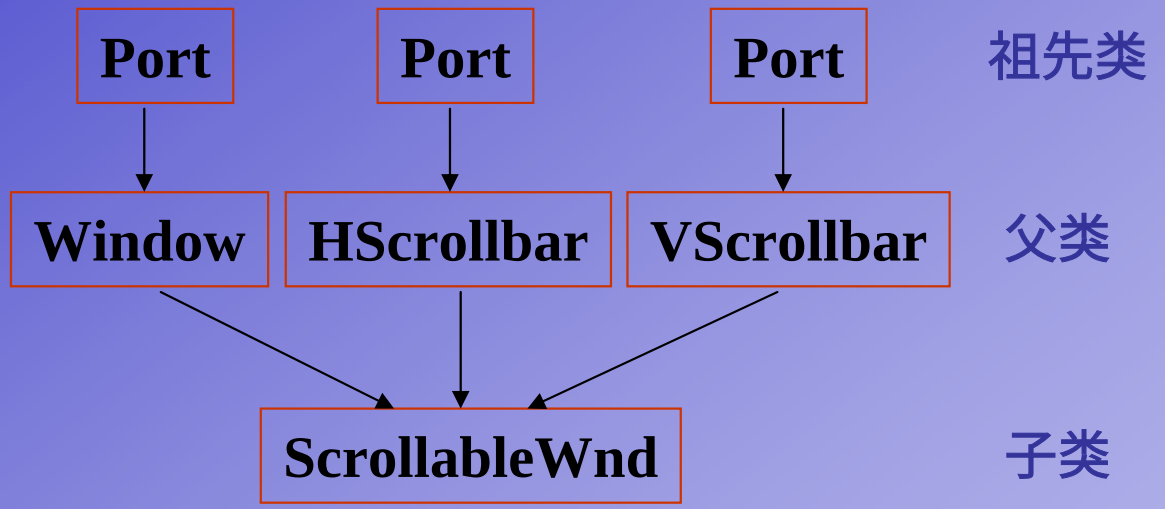
   public
   private
   protected

2. 派生类对象多次初始化同一基类数据成员问题。

派生类对象多次初始化同一基类数据成员问题

- 假设【例8.1】中，类Window、HScrollbar、VScrollbar都是从同一个基类Port派生，即：

```
class Port{ /*…*/};
class Window:public Port{ /*…*/};
class HScrollbar:public Port{ /*…*/};
class VScrollbar:public Port{ /*…*/};
class ScrollableWind:public Window,public HScrollbar,public VScrollbar{/*…*/};
```

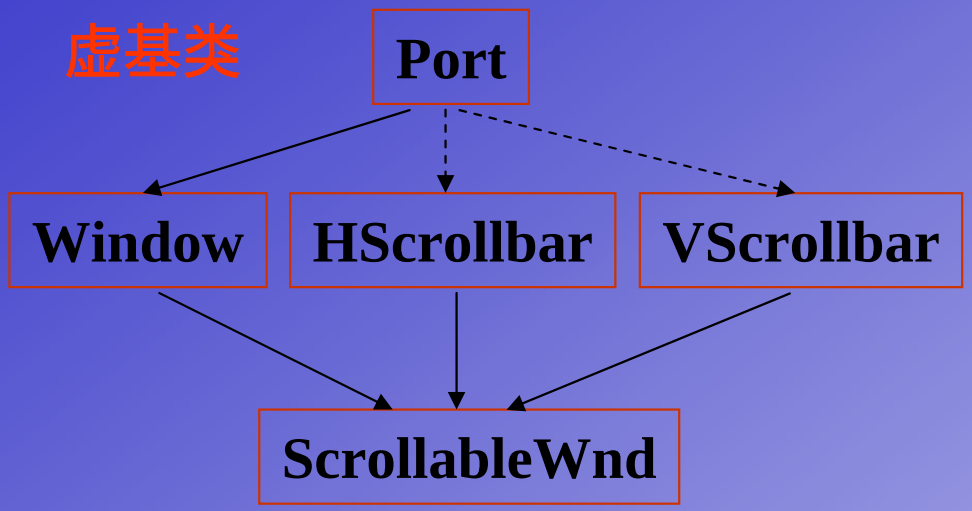


ScrollableWnd派生树图

创建ScrollableWnd对象时，Port的构造函数通过3条不同的路径，被调用了3次，从而将显示端口初始化3次。即，1个子类有3个同名祖先类，不符合实际！

8.2 虚基类

?



