继承——派生类

复习

- 1.有下面类的说明，有错误的语句是:

```
class X
{
A) const int a;
B) X();
public:
C) X(int val) {a=2};
D) ~X();
};
```

答案：C不正确，应改成 **X(int val) : a(2) {};**

- 2.下列静态数据成员的特性中，错误的是
  A)说明静态数据成员时前边要加关键字static来修饰
  B)静态数据成员在类体外进行初始化
  C)使用静态数据成员时,要在静态数据成员名前加<类名>和作用域运算符
  D)静态数据成员不是所有对象所共有的

答案：D错误，应该是共有的
C正确，可以是 类名::静态成员 或 对象名.静态成员

3. 下面程序段输出结果是？

```
class A
{       static int count;
          …
    public:
        A() { count++; }
        void f(int x) {count+=x};
};

int A::count=1;
A a;
A b;
b.f(5);
cout << a.count << endl;
```

```
int A::count=1;  //count=1
A a;             // count=1+1=2
A b;             //count=2+1=3
b.f(5);          //count=3+5=8
cout << a.count << endl; //输出8
```

继承的概念

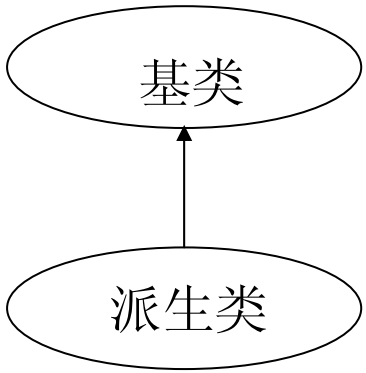
- 软件复用：新的软件可以在原先开发的基础上进行扩充，可以降低成本、提高软件可靠性
- 目前，不加修改地直接复用已有软件比较困难。已有软件的功能与新软件所需要的功能总是有差别的。解决这个差别有下面的途径：
  - 修改已有软件的源代码，它的缺点是：
    - 需读懂源代码
    - 可靠性差、易出错
    - 源代码难以获得
  - 继承机制（Inheritance）：
    - 一种面向对象的软件复用途径
    - 在定义一个新的类时，先把一个或多个已有类的功能全部包含进来，然后再给出新功能的定义或对已有类的某些功能重新定义。
  - C++的继承通过派生类来实现

为什么使用继承？

- 目的在于为软件复用提供有效手段
- 一方面，可以重用先前项目的代码，还可以在此基础上进行修改，提高程序设计灵活性
- 另一方面，如果一个项目使用了几个功能相似的类，可以通过派生类的继承性达到函数和数据的复用，减少重复设计

