第2章 继承性和派生性

教学要求、重点与难点

- 教学要求
  - 1、能通过继承现有的类建立新类；
  - 2、了解基类和派生类的概念
  - 3、能够用多重继承从多个基类派生出新类。

- 重点与难点
  - 用多重继承从多个基类派生出新类。

内　　容

1　基类和派生类

2　单继承

3　多继承

4　虚基类

5　应用实例——日期和时间

本章小结

本章作业

继承性

封装、继承和多态性是面向对象程序设计的三大机制，其中，继承性是面向对象程序设计中最重要的机制。

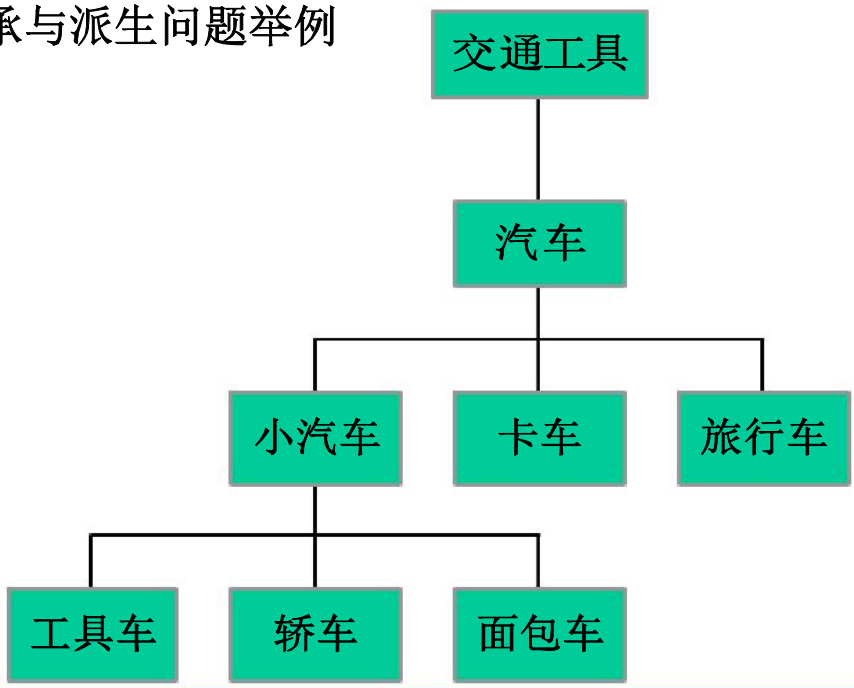
继承机制提供了无限重复利用程序资源的一种途径。利用继承机制，可扩充和完善就的程序设计一适应新的需求，这样既节省程序开发的时间和资源，又便于维护。

§2.1 基类和派生类

- 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自己的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(子类）