如何实现：1个子类通过3条不同的路径到达同一个祖先类？即，创建ScrollableWnd对象时，显示端口Port仅被初始化1次？

ScrollableWnd派生树和存储空间

```
class Window:virtual public Port{ /*…*/};
class HScrollbar:public virtual Port{ /*…*/};
class VScrollbar:public virtual Port{ /*…*/};
class ScrollableWind:public Window,public HScrollbar,public VScrollbar{/*…*/};
```

virtual和派生方式可以互换位置

虚基类使用说明

1. 仅用于多继承，因为同一个类不能多次作为某个派生类的直接基类，但可多次作为其间接基类，从而引起存储空间的浪费和其他问题。
2. 同一颗派生树中的**同名虚基类，共享同一个存储空间**；其**构造函数和析构函数仅执行1次**，且构造函数尽可能最早执行，而析构函数尽可能最晚执行。回上页
3. 如果虚基类与基类同名，则它们将分别拥有各自的存储空间，只有同名虚基类才共享存储空间，而同名基类则拥有各自的存储空间。
4. 虚基类和基类同名必然会导致二义性访问，编译程序会对这种二义性访问提出警告。当出现这种情况时，要么将基类说明为对象成员，要么将基类都说明为虚基类。分析【例8.3】

【例8.3】说明虚基类的二义性访问问题。

```
#include <iostream.h>
struct A{
    int a;
    A(int x) {  a=x; }
};
struct B: A{
    B(int x):A(x) {  }
};
struct C{
    C( ) { }
};
struct D: virtual A, C{
    D(int x):A(x) { }
};
struct E: B, D{
    E(int x):A(x), B(x+5), D(x+10) { }
};
```

```
void main(void)
{
    E  e(0);
  //cout<<"a="<<e.a;     //出现二义性？
    cout<<"a="<<e.B::a;
    cout<<"a="<<e.D::a;
}
```

输出：
a=5
a=0

为解决e.a产生的二义性，要么将E的基类B说明为对象成员，要么将B的基类A说明为虚基类。若将B的基类A说明为虚基类，则e.a、e.B::a及e.D::a都表示虚基类A的成员a 。

8.3 派生类成员——同名问题

- 当派生类有多个基类或虚基类时，基类或虚基类的成员之间可能出现同名；派生类和基类或虚基类的成员之间也可能出现同名。
- 当多个数据成员或函数成员的名称相同时，必须通过面向对象的作用域解析，或者用作用域运算符::指定要访问的成员，否则就会引起二义性问题。
  - 基类成员间的同名问题 【例8.2】
  - 派生类成员与基类成员同名问题【例8.4】
  - 虚基类与基类的成员同名问题【例8.5】

【例8.2】多继承基类成员间的同名问题

```
struct A{ int a; };
struct B{ int a; };
struct C: A, B{
    int c;
    int setc(int);
};
int C::setc(int c) {
    C::c=c;                  //指定访问数据成员C::c
    return c;
}
void  main(void)
{
    C c;
    int i=c.a;               //错误，出现二义性访问
    i=c.A::a;                //正确，全名指定访问A::a
    i=c.B::a;                //正确，全名指定访问B::a
}
```

考虑：若在C类中增加成员int a;
则：主函数中 int i=c.a;
是否正确？为什么？

return

【例8.4】多继承派生类的成员与基类成员同名问题

```
struct A{
    int a, b, c, d;
};
struct B{
    int b, c;
protected:
    int e;
};
class C: public A, public B{
    int a;
public:
    int b;
    int f(int c);
};
int C::f(int c)
{
    int i=a;          //访问C::a，其优先级高于A::a
    i=A::a;
    i=b+c;            //访问C::b和函数参数c
    i=A::b+B::b;      //限定访问数据成员
    return A::c;
}
```

```
int main(void)
{
    C   x;
    int  i=x.A::a;
    i=x.b;          //访问C::b
    i=x.A::b+x.B::b;
    i=x.A::c;
    return i;
}
```

当派生类成员和基类成员同名时，优先访问作用域小的成员，即，优先访问派生类的成员。

类作用域小→大顺序：

表达式→函数成员→派生类→基类→虚基类

【例8.5】虚基类与基类的成员同名问题

```
#include <iostream.h>
struct A{
    void f( ) { cout<<"A\n"; }
};
struct B: virtual A{
    void f( ) { cout<<"B\n"; }
};
struct C: B{ };
struct D: C, virtual A{  };
void  main(void)
{
    D  d;
    B  *pb=&d;
    D  *pd=&d;
    pb->f( );       //调用B::f( )
    pd->f( );       //调用B::f( )
}
```

