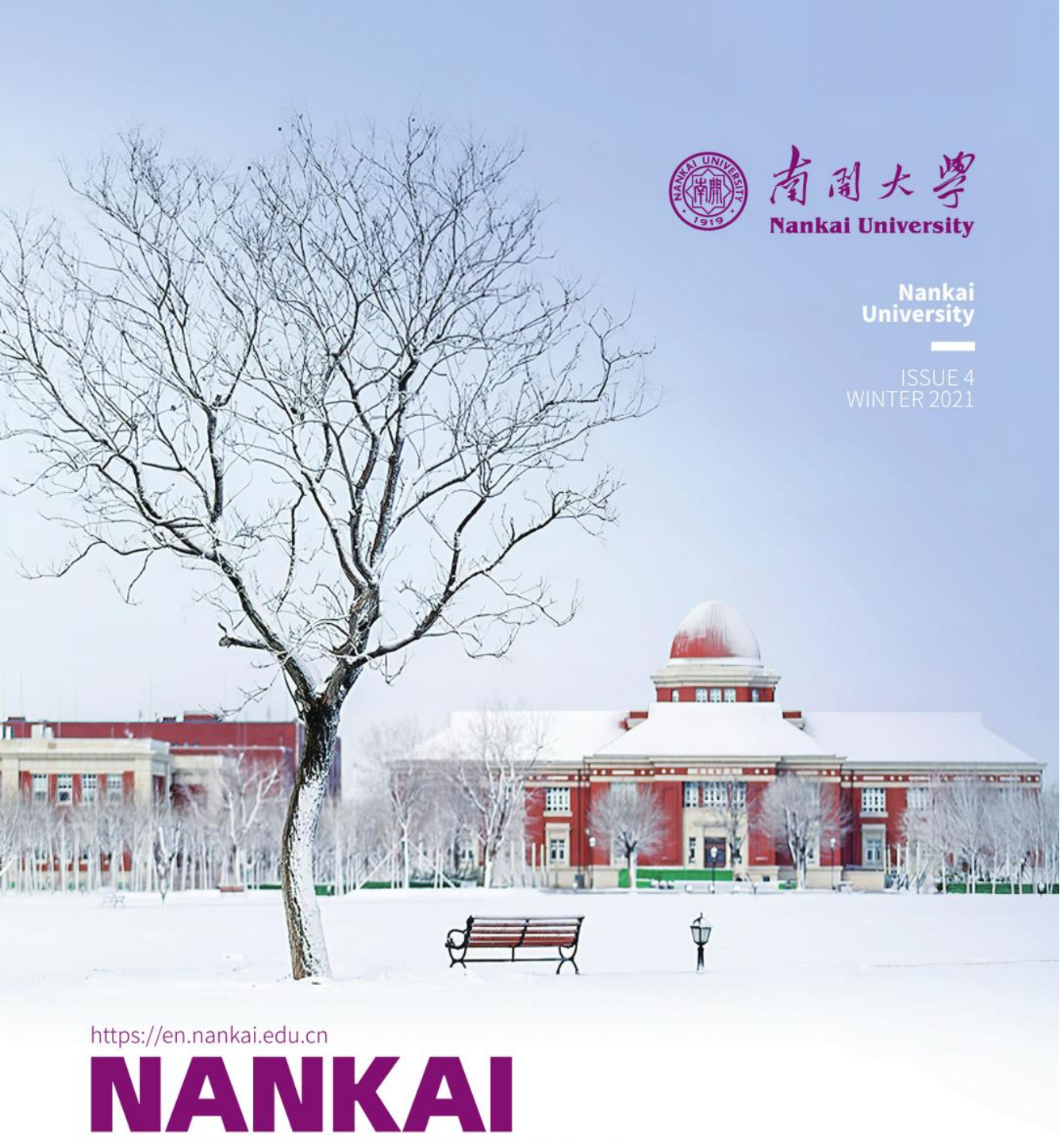

08-继承与复合

龚成

cheng-gong@nankai.edu.cn

继承方式

派生类说明

派生类继承方式

继承成员属性

派生类说明

- 派生类说明也是类说明，不过需要指明它所继承的基类，同时在类的成员中可增加一类保护成员。其一般格式为：

```
class<类名>:<基类说明表> {
private:
    <私有成员表>
protected:
    <保护成员表>
public:
    <公有成员表>
};
```