基类与派生类

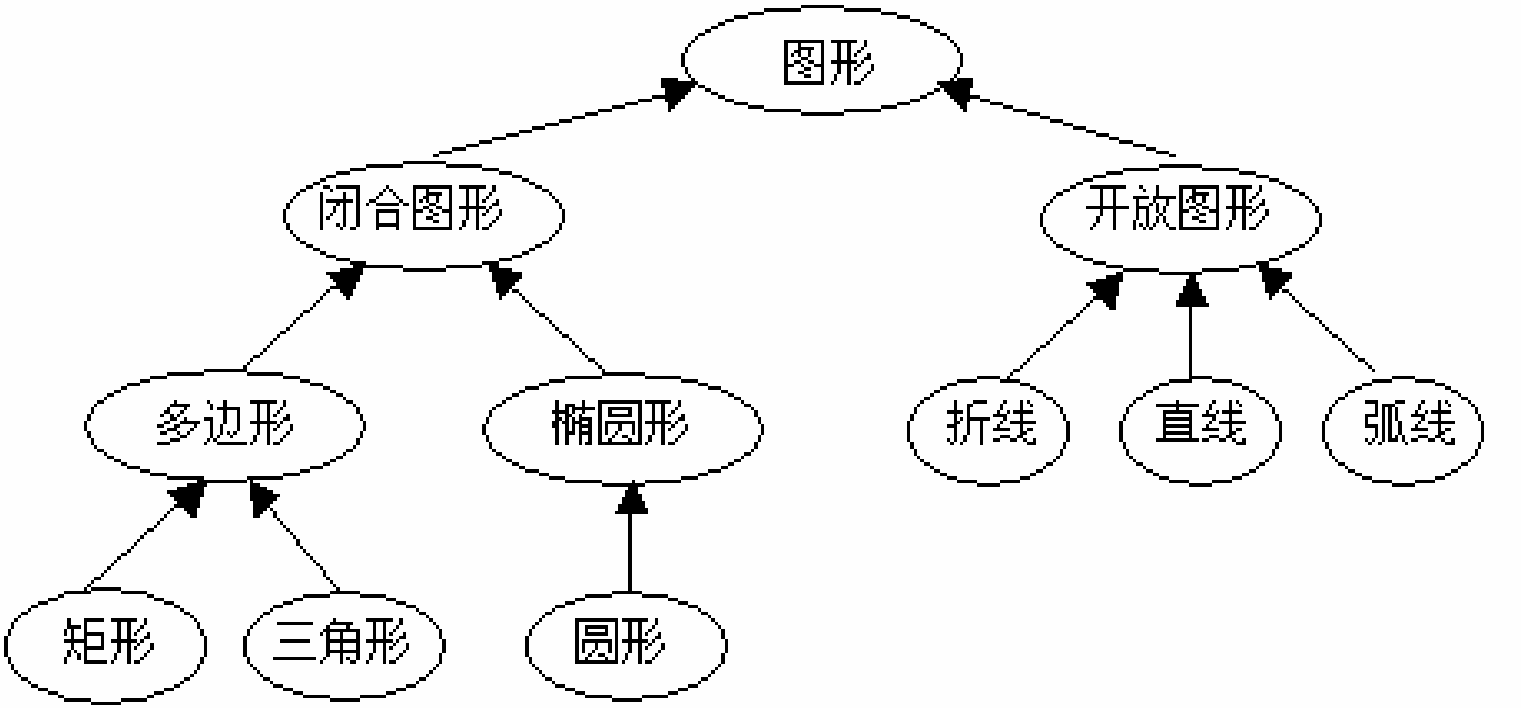
- 在继承关系中存在两个类：基类（或称父类）和派生类（或称子类）。派生类拥有基类的所有特征，并可以定义新的特征或对基类的一些特征进行重定义。



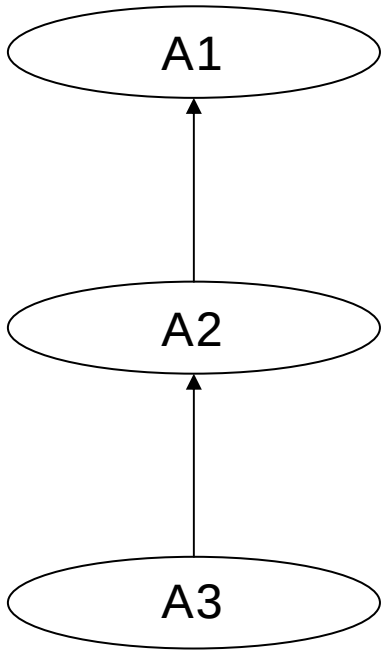
- 继承分为：单继承和多继承
  - 单继承：一个类最多有一个直接基类。
  - 多继承：一个类可以有多个直接基类。

