第三课 继承和派生

- 类的继承与派生
- 类成员的访问控制
- 单继承与多继承
- 虚拟继承
- 派生类的构造、析构函数
- 类成员的标识与访问

类的继承与派生

- 保持已有类的特性而构造新类的过程称为继承。
- 在已有类的基础上新增一些特性而产生新类的过程称为派生。
- 被继承的已有类称为基类(或父类)，派生出的新类称为派生类(或子类)。
- 继承的目的：实现代码重用。
- 派生的目的：新问题出现，原有程序无法解决(或不能完全解决)，需要对原有程序改造。

派生类的声明

```
class 派生类名：继承方式 基类名 //多继承则写继承列表
{
  // 新增成员声明;
};
```

继承时，基类一定要有完整定义，只有声明不行：

```
class  employee; // 只有声明，没有定义
class manager : public employee
{
 // 不行，employee没有定义
};
```

继承方式

- 三种继承方式
  - 私有继承 **private** （化公为私）
  - 保护继承 **protected** （折中）
  - 公有继承 **public** （原封不动）

保护级别降低

继承方式影响子类的访问权限：

- 派生类成员对基类成员的访问权限；
- 通过派生类对象对基类成员的访问权限；

类成员的访问权限

public权限表示类的这些成员在类内类外皆可以无限制地访问；

private权限表示类这些成员仅供本类的成员任意访问，对外是不开放的。

protected权限表示类这些成员基本同于私有，但比私有稍扩大了，即将“仅供本类的成员任意访问”，扩大为“派生类的成员也可以访问”。

继承的工作内容

三项工作：

- 吸收基类成员

构造函数、拷贝构造函数和析构函数不能继承

- 改造基类成员

1. 对基类成员访问权限的改变；（规则严格）
2. 对基类成员的覆盖；（有技巧）

- 新增派生类特有的成员

“青出于蓝而胜于蓝”（自由发挥）

公有继承(public)

类成员的访问控制

- 基类的public和protected成员的访问属性在派生类中保持不变，但基类的private成员不可直接访问。
- 派生类中的成员函数可以直接访问基类中的public和protected成员，但不能直接访问基类的private成员。
- 通过派生类的对象只能访问基类的public成员。

（前两条属类内访问，后条属类外访问）

类成员的访问控制

若有基类定义如下：

```
class Base
{
    [redacted]
    protected:    //保护成员
      void Bf2() { }
    public:       //公有成员
      void Bf3() { }
};
```

类成员的访问控制

子类公有继承：

```
class Derived: public Base
{
```

```
    public:          //公有成员
       void Bf3()  { }
       void Df3()  { }
};
```

公有继承举例

类成员的访问控制

```
class Point   //基类Point类的声明
{
public:         //公有函数成员
    void InitP(float xx=0, float yy=0)
     { X=xx;Y=yy; }
    void Move(float xOff, float yOff)
     { X+=xOff;Y+=yOff; }
    float GetX()    {return X;}
    float GetY()    {return Y;}
private:        //私有数据成员
    float X,Y;
};
```

```
class Rectangle: public Point  //派生类声明
{
public:      //新增的公有函数成员
   void InitR(float x, float y, float w, float h)
  { InitP(x,y);W=w;H=h; } //调用基类公有成员函数
   float GetH()    {return H;}
   float GetW()     {return W;}
private:     //新增的私有数据成员
   float W,H;
};
```

```
#include<iostream>
#include<cmath>

int  main()
{
    Rectangle rect;
    rect.InitR(2,3,20,10);
    rect.Move(3,2);
    std::cout<<rect.GetX()<<',‘<<rect.GetY()<<','
        <<rect.GetH()<<','<<rect.GetW()<<‘\n’;
}
```

保护继承(protected)

- 基类的public和protected成员都以protected身份出现在派生类中，但基类的private成员不可直接访问。
- 派生类中的成员函数可以直接访问基类中的public和protected成员，但不能直接访问基类的private成员。
- 通过派生类的对象不能直接访问基类中的任何成员。

（前两条属类内访问，后条属类外访问）

protected 成员的特点

- 对于本类模块来说，它与 **private** 成员的性质相同，只供类内访问。
- 对于其派生类来说，它没变得不可访问，仍然保持了原来的性质。
- 既实现了数据隐藏，又方便了子类实现代码重用。