- 基类说明表：列出所给类的基类。 每个基类说明格式为：<派生方式><基类名>。

派生类继承方式

- 派生方式：公有派生、保护派生或私有派生。Public 表示公有派生，protected 表示保护派生，private 表示私有派生。
- 派生方式决定了从基类继承过来的成员在派生类中的封装属性。派生类可以**有1至多个基类**，或者说派生类是由1至多个基类而派生出来的类（类型）。通过派生方式来指定各基类成员的被继承方式。
- **public 派生方式：**使基类的公有成员和保护成员在派生类中仍然是公有成员和保护成员，而基类的私有成员不可在派生类中被存取。
- **protected 派生方式：**使基类的公有成员和保护成员在派生类中都变为保护成员，而基类的私有成员不可在派生类中被存取。
- **private 派生方式：**使基类的公有成员和保护成员在派生类中都变为私有成员，而基类的私有成员不可在派生类中被存取。

继承成员属性

- 基类成员在各自派生类中的存取权限

表 8.1　基类成员在各自派生类中的存取权限

派生方式（基类的被继承方式）	在基类中的存取权限	在派生类中的存取权限
public	public	public
public	potected	protected
public	private	(inaccessible)
potected	public	potected
potected	potected	protected
potected	private	(inaccessible)
private	public	private
private	potected	private
private	private	(inaccessible)

继承成员属性

- 派生类中接收了基类的所有成员，通过派生方式的选择，进而又指定了被继承过来的每一个成员的新存取权限。派生类中可出现如下4 种成员（比普通类多了一种）：
  - **不可访问的成员：**基类的 private 私有成员和不可访问成员被继承过来后，这些成员在派生类中是不可访问的。(**注意：不可访问的成员也确实被继承到派生类中，可以通过测试派生类存储空间验证。**)
  - **私有成员：**包括在派生类中新增加的 private 私有成员以及从基类私有继承过来的基类的 public 以及 potected 成员，这些成员在派生类中是可以访问的，但不能被其派生类访问。
  - **保护成员：**包括在派生类中新增加的potected 保护成员以及从基类继承过来的成员，如公有继承过来的基类的potected 成员以及保护继承过来的基类的public 和 potected 成员。
  - **公有成员：**包括在派生类中新增加的 public 公有成员以及从基类公有继承过来的基类的 public 成员。

继承成员属性

- 派生类会继承基类的除掉**构造函数和析构函数**之外的所有成员，
- 基类的 private 成员在派生类中根本不可被访问，要想让某成员在派生类中能够被访问，就必须在基类中将其说明为 potected 或 public 属性。
- 派生方式的不同，也使得被继承过来的那些成员在派生类中具有了新的，可能与基类不同的存取权限。若以private 派生方式派生出的类作为基类再去派生它的派生类时，则在其派生类中，根本无法存取其基类（ private 派生方式派生出的父类）中的任何一个成员。

注意事项

- 继承与派生关系，有下面几点应注意：
  - 一个类可以派生出多个派生类。
  - 一个类可有一个或多个基类，称为单一继承和多重继承。
  - 派生类又可有派生类，称为多级继承。
  - 继承关系不可循环。例如，类 A继承类 B，类 B继承类 C，类 C 又继承类 A，这是非法的。
  - 基类的友元关系和基类的构造函数和析构函数都不能被派生类所继承。