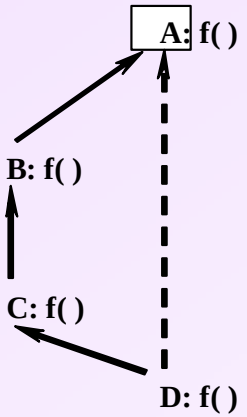


图8.3 派生类D的派生树

根据面向对象的作用域规则，
如果虚基类和基类的函数成员同名，
则优先访问的是基类的函数成员。

输出：

```
B
B
```

8.4 构造与析构

1. 在考虑多继承派生类构造函数的执行顺序时，必须注意派生类可能有虚基类、基类、对象成员、const成员以及引用成员。当虚基类、基类和对象成员的构造函数带参数时，派生类必须定义自己的构造函数，而不能利用C++提供的缺省构造函数。(因为缺省构造函数只能调用基类的无参构造函数，而上述基类的构造函数都是有参数的)
2. 对于虚基类、基类和对象成员来说，如果它们没有定义自己的构造函数，则编译程序就会为它们提供缺省的无参构造函数。对于虚基类、基类和对象成员的无参构造函数，无论它们是自定义的还是由编译程序提供的，都会被派生类构造函数按定义顺序自动地调用。

派生类对象的构造顺序描述

①按定义顺序自左至右、自下而上地构造所有虚基类；

②按定义顺序构造派生类的所有直接基类；

③按定义顺序构造派生类的所有数据成员，包括对象成员、const成员和引用成员；

④执行派生类自身的构造函数体；

⑤如果虚基类、基类、对象成员、const成员以及引用成员又是派生类对象，重复上述派生类对象的构造过程，但同名虚基类对象在同一棵派生树中仅构造一次。 分析【例8.6】

虚基类→基类→按定义顺序构造派生类的所有数据成员→派生类自身构造函数

（对象成员、const成员和引用成员）

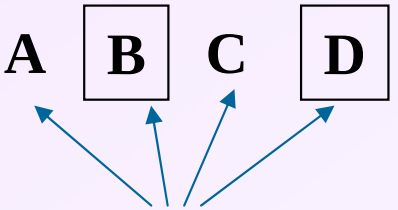
析构派生类对象的顺序与构造逆序

【例8.6】多继承派生类的构造过程

```
#include <iostream.h>
struct A {
    A( ) { cout<<'A';}
};
struct B {
    B( ) { cout<<'B';}
};
struct C {
    int a;
    int &b;
    const int c;
    C(char d): c(d), b(a)
     { a=d; cout<<d;}
};
struct D{
    D( ) { cout<<'D';}
};
```

```
struct E: A, virtual B, C, virtual D{
    A x, y;
    B z;
    E( ):z( ), y( ), C('C')
    {
        cout<<'E';
    }
};

void main(void)
{
    E  e;
}
```