保护继承举例

```
#include <iostream>
class base
{
  int x;
protected:
    int y;
public:
  base(int xx=10,int yy=20) : x(xx), y(yy) {}
  void show()
  {
     std::cout << “x= “ << x << “, y= “ << y << ‘\n’;
  }
};
```

```
class derived : protected base   //保护派生类
{
  int a;
protected:
    int b;
public:
  derived(int aa=50,int bb=60,int cc=70,int dd=80)
    : base(cc,dd), a(aa), b(bb) {}
  void show()
  {
     base::show();  // 调用基类的public成员
     std::cout<<“ a= “ << a<<“, b= “ << b << ‘\n’;
  }
};
```

```
int main ()
{
   base obj1(100,200);
   obj1.show();
   derived obj2(400,600);
   obj2.show();
}
```

访问权限汇总

在父类中的访问权	继承方式	在子类中的访问权
public	public	public
protected	public	protected
private	public	不能直接访问
public	protected	protected
protected	protected	protected
private	protected	不能直接访问
public	private	private
protected	private	private
private	private	不能直接访问

思考题：

现有如下的基类和子类：

```
#include <string>
class person
 {
  public:
     person();              // 为简化，省略参数
     ~person();
     // ...
  private:
    std::string name, address;
};
```

```
class student : public person
 {
   public:
      student();                    // 为简化，省略参数
      ~student();
      ...
  private:
      string schoolname, schooladdress;
};

student return_student(student s)
{
      return s;
}

student Plato;
student  ss = return_student(Plato); // 多少次拷贝和析构？
```

继承与静态成员

- 类的静态成员不受继承方式的影响，完全可以直接访问。

  访问形式： 基类名:: 静态成员名
- 子类可访问基类的静态成员。
- 类外对静态成员的访问，取决于该成员的访问权限。

派生子类共享静态数据成员

```
#include <iostream>
class Base                          //普通的基类Base
{
  public:
     Base(int a)  : x(a) { }
     void Setx(int a)  {x=a;}
     static void Sety(int a)  { y = a;}     //静态成员函数
     void Show()
     { std::cout << "x = “ << x <<“, y = “ << y << ‘\n’; }
  private:
      int x;
      static int y;                 //静态数据成员
};

int Base::y = 10;
```

```
class Derived : public Base
{
  public:
     Derived(Base a, int b) : Base(b), ob(a)  {  }
     void Show()
     {
          //显示了所继承的数据成员及共享的静态成员
          Base::Show();
          //显示了组合对象的数据成员及共享的静态成员
          ob.Base::Show();
      }
  private:
       Base ob;          //组合了基类的对象
};
```

```
int main()
{
    Base A(99);
    A.Show();
    A.Setx(100);
    A.Sety(200);
    A.Show();

    Derived D(A,1234);
    D.Show();
}
```

运行结果：

x = 99, y = 10

x = 100, y = 200

x = 1234, y = 200

x = 100, y = 200

类型兼容(向上转换)

class Person { ... };

class Student: public Person { ... };

从日常经验中我们知道，每个学生是人，但并非每个人是学生。这正是上面的层次结构所声明的。我们希望，任何对 "人" 成立的事实 ---- 如都有生日 ----也对 "学生" 成立；但我们不希望，任何对 "学生" 成立的事实 ---- 如都在某一学校上学 ----也对 "人" 成立。人的概念比学生的概念更广泛；学生是一种特定类型的人。