举例说明

```
class B {
        int priDat;
protected:
        int proDat;
public:
        int pubDat;
};
class D11 : public B {
        void f11() {
           pubDat=11;          //pubDat 仍具有 public 存取权限（公有继承）
           proDat=12;          //proDat 仍具有 protected 存取权限（公有继承）
           priDat=13;          //ERROR!   不可存取基类的 private 成员
        }
};
class D21 : public D11 {
        void f21() {
           pubDat=121;         //仍为 public
           proDat=122;         //仍为 protected
           priDat=123;         //ERROR!   仍无法访问基类的 private 成员
        }
};
```

```
class D12 : protected B {
        void f12() {
           pubDat=21;          //pubDat 变为 protected 存取权限（保护继承）
           proDat=22;          //proDat 仍为 protected 存取权限（保护继承）
           priDat=23;          //ERROR!   不可存取基类的 private 成员
        }
};
class D22 : public D12 {
        void f22() {
           pubDat=221;         //仍为 protected
           proDat=222;         //仍为 protected
           priDat=223;         //ERROR!   在继续派生的新类中仍无法访问基类的 priva
        }
};
class D13 : private B {
        void f13() {
           pubDat=31;          //pubDat 变为 private 存取权限（私有继承）
           proDat=32;          //proDat 变为 private 存取权限（私有继承）
           priDat=33;          //ERROR!   不可存取基类的 private 成员
        }
};
```

举例说明

```
class D12 : protected B {
    void f12() {
        pubDat=21;      //pubDat 变为 protected 存取权限（保护继承）
        proDat=22;      //proDat 仍为 protected 存取权限（保护继承）
        priDat=23;      //ERROR!  不可存取基类的 private 成员
    }
};
class D22 : public D12 {
    void f22() {
        pubDat=221;     //仍为 protected
        proDat=222;     //仍为 protected
        priDat=223;     //ERROR!  在继续派生的新类中仍无法访问基
    }
};
class D13 : private B {
    void f13() {
        pubDat=31;      //pubDat 变为 private 存取权限（私有继承）
        proDat=32;      //proDat 变为 private 存取权限（私有继承）
        priDat=33;      //ERROR!  不可存取基类的 private 成员
    }
};
```

```
class D23 : public D13 {
    void f23() {
        pubDat=321;     //ERROR!  不可存取基类的 private 成员
        proDat=322;     //ERROR!  不可存取基类的 private 成员
        priDat=323;     //ERROR!  在继续派生的新类中仍无法访问基类的 privat
    }
};
void main() {
    B ob0;
    ob0.pubDat=1;       //可以存取公有成员
    ob0.proDat=2;       //ERROR!  不可存取 protected 成员
    ob0.priDat=3;       //ERROR!  不可存取 private 成员
    D11 d11;   D21 d21;   D22 d22;   D23 d23;
    d11.pubDat=4;       //可以存取公有成员
    d11.proDat=5;       //ERROR!  不可存取 protected 成员
    d11.priDat=6;       //ERROR!  不可存取 private 成员
    d21.pubDat=7;       //可以存取公有成员
    d21.proDat=8;       //ERROR!  不可存取 protected 成员
    d22.pubDat=9;       //ERROR!  不可存取 protected 成员
    d23.pubDat=10;      //ERROR!  不可存取 private 成员
}
```

举例说明

- Parent和Child分别占用几个字节的空间呢？

```
class Parent{
protected:
        int a;
public:
        Parent(int a){a=a;}
};

class Child: public Parent{
private:
        int b;
public:
        Child(int a, int b):Parent(a),b(b){}
};
```

派生类的构造函数

派生类的构造

- 派生类对象的创建和初始化与基类对象的创建和初始化有关：虽然类 B的派生类 A只可以存取类 B的保护成员和公有成员，但是，**派生类 A 的对象的创建，则必须包含着基类 B 的对象的创建，也包括对这里面基类 B 的私有成员的初始化构造（虽然无法访问）**。
- 构造派生类对象时，要对其**基类数据成员、所含对象成员的数据成员以及其他的新增数据成员**一起进行初始化，由派生类的构造函数通过**初始化符表**来完成的。

派生类的构造

- 派生类的构造函数的一般格式如下：

```
<派生类名>(<参数总表>): <初始化符表>  {
    <构造函数体>
}
```

- 而〈初始化符表〉按如下格式构成：

〈基类名 1〉(〈基类参数表 1〉)，…，〈对象成员名 1〉(〈对象成员参数表 1〉)，...