继承对程序设计的支持

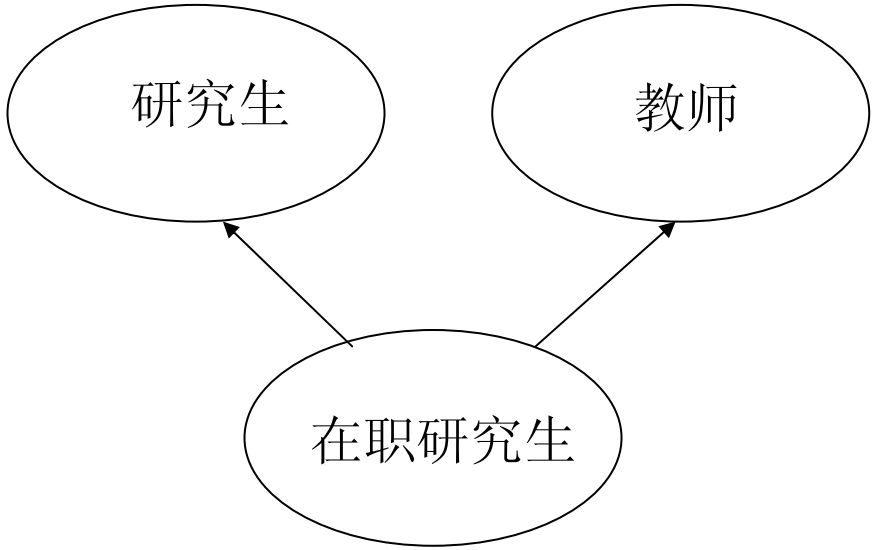
- 继承机制除了支持软件复用外，它还具有下面的作用：
  - 对事物进行分类：层次结构， is-a-kind-of



- 支持软件的增量开发：新的版本可以从老的版本扩展和完善



- 对概念进行组合：例如在职研究生既是教师，又是研究生，综合了两种特征



单继承

- 在定义单继承时，派生类只能有一个直接基类，其定义格式如下：

  class <派生类名>：[<继承方式>] <基类名>

  { <成员说明表>

  };

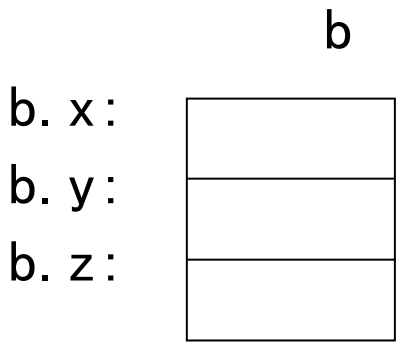
  - <派生类名>为派生类的名字。
  - <基类名>为直接基类的名字。
  - <成员说明表>是在派生类中新定义的成员，其中包括对基类成员的重定义。
  - <继承方式>指出派生类的实例（对象）用户以及派生类的派生类对该派生类的基类成员的访问控制

```
class A //基类
{     int x,y;
  public:
      void f();
      void g();
};
class B: public A //派生类
{     int z; //新成员
  public:
      void h(); //新成员
};
```

关于派生类的说明

- 1. 除了拥有新定义的成员外，派生类还拥有基类的所有成员（基类的构造函数和赋值操作符重载函数除外）。例如：

```
B b;
```




```
b.f();   //A类中的f
b.g();   //A类中的g
b.h();   //B类中的h
```

- 2. 定义派生类时一定要见到基类的定义。

```
class A;  //声明
class B: public A  //Error
{  int z;
  public:
    void h() { g(); }  //Error，编译程序不知道基类中是否有函数g以及函数g的原型。
};
.....
B b; //Error，编译无法确定b所需内存空间的大小。
```

- 3. 友元不具有继承性：如果在派生类中没有显式说明，基类的友元不是派生类的友元；如果基类是另一个类的友元，而该类没有显式说明，则派生类也不是该类的友元。
- 4. 派生类不能直接访问基类的私有成员。例如：

```
class A
{       int x,y;
  public:
        void f();
        void g() { ... x ... }
};
class B: public A
{       int z;
  public:
        void h()
        {    ... x ...   //Error，x为基类的私有成员。
             g();  //OK，通过函数g访问基类的私有成员x。
        }
};
```

同名成员的处理原则

- 情况1：
  - 派生类可以定义新的成员，也可以对基类成员重新进行定义。如果在派生类中新定义的成员与基类的某个成员同名，则在派生类中对该成员的访问是指派生类中定义的成员。例如：

```
class A //基类
{       int x,y;
    public:
        void f() {x=1; cout<< “A.f,” <<x; } ;
        void g() {y=2; cout<< “A.g,” <<y; };
};
class B: public A //派生类
{       int x,z; //x 被重定义
    public:
       int f() {x=3; cout<< “B.f,” <<x; }; //f被重定义
      int h() {
           f(); //使用B类的f()
           g(); //使用A类的g()
           cout<< “B.h,” <<x<<; //输出B类的x
       };
};
B b;
b.f(); //使用B类中的f，输出 B.f, 3
b.g(); //使用A类中的g，输出 A.g, 2
b.h(); //使用B类中的h，输出 B.f,  3  A.g,  2  B.h, 3
```