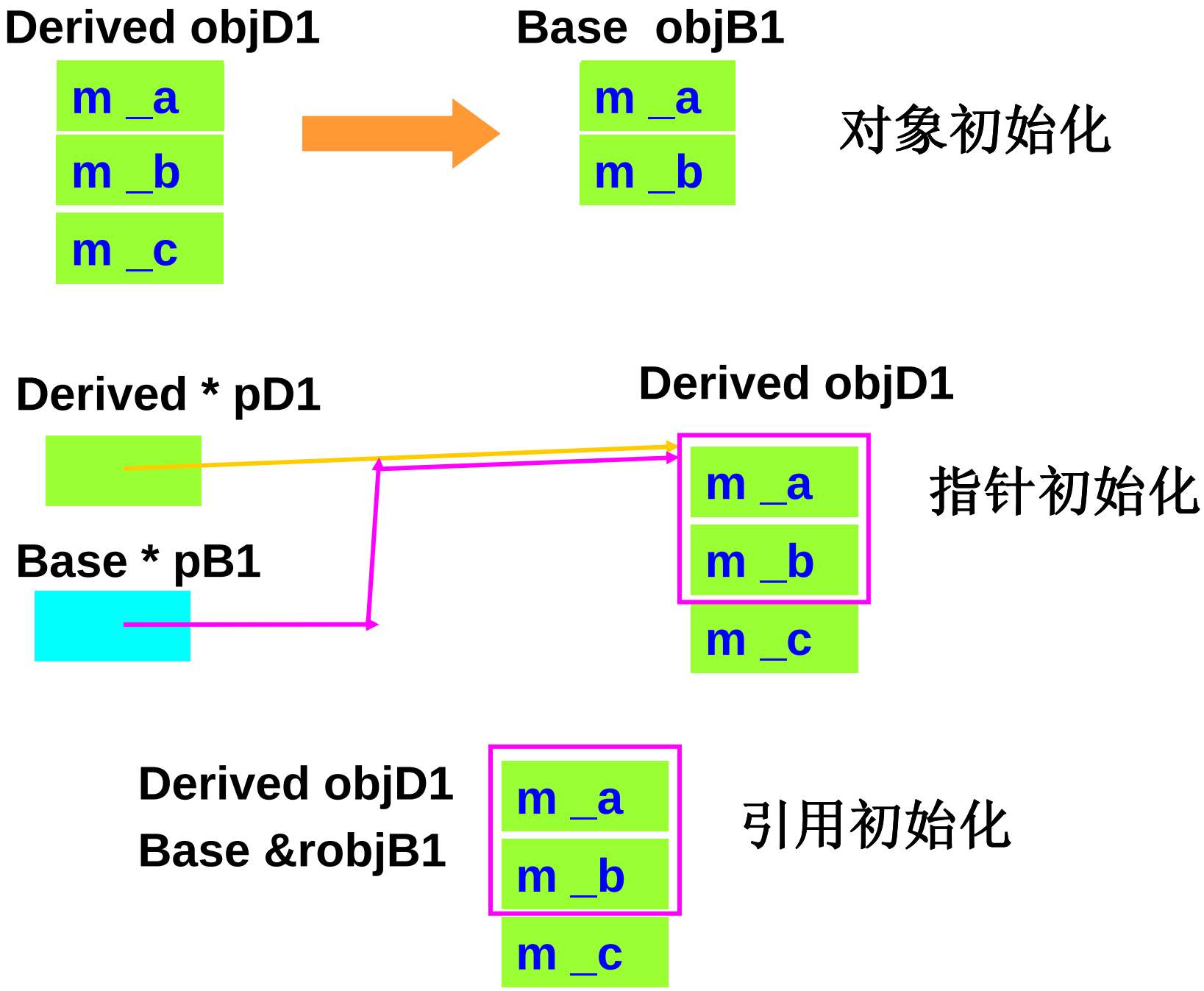
类型兼容规则

- 一个公有派生类的对象可以自动转换成基类的对象(反过来则禁止)
  - 派生类的对象可以被赋值给基类对象。
  - 派生类的对象可以初始化基类的引用。
  - 指向基类的指针也可以指向派生类对象。
- 通过基类对象名、引用名、指针只能使用从基类继承来的成员。
- 此规则又称“向上转换”。
- 思考：C++中为何允许这样自动转换？

```
// 类型兼容的例子：
struct Base
{
  int m_a , m_b ;
  Base(int a = 0, int b = 0) : m_a(a), m_b(b) {}
};

struct Derived : public Base
{
  int m_c;
  Derived(int a = 0, int b = 0, int c = 0) : Base(a, b), m_c(c) {}
};

int main( void )
{
  Derived objD1;
  Base objB1= objD1;       // 发生转换
  Derived *pD1 = &objD1;
  Base *pB1 = pD1;          // 指针类型转换
  Base & robjB1 = objD1; // 引用转换
}
```


Derived objD1
m _a
m _b
m _c
Base objB1
m _a
m _b
对象初始化
Derived * pD1
Base * pB1
Derived objD1
m _a
m _b
m _c
指针初始化
Derived objD1
Base &robjB1
m _a
m _b
m _c
引用初始化