- 其中的<基类参数表 i >是与<基类名 i >所指**基类的某个构造函数相呼应的参数表**；而〈对象成员参数表i>是与<对象成员名 i >所属**类的某个构造函数相呼应的参数表**。
- 构造函数中出现的基类名与对象成员名的**次序无关紧要，各自出现的顺序可以任意调整，其调用顺序与派生顺序有关，与构造函数内顺序无关。**

派生类的构造

- 通过继承可使派生类中“拥有”一个基类的对象；通过将类中的数据成员说明成是另一个类的对象时，也使得在该类中“拥有”了那一个类的对象，但两者在概念和使用上既有关联又有较大的区别。
  - 当类的数据成员为另一个类的对象时，意味着 A类对象中总含有一个 B类对象（对象成员），它们属于**整体与部分的关系（“has a”关系）**。如汽车和马达，马达是汽车的一部分。可以称包含类对象的类为“组装类”。（我拥有你：两个对象）
  - 当使用类的继承产生派生类后，派生类对象中也总“拥有”基类的对象成员，这意味着**派生类的对象必然是一个基类对象（“is a”关系）**，如汽车和轿车，首先轿车就是汽车（具有汽车的所有特征），另外它相对于汽车还具有另外一些特殊属性。（我就是你：一个对象）

派生类的构造

- 使用派生类和对象成员时：
  - 一是要注意构造函数和析构函数的执行次序：派生类构造函数应负有的“责任”——**既要对所包含的每一个对象成员的初始化负责（若含有对象成员），又要对其直接基类的初始化负责**。
  - 二是注意基类与对象数据成员的关系的不同，决定了对其对象成员（或基类成员）的访问方式以及对其对象可施加操作的某些不同。**对基类成员可以直接使用this指针访问，对于对象数据成员必须使用对象名调用。**

派生类的构造

- 创建派生类对象时系统按下列步骤工作（**注意顺序**）：
  - 调用各基类的构造函数，调用顺序按照它们被继承时声明的顺序（从左到右）。
  - 再调用各对象成员的构造函数，调用顺序按照它们在派生类中声明的顺序（从左到右）。
    - 注意：在派生类中声明对象成员的顺序可以与派生类构造函数所列对象成员的顺序不同，它们之间没有必然联系。
  - 最后调用派生类自己的构造函数（执行其函数体）。

派生类的析构函数

派生类的析构

- 构造函数和析构函数用来创建和释放该类的对象，当这个类是派生类时，其对象的创建和释放应与其基类对象及成员对象相联系。
- 释放派生类对象时系统的工作步骤与构造相反：
  - 先调用派生类自己的析构函数。
  - 再调用对象成员的析构函数，调用顺序按照它们在派生类中声明的相反顺序（从右到左）。
  - 最后调用各基类的析构函数，调用顺序按照它们被继承时声明的相反顺序（从右到左）。

关系继承

友元关系

静态成员

赋值关系（赋值构造函数）

友元关系的继承

- 前文已经指出，基类的友元不继承。即如果基类有友元类或友元函数，则其派生类不会因继承关系也一定有此友元类或友元函数。
- 如果基类是某类的友元，则这种友元关系是被继承的。即被派生类继承过来的成员，如果原来是某类的友元，那么它作为派生类的成员仍然是某类的友元。

静态成员的继承

- **如果基类的静态成员是公有的或是保护的，则它们被其派生类继承为派生类的静态成员。**
- 这些成员通常用“<类名>::<成员名>”方式引用或调用。这些成员无论有多少个对象被创建，都只有一个拷贝。**它为基类和派生类的所有对象所共享。**

```
class A{
    public:
    static int a;
};
int A::a = 0;
```

```
class B: public A{
    public:
    static int b;
};
int B::b = 0;
```

```
int main(){
    B b;
    A a;
    b.a=10;
    cout<<a.a<<","<<b.a<<endl;
    return 0;
}
```