同名成员的处理原则

- 情况2：
  - 如果派生类中定义了与基类同名的成员，则基类的成员名在派生类的作用域内不直接可见。访问基类同名成员时要用基类名受限。例如：

```
class B: public A
{    int z;
   public:
     void f();
     void h()
     {    f();  //B类中的f
          A::f();  //A类中的f
     }
};
B b;
b.f();  //B类中的f。
b.A::f();  //A类中的f
```

同名成员的处理原则

- 情况3：
  - 即使派生类中定义了与基类同名但参数不同的成员函数，基类的同名函数在派生类的作用域中也是不直接可见的。

```
class B: public A
{       int z;
   public:
        void f(int);
        void h()
        {    f(1);  //OK
             f();  //Error
             A::f();  //OK
        }
};
......
B b;
b.f(1);  //OK
b.f();  //Error
b.A::f();  //OK
```

继承方式

- 在C++中，派生类拥有基类的所有成员。问题是：基类的成员变成派生类的什么成员呢（public、private或protected）？
- 上面的问题由继承方式决定。继承方式在定义派生类时指定：

  class <派生类名>: [<继承方式>] <基类名>
  { <成员说明表>
  };

  - 继承方式可以是：public、private和protected。
  - 默认的继承方式为：private。

继承方式的含义

基类 / 派生类 / 继承方式	public	private	protected
public	public	不可直接访问	protected
private	private	不可直接访问	private
protected	protected	不可直接访问	protected

派生类对基类成员的访问权
原则 （1）对基类私有数据不可见 （2）以最小的权限为准

例：派生类的访问权限

```
class A
{  public:
        void a();
   protected:
        void b();
   private:
        void c();
};
```

1. 公有派生：

```
class B: public A
{   public:
          void d();
//在B类里，a()和d()是public,b()是protected，c()不可见
};
B x; x.a(); //OK
```

2. 保护派生:

```
class B: protected A
{   public:
          void d();
//在B类里，d()是public, a(),b()是protected，c()不可见
};
B x; x.a(); //error,a()是protected
```

3. 私有派生:

```
class B: private A
{   public:
          void d();
//在B类里，d()是public, a(),b()是private，c()不可见
};
B x; x.a(); //error,a()是private
```

- 不管以何种派生方式，B类都可以访问A类的a()和b()，不可访问c()

继承方式的调整

- 在派生类中，可以通过访问声明来调整访问权限
- 在任何继承方式中，除了基类的private成员，都可进行访问控制调整
- 调整的方法：[public : protected : private] 基类名::<基类成员名>，例如

```
class A
{ public:
    void f1();
    void f2();
  protected:
    void g1();
    void g2()
  private:
    void p1();
}
class B: private A
{ public:
    A::g1;
    A::p1;          //Error,private成员不允许调整
  protected:
    A::f2;
    A::g2;
  private:
    A::f1;
}
```

思考：派生类中如何访问基类的私有数据？

- 方法1：通过基类的接口（公有函数）来访问
- 方法2：把基类的private数据改成protected
- 方法3：将派生类或其成员函数声明为基类的友元，例如

```
class B;
class A
{ private: int x;
  …
  friend class B; // 或friend B::f();
}
class B: public A
{
   void f() { x=3 };
}
```

封装与继承的矛盾

- 在派生类中定义新的成员时，往往需要用到基类的一些private成员。（矛盾）
- 在继承机制中，一个类的成员有两种被外界使用的场合：
  - 通过类的对象使用
  - 在派生类中使用
- 在C++中，提供了另外一种类成员访问控制：protected，用它说明的成员不能被对象使用，但可以在派生类中使用。
- protected访问控制缓解了封装与继承的矛盾

```
class A
{  protected:
     int x,y;
   public:
     void f();
};
class B: public A
{    ......
     void h()
     {    f();  //OK
          ... x ...   //OK
          ... y ...   //OK
     }
};
void f()
{  A a;
   a.f();  //OK
   ... a.x ...   //Error
   ... a.y ...   //Error
}
```