类型兼容规则举例

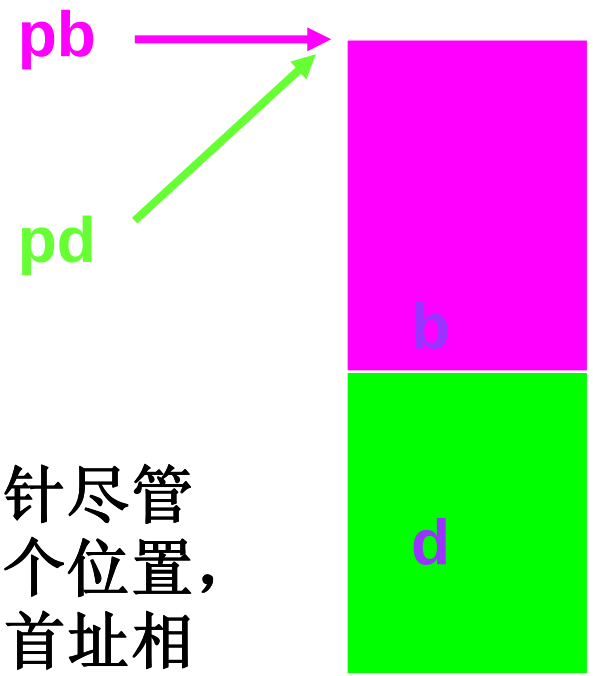
```
#include <iostream>
class B0    //基类B0声明
{
public:
  void display() const
  {
    std::cout<< “B0::display()” << ‘\n’;
  }
};
```

```
class B1: public B0
{
 public:
  void display() const
  { std::cout << "B1::display()"<< '\n'; }
};
class D1: public B1
{
public:
  void display() const
  { std::cout << "D1::display()" << '\n';}
};
void fun(B0 *ptr)
{  ptr->display();     /*"对象指针->成员名"  */  }
```

```
int main()     //主函数
{
   B0 b0;      //声明B0类对象
    B1 b1;     //声明B1类对象
    D1 d1;     //声明D1类对象
    B0 *p;     //声明B0类指针
    p=&b0;     //B0类指针指向B0类对象
    fun(p);
    p=&b1;     //B0类指针指向B1类对象
    fun(p);
    p=&d1;     //B0类指针指向D1类对象
    fun(p);
}
```

单继承时的对象指针

```
class B
{
public:
  // …
private:
     int b;
};
class D :public B
{
public:
// …
private:
  int d;
};
D  objd;
B * pb = &objd;
D * pd = &objd;
```



这两个指针尽管指在同一个位置，那仅仅是首址相同而已，所涵盖的区域并不相同。是类型决定了识别域的差别，或者说访问权限不同。

多继承时派生类的声明

class 派生类名：继承方式1 基类名1，
　继承方式2 基类名2，...

{

　// ... 新增成员声明

};

注意：每一个“继承方式”，只用于控制紧随其后之基类的继承。

默认的继承方式是私有继承。

多继承举例

```
class A
{
public:
   void setA(int);
   void showA() const;
private:
   int a;
};
class B
{
public:
   void setB(int );
   void showB( ) const;
private:
      int b;
};
```

```
class C : public A, private B
{
   public:
      void setC(int, int, int);
      void showC() const;
   private:
      int c;
};
```

```
void  A::setA(int x)
{   a = x;  }

void B::setB(int x)
{   b = x;  }

void C::setC(int x, int y, int z)
{   //派生类成员直接访问基类的公有成员
    setA(x);
    setB(y);
    c = z;
}
//其它函数实现略
```

```
int main()
{
   C obj;
   obj.setA(5);
   obj.showA();
   obj.setC(6,7,9);
   obj.showC();
   obj.setB(6);  // error
   obj.showB(); // error
}
```

多继承时的构造函数

派生类名::派生类名

(基类1形参，基类2形参，... 基类n形参，

本类形参)

: 基类名1(参数), 基类名2(参数), ...基类名n(参数)

{

本类成员赋初值语句;

}

初始化列表

构造函数的调用次序

1. 首先调用基类构造函数，调用顺序按照它们被继承时声明的顺序（从左向右）。
2. 然后调用成员对象的构造函数，调用顺序按照它们在类中声明的顺序。
3. 最后，执行派生类的构造函数体中的语句。