E{x(A) y(A) z(B)}

输出：

BDACAABE

B→D→A→C→x(A)→y(A)→z(B)→E

8.5 类的存储空间

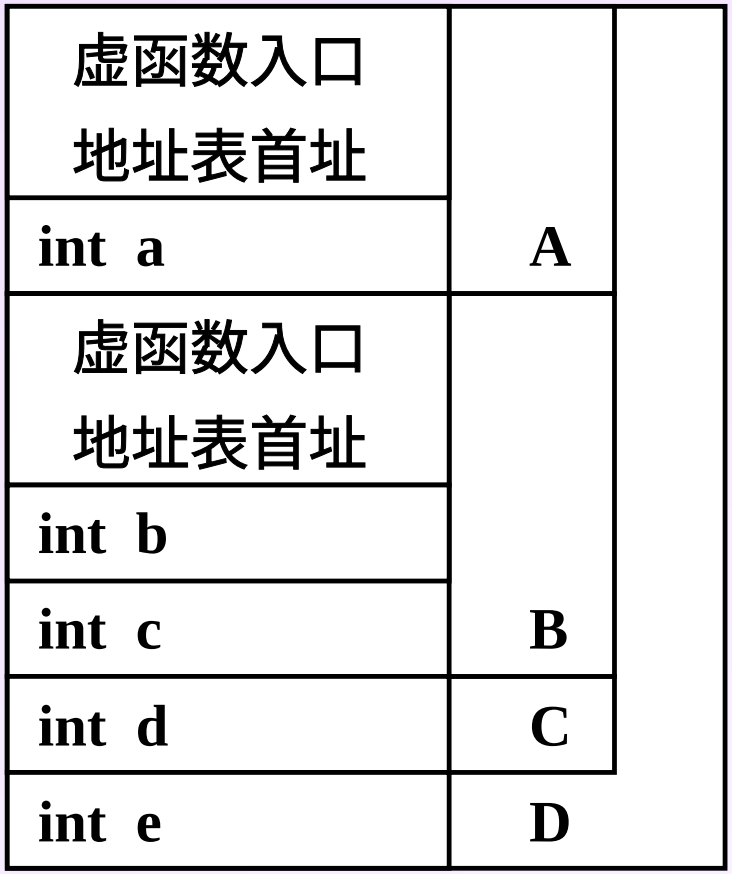
- 派生类无虚基类的情况下

① 若派生类的第一个基类建立了虚函数入口地址表（VFT），则派生类就共用该表首址所占用的存储单元；

② 若派生类的第一个基类没有定义虚函数，派生类就在建立完所有基类的存储空间之后，根据派生类中是否定义了新的虚函数，确定是否为VFT表首址分配一个存储单元，然后为新定义的数据成员建立存储空间。分析【例8.8】

【例8.8】①无虚基类的多继承派生类存储空间的建立

```
#include <iostream.h>
class A{
   int a;
public:
   virtual void f1( ) { };
};
class B{
   int b, c;
public:
   virtual void f2( ) { };
};
class C{
   int d;
public:
   void f3( ) { };
};
class D: A, B, C{
   int e;
public:
   virtual void f4( ) { };
};
```



派生类D的存储空间示意图

D的第1个基类A已建立了VFT首址。D共用该表首址所占用的存储单元。

sizeof(D)=sizeof(A)+sizeof(B)+sizeof(C)+sizeof(e)

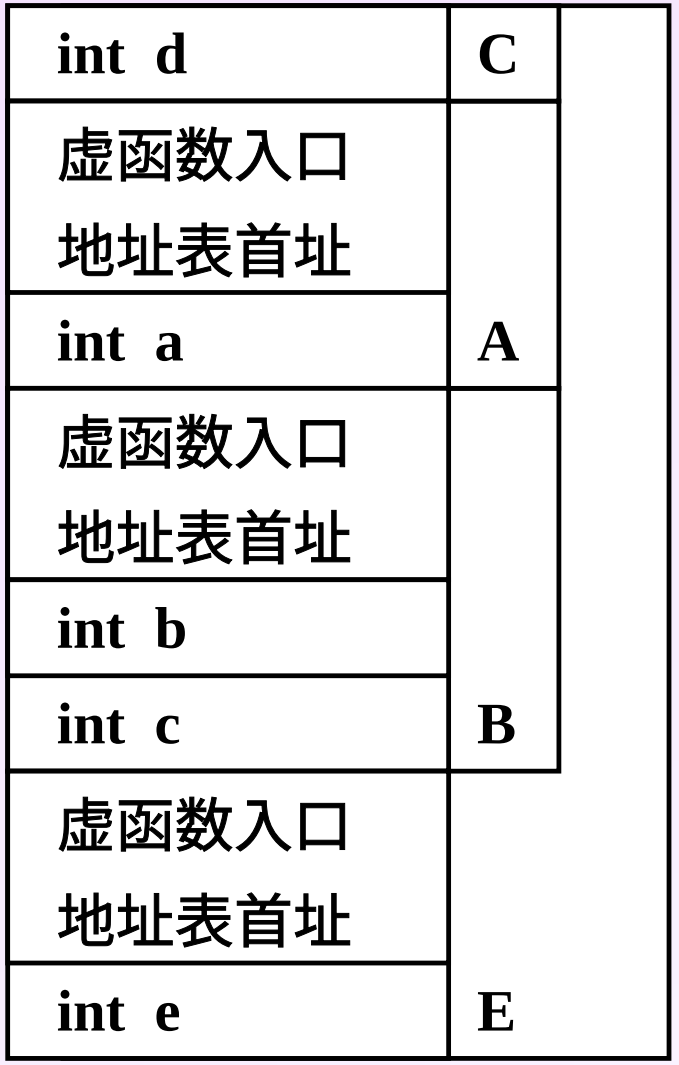
sizeof(A)=sizeof(void *)+sizeof(a)

sizeof(B)=sizeof(void *)+sizeof(b)+sizeof(c)

sizeof(C)=sizeof(d)