派生类对象的初始化

- 派生类对象的初始化由基类和派生类共同完成：
  - 基类的数据成员由基类的构造函数初始化，
  - 派生类的数据成员由派生类的构造函数初始化。
- 当创建派生类的对象时，派生类的构造函数在进入其函数体之前会去调用基类的构造函数。
- 默认情况下，调用基类的默认构造函数，如果要调用基类的非默认构造函数，则必须在派生类构造函数的成员初始化表中指出。
- 原则：先祖先（基类），再客人（成员对象），后自己（派生类）

```
class A
{       int x;
   public:
        A() { x = 0; }
        A(int i) { x = i; }
};
class B: public A
{       int y;
   public:
        B() { y = 0; }
        B(int i) { y = i; }
        B(int i, int j):A(i) { y = j; }
};
......
B b1;  //执行A::A()和B::B()，b1.x等于0，b1.y等于0。
B b2(1);  //执行A::A()和B::B(int)，b2.x等于0，b2.y等于1。
B b3(1,2);  //执行A::A(int)和B::B(int,int)，b3.x等于1，
                  //b3.y等于2。
```

- 如果一个类D既有基类B、又有成员对象类M，则
  - 在创建D类对象时，构造函数的执行次序为：
    B->M->D
  - 当D类的对象消亡时，析构函数的执行次序为：
    D->M->B

例：利用一个线性表类实现一个队列类。

- 线性表由若干元素构成，元素之间有线性的次序关系。

```
class LinearList
{  ......
  public:
    bool insert( int x, int pos );
    bool remove( int &x,  int pos );
    int element( int pos ) const;
    int search( int x ) const;
    int length( ) const;
};
```

- 队列是一种特殊的线性表，插入操作在一端，删除操作在另一端。又称先进先出表（First In First Out，FIFO）
- Queue的实现（继承）：

```
class Queue: private LinearList
{ public:
      bool en_queue(int x)
      {  return insert(x,length());
      }
      bool de_queue(int &x)
      {  return remove(x,1);
      }
};
```

小结

- 继承的概念
- 为什么使用继承？
- 派生类的声明和使用
- 同名成员的处理原则
- 继承方式和访问权限
- 派生类对象的初始化
- 利用线性表类实现一个队列类

复习

- 1.下面关于友元的描述中，错误的是
  A）友元函数可以访问该类的私有数据成员。
  B）一个类的友元类中的成员函数都是这个类的友元函数。
  C）友元可以提高程序的运行效率。
  D）类与类之间的友元关系可以继承

答案：D错误，友元关系不可继承，不对称，也不可传递

- 2. 下列对派生类的描述中，错误的是
  A）一个派生类可以作另一个派生类的基类
  B）派生类至少有一个基类
  C）派生类的成员除了它自己的成员外， 包含了它的基类的成员
  D）派生类中继承的基类成员的访问权限到派生类保持不变

答案：D

3. 下面程序段输出结果是？

```
class A //基类
{  public:
        int x,y;
        int f() {x=1; return x} ;
        int g() {y=2; return y};
};
class B: public A //派生类
{       int x,z;
   public:
        int f() {y=3; return y;};
        int h() { x=f()+g()+y; return x};
};
B b;
cout<<b.h();
```

解答：根据同名处理的原则，
调用h()函数时，f()使用B的f，返回3，
g()使用A的g，返回2，y等于2，
故f()+g()+y=3+2+2=7

- 4.下列A,B,C,D处函数调用正确的是：

```
class A
{   public:
          void a();
    protected:
          void b();
    private:
          void c();
};
class B: private A
{   public:
          void d();
   public: A::b;
};
class C  protected B
{
   void f()
   {
A)      a();
B)      b();
C)      c();
D)      d();
   }
}
```