赋值关系

- 基类对象和派生类对象之间的赋值关系如下：
  - 允许派生类对象给基类对象赋值：基类对象=派生类对象，但只赋“共性成员”部分。注意：派生类对象除含有基类对象的成员外，通常还具有自己特有的成员以区别于其基类。
  - 不允许基类对象赋值给派生类对象！
  - 允许基类的指针指向派生类对象地址：指向基类对象的指针=派生类对象的地址。指向基类型的指针可以直接访问基类成员部分，但访问非基类成员部分时，要经过指针类型的强制转换。
  - 不允许派生类指针指向基类对象地址！
  - 允许派生类的对象赋值给基类的引用对象：基类的引用 = 派生类对象。允许该形式的赋值即派生类对象可以初始化基类的引用。注意：**通过引用只可以访问基类成员部分，而不可访问非基类成员部分**，因为不可将基类的引用强制转换为其派生类类型。
  - 不允许基类对象赋值给派生类对象引用！

赋值关系

- 派生类对象间的赋值操作依据下面的原则：
  - 如果派生类有自己的赋值运算符的重载定义，即按该重载函数处理。
  - 如果派生类未定义自己的赋值操作，而基类定义了赋值操作，则系统自动定义派生类赋值操作，其中基类成员的赋值按基类的赋值操作进行。
  - 二者都未定义专门的赋值操作，系统自动定义缺省赋值操作（浅拷贝）。

```
class A{
    public:
    int* p;
    A(){
        p = new int[10];
    }
    A(A& a){
        p = new int[10];
        for(int i=0;i<10;i++){
            p[i] = a.p[i];
        }
    }
};
```

```
class B: public A{
    public:
    int* q;
    B(){
        q = new int[10];
    }
};
```

```
int main(){
    B b1;
    B b2(b1);
    cout<<b1.p<<"=?"<<b2.p<<endl;
    cout<<b1.q<<"=?"<<b2.q<<endl;
}
```

赋值关系

- 任意类型之间的赋值操作依据下面的原则：
  - **如果类有自己的赋值运算符的重载定义，即按该重载函数处理。**
  - **如果类未定义自己的赋值操作，则按照系统默认来处理：**
    - 同类型之间执行浅拷贝
    - 不同类型之间提示出错

二义性处理

重名成员处理

重名成员处理

- **单一继承时基类与派生类间重名成员的处理：**对派生类而言，在派生类定义范围内以及通过派生类对象访问重名成员时，**不加类名限定时默认为是处理派生类成员**，而要访问基类重名成员时，则要通过**类名限定**。

```
class A{
    public:
    int name;
   };

class B: public A{
    public:
    int name;
    B(){
        this->name = 1;
        this->A::name = 2;
    }
};
```

重名成员处理

- **多重继承情况下两基类间重名成员的处理：**解决方法和单一继承一样，对派生类而言（在派生类定义范围内以及通过派生类对象访问重名成员时），**不加类名限定时默认为是处理派生类成员，而要访问基类重名成员时，则要通过类名限定。**

```
class A1{
    public:
    int name;
   };
class A2{
    public:
    int name;
   };
```

```
class B: public A1, public A2{
    public:
    int name;
    B(): A1(), A2(){
        this->name = 1;
        this->A1::name = 2;
        this->A2::name = 20;
    }
};
```

重名成员处理

- **多级混合继承包含两个基类实例情况的处理：**解决方法和多重继承一样，对派生类而言（在派生类定义范围内以及通过派生类对象访问重名成员时），**不加类名限定时默认为是处理派生类成员，而要访问基类重名成员时，则要通过类名限定**。但是这种方式不能解决更多重的继承问题。

```
class A{
    public:
    int name;
};
```

```
class B1: public A{
    public:
    B1(): A(){
        this->A::name = 2;
    }
};
```

```
class B2: public A {
    public:
    B1(): A(){
        this->A::name = 20;
    }
};
```

```
class C: public B1,B2{
    public:
    int name;
    B1(): B1(), B2(){
        this->name = 1;
        this-> B1::name = 2;
        this-> B2::name = 2;
    }
};
```

```
class D: public C{
    public:
    int name;
    B1(): C(){
        this->name = 1;
        this-> C::name = 2;}
};
```

??