友元关系不能继承

- 我们已经知道：构造函数、拷贝构造函数以及析构函数都不能继承。

- 如果B类是A类的友元， B类的子类不会自动成为A类的友元类。(借来的东西不是遗产。)

```
#include <iostream>
class A
{
//  int j;
protected:
    int i;
    friend class FriendB;
public:
    A() : i(0) {}
};

class FriendB
{
public:
    void show(const A & a) const
    {
      std::cout << “A::i of FriendB is : “ << a.i <<‘\n’;
    }
};
```

```
class FriendD : public FriendB
{
public:
    void show(const A & a) const
    {
      //   std::cout << “A::i  of FriendD is : “
                        <<a.i<<endl;
    }
};

int main()
{
    A a;
    FriendD FD;
    FD.show(a);
    std::cout << std::endl;
}
```

小结：使用初始化列表的场合

- 当类关系是组合时，为作为成员的对象隐式调用构造函数，产生有名对象之用。此时初始化列表为构造函数传递实参。
- 当类关系是继承时，为作为子类组成部分的父类成员显式调用构造函数，产生无名对象之用。此时初始化列表也为构造函数传递实参。
- 也可以为类自身的数据成员赋初值之用。尤其是为常数据成员和引用型数据成员初始化时之用。

继承时的析构函数

- 析构函数不被继承，派生类需自行声明
- 声明方法与一般（无继承关系时）类的析构函数相同。
- 不需要显式地调用基类的析构函数，系统会自动隐式调用。
- 析构函数的调用次序与构造函数相反。

派生类析构函数举例

```
#include <iostream>
class B1  //基类B1声明
{
public:
  B1(int i)   { std::cout << "constructing B1 “ << i << std::endl; }
  ~B1() { std::cout<<"destructing B1 "<< std::endl; }
};
class B2  //基类B2声明
{
public:
     B2(int j) { std::cout << "constructing B2 “ << j << std::endl; }
     ~B2() { std::cout<<"destructing B2 "<< std::endl;}
};
class B3  //基类B3声明
{
public:
     B3() { std::cout << "constructing B3 *“ << std::endl; }
     ~B3() { std::cout << "destructing B3 “ << std::endl;}
};
```

```
class C: public B2, public B1, public B3
{
public:
    C(int a, int b, int c, int d) :
    B1(a), memberB2(d),memberB1(c),B2(b)
    {  std::cout << “constructing C ”<< std::endl;   }
    ~C() { std::cout<<“destructing C ”<< std::endl;   }
private:
    B1 memberB1;
    B2 memberB2;
    B3 memberB3;
};
int main()
{
   C obj(1, 2, 3, 4);
}
```

尽管没显式给出析构函数的调用关系，但并不影响其连锁调用机制。

此处调用了几次析构？

同名成员函数隐藏

- 当派生类与基类有同名成员函数时,派生类中的成员函数将屏蔽基类中的同名成员函数，即使成员函数参数不同也不能重载，习惯称之为“覆盖”或“隐藏”。
- 若未特别指明，则通过派生类对象使用的都是派生类中的同名成员函数;
- 如要通过派生类对象访问基类中被屏蔽的同名成员，应使用基类名限定(::)，如果想使成员函数重载，可以使用using来做到。

同名覆盖举例

```
#include <string>
class A//声明基类B1
{
  std::string m_a;
public:
  A(const std::string a = "A") : m_a(a) {}
  void reset(const std::string& a) { m_a = a; }
};

class B : public A
{
  double m_b;
public:
  B(const std::string& a = "", double b = 0.0) : A(a), m_b(b) {}
  explicit B(const B& b) : A(b), m_b(b.m_b) {}
  using A::reset;                          // 重载
  void reset(double b) { m_b = b; }
};
```

```
class C : public B
{
public:
  C(const std::string& a = "", double b = 0.0) : B(a, b) {}
  void reset(const std::string& str, double d)
  {
      B::reset(str);                          // ok, A::reset
      B::reset(d);
  }
};

int main()
{
      B b;
      b.reset("new A");                       // ok, A::reset

      C c;
      c.B::reset("new A from C");             // ok, A::reset
      c.A::reset("new A from C again");       // ok, A::reset
      c.reset("new A from C again");          // error
}
```