解答：
类B: public：d和b， private: a，c不可见
类C：protected d,b, 其他不可见
所以B,D正确，A,C错误

子类型

- 子类型：
  - 如果一个类型S是另一个类型T的子类型，则对用T表达的所有程序P，当用S去替换程序P中的所有T时，程序P的功能不变。
  - 一个类型的操作也适合于它的子类型以及一个子类型的值可以赋值或作为函数参数传给基类型变量。
- 继承可以实现子类型：派生类可以看成是基类的子类型
- 子类型的作用：
  - 发给基类对象的消息也能发给派生类对象
  - 在需要基类对象的地方可以用派生类对象去替代

例如，假设A是基类，B是A的派生类，f是A的成员函数，g是B的成员函数

```
A a,*p;
B b,*q;
b.f();  //如果B中有f，则调用B的f；否则调用A的f。
a = b;  //用b去改变a的状态，属于B但不属于A的数据成员将被忽略。
p = &b;  //A类指针p指向B类对象b。
b = a;  //Error，它将导致b有不确定的成员数据（a中没有这些数据）。
q = &a;  //Error，它将导致通过q向a发送它不能处理的消息，如：q->g();
```

多继承

多继承

- 对于下面的两个类A和B：

```
class A
{   int m;
  public:
    void fa();
};
class B
{   int n;
  public:
    void fb();
};
```

- 如何定义一个类C，它包含A和B的所有成员，另外还拥有新的数据成员r和成员函数fc？

- 用单继承实现：

```
class C: public A
{    int n,r;
   public:
     void fb();
     void fc();
};
```

  - 或者

```
class C: public B
{    int m,r;
   public:
     void fa();
     void fc();
};
```

- 不足之处：
  - 概念混乱
  - 易造成不一致

- 用多继承实现：

```
class C: public A, public B
{   int r;
  public:
    void fc();
};
```

- 多继承是指派生类可以有一个以上的直接基类。多继承的派生类定义格式为：

```
class <派生类名>:  [<继承方式>]<基类名1>,
                                [<继承方式>]<基类名2>, …
{ <成员说明表>
};
```

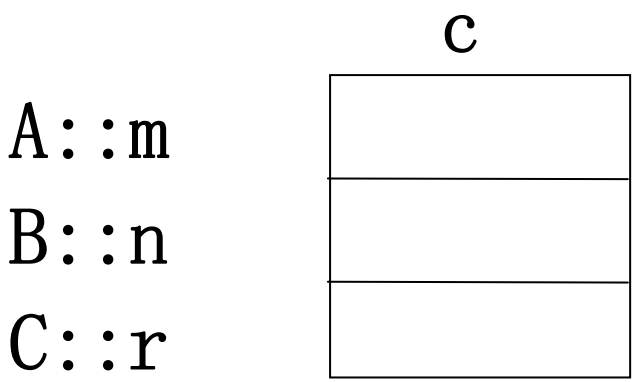
  - 继承方式及访问控制的规定同单继承。
  - 派生类拥有所有基类的所有成员。
  - 基类的声明次序决定：
    - 对基类构造函数/析构函数的调用次序
    - 对基类数据成员的存储安排。

多继承的初始化

- 多继承的构造函数的调用顺序：
  - （1）先祖先（基类），再客人（成员对象），后自己（派生类）
  - （2）多个基类的初始化顺序：按照定义的顺序来区分先后
- 析构函数和构造函数相反

```
class A
{       int m;
   public:
        void fa();
};
class B
{       int n;
   public:
        void fb();
};
class C: public A, public B
{       int r;
   public:
        void fc();
};
.......
C c;
```

- 对象c的内存空间布局是：



- 构造函数的执行次序是：
  - A()、B()、C()　　（A()和B()实际是在C()的成员初始化表中调用。）
- 下面的操作是合法的：

```
c.fa();
c.fb();
c.fc();
```

多继承中的同名问题(二义性)

```
class A
{     ……
   public:
       void f();
       void g();
};
```

```
class B
{     ……
   public:
       void f();
       void h();
};
```

```
class C: public A, public B
{   ……
   public:
     void func()
     {   f(); //Error，是A的f，还是B的f?
     }
};
……
C c;
c.f();  //Error，是A的f，还是B的f?
```