【例8.8】②无虚基类的多继承派生类存储空间的建立

```
#include <iostream.h>
class A{
    int a;
public:
    virtual void f1( ) { };
};
class B{
    int b, c;
public:
    virtual void f2( ) { };
};
class C{
    int d;
public:
    void f3( ) { };
};
class E: C, A, B{
    int f;
public:
    virtual void f5( ) { };
};
```



派生类E的存储空间示意图

E的第1个基类C没有定义虚函数，在E中为VFT首址分配一个存储单元

sizeof(E)=sizeof(C)+sizeof(A)+sizeof(B)
+sizeof(void *)+sizeof(e)

8.5 类的存储空间（续）

- 派生类有虚基类的情况下，虚基类的存储空间建于派生类的尾部，且按虚基类的构造顺序建立。具体步骤如下：

① 派生类依次处理每个直接基类或虚基类，如果为直接基类，则为其建立存储空间，如果为直接虚基类则建立一个到虚基类的偏移。

② 如果派生类继承的第一个类为基类，且该基类定义了虚函数地址表，则派生类就共享该表首址占用的存储单元。对于其他任何情形，派生类在处理完所有基类或虚基类后，根据派生类是否新定义了虚函数，确定是否为该表首址分配存储单元。

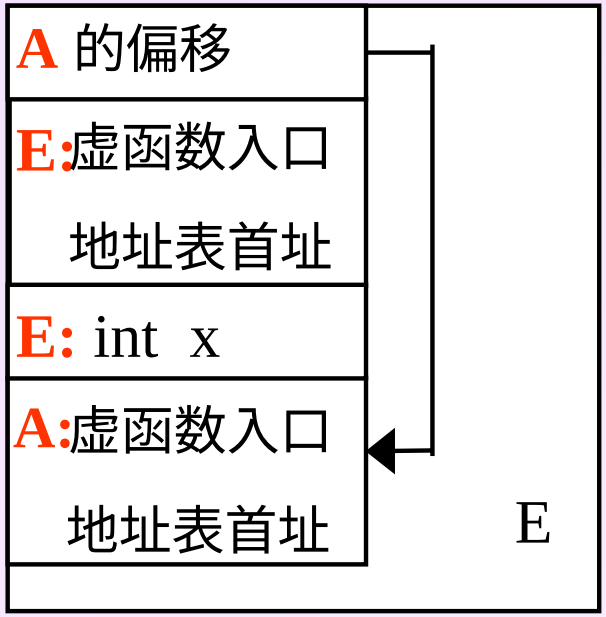
③ 派生类依次处理自定义的数据成员，为每个数据成员建立相应的存储空间。

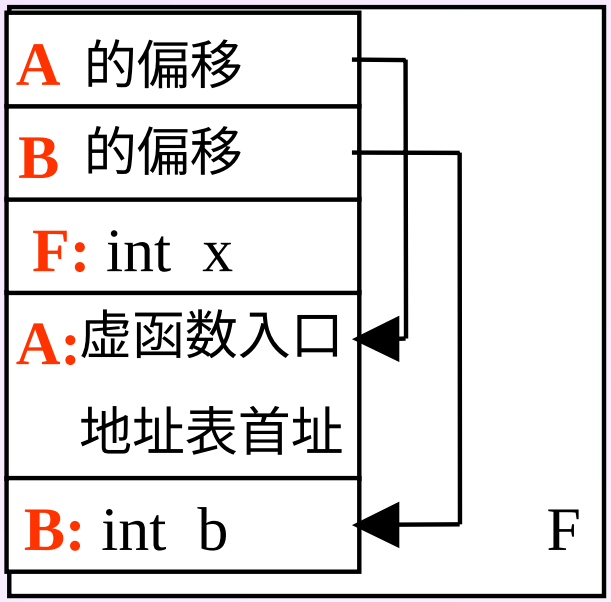
④ 派生类根据虚基类偏移的建立顺序，依次为虚基类建立存储空间，同名虚基类仅在派生类存储空间内建立一次。分析【例8.9】