虚拟继承

- 虚拟继承的引入
  - 用于有共同基类的多继承场合(多层共祖)
- 声明
  - 以virtual修饰说明共同的直接基类
    例：**class B1 : virtual public B**
- 作用
  - 用来解决多继承时可能发生的对同一基类继承多次和多层而产生的二义性问题.
  - 为最远的派生类提供唯一的基类成员，而不重复产生个副本。
- 注意：
  - 在第一级继承时就要将共同基类设计为虚基类。

虚拟继承举例

```
class B  { public: int b; };
class B1 : virtual public B { private: int b1;};
class B2 : virtual public B { private: int b2;};
class C: public B1, public B2{ private: float d;};
```

在子类对象中，继承的基类B数据成员是唯一的。

于是下面的访问是正确的：

```
C  cobj;
cobj.b;
```

有虚拟继承时构造函数的调用次序

- 无论虚拟继承与产生对象的派生类相隔多远，首先调用虚拟继承基类的构造函数；
- 然后按继承次序调用直接基类的构造函数；
- 如果有包含的对象，再按声明次序调用所包含对象类的构造函数；
- 最后才是普通类型数据成员的初始化。

选择继承的理由

要表现两事物共同的属性或行为，不是定义两个孤立的类，让它们都含有相同的部分，那肯定不是**oop**。

要使用继承。但如何继承？那要看事物原本的逻辑关系。比如事物是人和学生，则要让人作为基类，学生是子类。这是简单的继承。

若事物是同一个公司的技术员和工人，就不是简单的继承了。而要让他们都去继承公司中并不存在的雇员类。

若事物是某公司的销售经理，而同一个公司还有销售员和经理，于是可以用多继承。

采用组合(包含)还是继承？

- 继承和包含都是重要的重用方法
- 在OO开发的早期，继承被过度使用
- C++的“继承”特性可以提高程序的可复用性。正因为“继承”太有用、太容易用，才要防止滥用“继承”。我们应当给“继承”一些使用规则。

- 规则1：如果类A 和类B 毫不相关，不能为了使B 的功能更多些而让B继承A 的功能和属性。
- 规则2：若在逻辑上B 是A 的“一种”，则允许B 继承A 的功能和属性。

采用包含还是继承？

- 例如：男人（Man）是人（Human）的一种，男孩（Boy）是男人的一种。 那么类Man 可以从类Human 派生，类Boy 可以从类Man 派生。

```
class Human
 {
    …
 };

class Man : public Human
 {
    …
 };

class Boy : public Man
 {
    …
 };
```

采用包含还是继承？

- 看起来很简单，但是实际应用时可能会有意外，继承的概念在程序世界与现实世界并不完全相同。

- 例如从生物学角度讲，鸵鸟（Ostrich）是鸟（Bird）的一种，按理说类Ostrich 应该可以从类Bird 派生。但是鸵鸟不能飞，那么Ostrich::Fly 怎么理解？

- ```
  class Bird
  {   public:
          virtual void Fly(void);
      …
  };
  class Ostrich : public Bird
  {
      …
  };
  ```
  ```

采用组合(包含)还是继承？

- 又如：从数学角度讲，圆（Circle）是一种特殊的椭圆（Ellipse），按理说类Circle 应该可以从类Ellipse 派生。但是椭圆有长轴和短轴，如果圆继承了椭圆的长轴和短轴，是不是画蛇添足？

- 所以更加严格的继承规则应当是：若在逻辑上B 是A 的“一种”，并且A 的所有功能和属性对B 而言都有意义，则允许B 继承A 的功能和属性。

- 若在逻辑上A 是B 的“一部分”，则不允许B 从A 派生，而是要用A 和其它东西组合出B。

采用包含还是继承？

- 例如眼（Eye）、鼻（Nose）、口（Mouth）、耳（Ear）是头（Head）的一部分，所以类Head 应该由类Eye、Nose、Mouth、Ear 组合而成，不是派生而成。
- 如果允许Head 从Eye、Nose、Mouth、Ear 派生而成，那么Head 将自动具有Look、Smell、Eat、Listen 这些功能。下面程序十分简短并且运行正确，但是这种设计方法却是不对的。