- 解决名冲突的办法是：基类名受限

```
class C: public A, public B
{  ......
  public:
    void func()
    {     A::f(); //OK，调用A的f。
          B::f(); //OK，调用B的f。
    }
};
......
C c;
c.A::f(); //OK，调用A的f。
c.B::f(); //OK，调用B的f。
```

重复继承——虚基类

- 多继承会造成重复继承问题：下面的类D从类A继承两次，称为重复继承。

```
class A
{  int x;
   ......
};
class B: public A { ... };
class C: public A { ... };
class D: public B, public C { ... };
```

- 重复继承会带来二义性：上面的类D将包含两个x成员：B::x和C::x。
- 为什么要使用虚基类？
  可以消除重复继承造成的二义性

虚基类的概念和定义

- 虚基类的概念
  - 当在多条继承路径上有一个公共的基类，这个公共的基类就会产生多个实例，若只想保存这个基类的一个实例，可以将这个公共基类声明为虚基类
- 定义方法：基类前加上virtual关键字
- 例：如果需要类D中只有一个x，则应把A定义为B和C的虚基类：

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

虚基类的初始化

- 虚基类的初始化和一般多继承的初始化语法一样，但构造函数调用次序不同
- 构造函数的调用次序
  - 虚基类的构造函数优先于非虚基类构造函数调用
  - 若同一层次包含多个虚基类，则按照他们的声明先后顺序调用
  - 若虚基类由非虚基类派生而来，则仍然先调用基类构造函数，再调用派生类构造函数

例：虚基类的初始化

```
class A
{  int x;
  public:
   A(int i) { x = i; }
};
class B: virtual public A
{  int y;
public:
   B(int i): A(1) { y = i; }
};
class C: virtual public A
{  int z;
  public:
   C(int i): A(2) { z = i; }
};
```

```
class D: public B, public C
{  int m;
public:
   D(int i, int j, int k): B(i), C(j), A(3) { m = k; }
};
class E: public D
{  int n;
  public:
   E(int i, int j, int k, int l): D(i,j,k), A(4) { n = l; }
};
.......
```

D d(1,2,3); //这里，A的构造函数由D调用，d.x初始化为3。

- 调用的构造函数及它们的执行次序是：
  A(3)、B(1)、C(2)、D(1,2,3)

E e(1,2,3,4); //这里， A的构造函数由D调用，e.x初始化为4。

- 调用的构造函数及它们的执行次序是：
  A(4)、B(1)、C(2)、D(1,2,3)、E(1,2,3,4)

类作为模块

- 在面向对象程序中，可以把一个类当作一个模块，也可以把多个具有继承关系的类作为一个模块。
- 一个C++模块一般由两个源文件构成，一个是.h文件，另一个是.cpp文件。对于一个由类构成的C++模块，它的.h文件中存放的是类的定义，.cpp文件中存放的是类成员函数的定义（实现）。

```
//modele2.h
class A
 { int I,j;
   void h();
   public:
   void f();
   void h();
 }
```

```
//module2.cpp
#include “module2.h”;
void A::f()
{…i…
 …h()…
}
void A::g()
{……
}
void A::h()
{……
}
```

```
//modele1.cpp
int main()
 { A a;
   a.f();
   a.g();
 }
```

Demeter法则

- 良好的程序设计风格
  - 过程式——结构化程序设计
  - 对象式——？
- 良好的对象程序设计风格的目标是降低模块间的耦合度，增强模块的可复用性和易维护性
- Demeter法则：一个类的成员函数除了能访问自身类结构的直接子结构（表层子结构）外，不能以任何方式依赖于任何其他类的结构；并且每个成员函数只应对应于某个有限类集合中的对象发送消息。
  - 自身类结构的直接子结构：指本类的数据成员
  - 成员对象类的数据成员不包含在内

小结

- 子类型
- 多继承
  - 概念
  - 初始化
  - 两种二义性和解决办法
    - 同名冲突问题：类名受限
    - 重复继承问题：虚基类
- 类作为模块
- Demeter法则