虚基类

虚基类定义

虚基类实例化

虚基类定义

- 虚基类：类 B 作为类 D1，D2，…，Dn的基类，当把类 B定义为派生类 D1，D2，…，Dn 的**虚基类时，各派生类的对象共享其基类 B 的一个拷贝，这种继承称为**共享继承。
- 虚基类是为了解决由混合（多重多级）继承造成的二义性问题，其说明形式需要在继承前添加virtual关键字：

```
class A{…};
class B: virtual public A{…};
class C: virtual public A{…};
class D: public B, public C{…};
```

虚基类定义

- 如此说明之后，类 D 的对象 Dobj它将只包含类 A 的一个拷贝，上一节提到的二义性问题将不再发生。
- 虚基类的说明是在定义派生类时靠**增加关键字“virtual”**来指出的。在使用了虚基类后，系统将进行“干预”，使各派生类的对象共享其基类的同一个拷贝。
- 非虚基类和虚基类继承的D 类对象的存储结构示意：

（（（A）B）（（A）C） D ）➔（（（A） B C ） D ）

```
class A{…};
class B: virtual public A{…};
class C: virtual public A{…};
class D: public B, public C{…};
```

虚基类的实例化

- 若虚基类的构造函数具有参数，则对其任一个直接或间接派生类的构造函数来说，它们的**成员初始化列表中都必须包含有对该虚基类构造函数的直接调用**。
- **为了保证虚基类子对象只被初始化一次，规定只在创建对象的派生类的构造函数中调用虚基类的构造函数，而忽略该派生类的各基类构造函数中对虚基类构造函数的调用。**

```
class ClaA{…};
class ClaB: virtual public ClaA{…};
class ClaC: virtual public ClaA{…};
class ClaD: public ClaB, public ClaC{
    ClaD(): ClaA(…), ClasB(), ClaC(){}
};
```

虚基类的实例化

- 在派生类构造函数的**成员初始化列表**中，若有被调用的**虚基类构造函数，则对它们的调用将优先于非虚基类构造函数**。（记住初始化顺序哈！虚基类优先）
- 例如：但要注意虚基类 ClaA 的构造函数具有参数，在它的间接派生类 ClaD 的构造函数处，其成员初始化列表中还**必须包含对该虚基类构造函数的直接调用，若 ClaA 非虚基类时，则不需要也不可以这样做。**

```
class ClaA{…};
class ClaB: virtual public ClaA{…};
class ClaC: virtual public ClaA{…};
class ClaD: public ClaB, public ClaC{
    ClaD(): ClaA(…), ClasB(), ClaC(){}
};
```

虚基类的实例化

- 若虚基类 ClaA 的构造函数具有参数，ClaD 是虚基类 ClaA 的间接派生类。类 ClaD 的构造函数除去要对其直接基类 ClaB 和 ClaC 的初始化负责外，**还必须直接调用ClaA的构造函数直接对虚基类 ClaA 进行初始化**。

```
class ClaA{…};
class ClaB: virtual public ClaA{…};
class ClaC: virtual public ClaA{…};
class ClaD: public ClaB, public ClaC{
    ClaD(): ClaA(…), ClasB(), ClaC(){}
};
```

虚基类的实例化

- 在该派生类构造函数的成员初始化列表中，对虚基类构造函数的调用“ClaA(…)”将优先于对其他两个非虚基类构造函数的调用（与在初始化成员表中的顺序无关）。注意，**虚基类子对象只是通过“ClaA(…)”被初始化了一次**，该派生类的两个基类构造函数中，通过“ClaB()”及“ClaC()”对虚基类构造函数的调用**都将被忽略**。
- **如果虚基类的构造函数没有参数或者没提供显式的虚基类构造函数**，此时任一直接或间接派生类的构造函数的成员初始化列表中都需要包含有对该虚基类构造函数的直接调用。

```
class ClaA{…};
class ClaB: virtual public ClaA{…};
class ClaC: virtual public ClaA{…};
class ClaD: public ClaB, public ClaC{
    ClaD(): ClaA(…), ClasB(), ClaC(){}
};
```