```
class Head : public Eye, public Nose, public Mouth, public Ear
{
    ……
};
```

编程练习

- **1.使用下面的理论选择某一组编程设计一个32位的随机数产生器(用C++的类来做)，但允许以使用C库函数rand()的格式使用(包装成C API的形式)。**

下面是设计这个随机数产生器的理论基础。

古老的LCG(linear congruential generator)代表了最好的伪随机数产生器算法。尽管随机性不太理想，但是容易理解，容易实现，而且速度快。这种算法数学上基于：

X(n+1) = (a * X(n) + c) % m 这样的公式，其中：

模m, m > 0
系数a, 0 < a < m
增量c, 0 <= c < m
原始值(种子) 0 <= X(0) < m
其中参数c, m, a比较敏感，或者说直接影响了伪随机数产生的质量。
一般而言，高LCG的m是2的指数次幂(一般2^32或者2^64)，因为这样取模操作截断最右的32或64位就可以了。多数编译器的库中使用了该理论实现其伪随机数发生器rand()。

下面是部分编译器使用的各个参数值：

Source	m	a	c
Numerical Recipes	2^32	1664525	1013904223
Borland C/C++	2^32	22695477	1
glibc (used by GCC)	2^32	1103515245	12345
ANSI C: Watcom, Digital Mars, CodeWarrior, IBM VisualAge C/C++	2^32	1103515245	12345
Borland Delphi, Virtual Pascal	2^32	134775813	1
Microsoft Visual/Quick C/C++	2^32	214013	2531011
Apple CarbonLib	2^31-1	16807	0

为了降低编程难度，下面给出一个例子，你可以参考这个例子来做。

```
#include <ctime>
// the code below only works on the environment of C++ compiler compatible to C++ 0x well,
// such as GNU g++4.5.x, g++4.6.x, Microsoft Visual C++ 2010, Intel C++ 11.1 ... or latest

class random
{
private:
  typedef uint32_t UInt32; // 如果你的编译器不认识uint32_t，就用unsigned long代替
  UInt32 a;
  UInt32 b;
  UInt32 c;
  UInt32 d;

  UInt32 randomize()
  {
    UInt32 e = a - ((b << 27) | (b >> 5));
    a = b ^ ((c << 17) | (c >> 15));
    b = c + d;
    c = d + e;
    d = e + a;
    return d;
  }

 void init()
  {
     for(UInt32 i = 0; i < 20; ++i)
        randomize();
  }

public:
```

```
 explicit random(UInt32 s = std::time(0)) : a(4058668781ul), b(s), c(s), d(s)
 {
    init();
 }

 void reset(UInt32 s = 0)
 {
   if(0 == s)
     s = std::time(0);
   a = 4058668781ul;
   b = c = d = s;
   init();
 }

 UInt32 rand()
 {
 //returns a random integer in the range [0, 4294967296)
   return randomize();
 }

 double real()
 {
 //returns a random real number in the range [0.0, 1.0)
   return randomize() / 4294967296.0;
 }

 UInt32 operator()() { return rand(); }
};
```

```
class smrand_wrapper
{
  static random r;
public:
  smrand_wrapper() {}
  void operator()(uint32_t s) { r.reset(s); }
  uint32_t operator()() const { return r.rand(); }
  double operator()(double) { return r.real(); }
};

random smrand_wrapper:: r;

void reset(uint32_t ns) { smrand_wrapper()(ns); }       // reset seed
inline uint32_t rand32() { return smrand_wrapper()(); }  // for 32 bit integers
inline uint16_t rand16() { return rand32() >> 16; }      // for 16 bit integers
inline double rand53() { return smrand_wrapper()(0.0); } // for real numbers

// 经过上面的设计，我们就可以直接这样使用它了，省掉了使用前首先初始化的麻烦(想想使用C库函数rand前
      首先需要用srand初始化就明白了)，假设在32位系统上。
#include <iostream>
int main()
{
  for(unsigned i = 0; i < 10; ++i)
    std::cout << rand32();  // 显式10个32位的随机正整数，范围[0, 4294967296)
  std::cout << rand53(); // 显示一个随机浮点数，范围在[0.0, 1.0)
}
```

- **2. 使用继承来设计两个类，雇员和经理，数据成员自拟，成员函数自拟。设计和编码完毕后，测试一下，通过输出一点必要信息的形式来了解构造函数、拷贝构造函数以及析构函数的级联调用关系和调用时机。**

- **3. 调试运行一下本课件中的思考题，想想为什么调用那么多次拷贝和析构。**

Thanks!