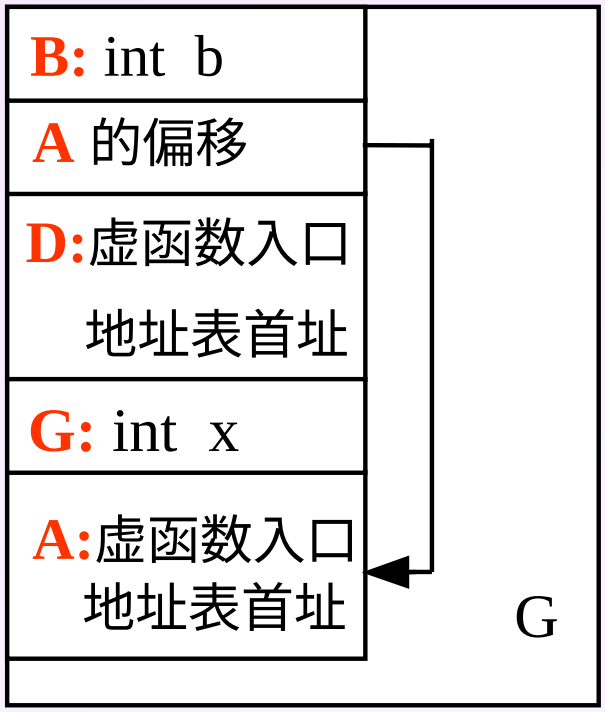
⑤ 如果直接基类和虚基类又是派生类，则在派生类的存储空间内重复步骤①至⑤。如果数据成员又为派生类类型，则在数据成员的存储空间内重复步骤①至⑤。

【例8.9-1】含有虚基类的多继承派生类存储空间的建立

```
#include <iostream.h>
struct A{
   virtual void fa( ) { };
};
struct B{
   int b;
   void fb( );
};
struct E: virtual A{
   int  x;
   virtual void fe( ) { };
};
struct F:virtual A, virtual B{
   int  x;
   void ff( ) { };
};
struct G: B, virtual A{
   int  x;
   virtual void fg( ) { };
};
```

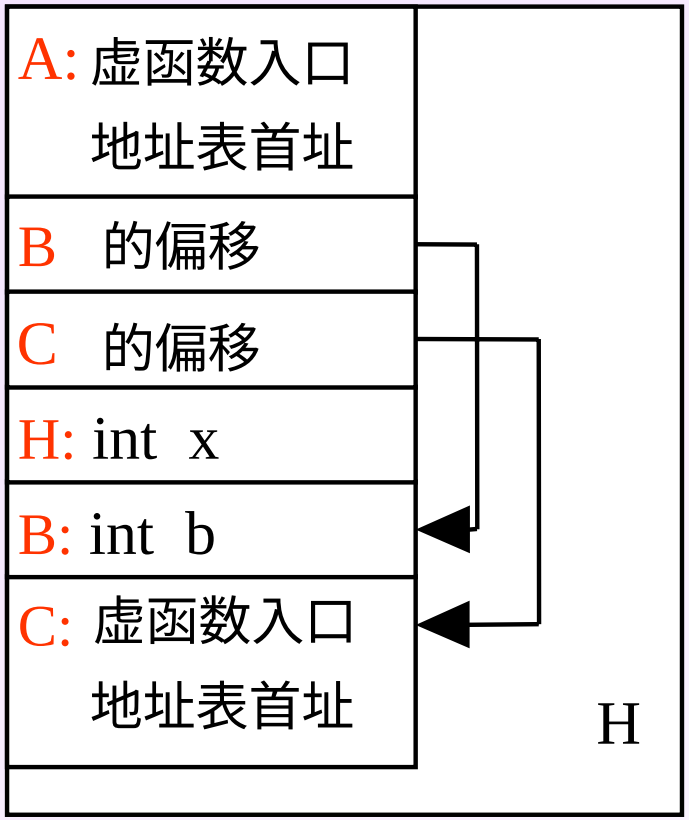






【例8.9-2】含有虚基类的多继承派生类存储空间的建立

```
#include <iostream.h>
struct A{
    virtual void fa( ) { };
};
struct B{
    int b;
    void fb( );
};
struct C{
    virtual void fc( );
};
struct H: A, virtual B, virtual C{
    int  x;
    void ff( ) { };
};
```



【例8.9-3】含有虚基类的多继承派生类存储空间的建立

```
#include <iostream.h>
struct A{
    virtual void fa( ) { };
};
struct B{
    int b;
    void fb( );
};
struct C{
    virtual void fc( );
};
struct D{
    virtual void fd( );
};
struct I: A, virtual B, virtual C, D{
    int  x;
    virtual void fg( ) { };
};
```