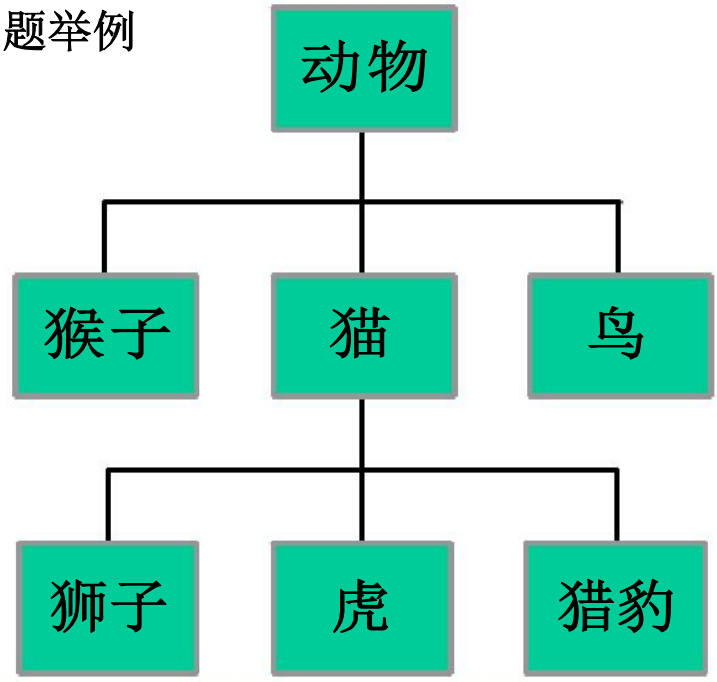
§2.1 基类和派生类

- 继承与派生问题举例



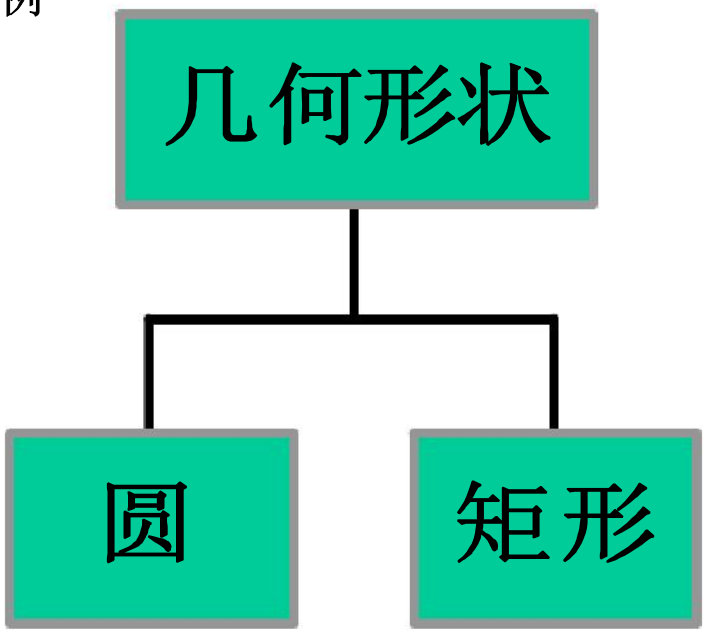
§2.1 基类和派生类

- 继承与派生问题举例



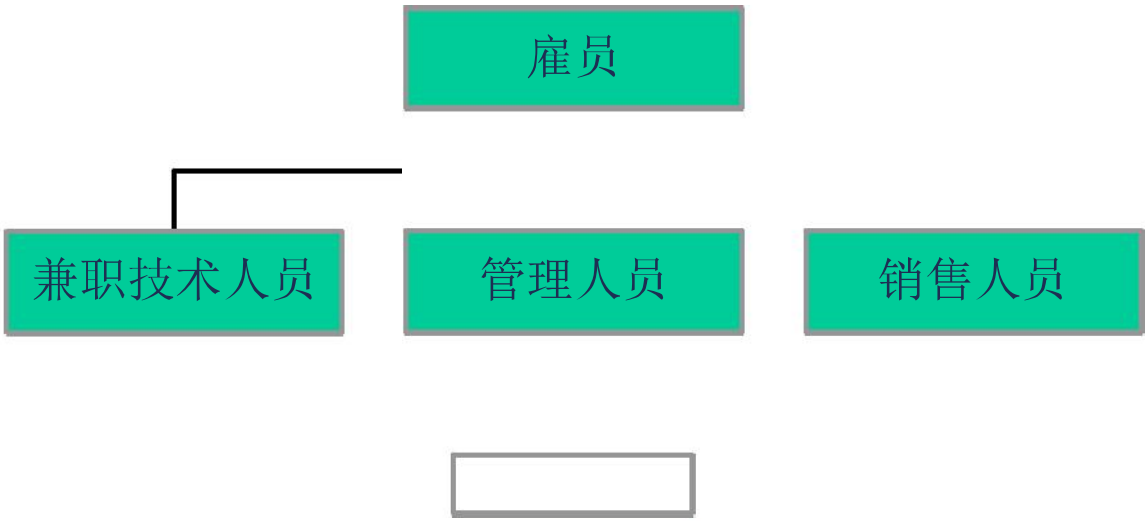
§2.1 基类和派生类

- 继承与派生问题举例

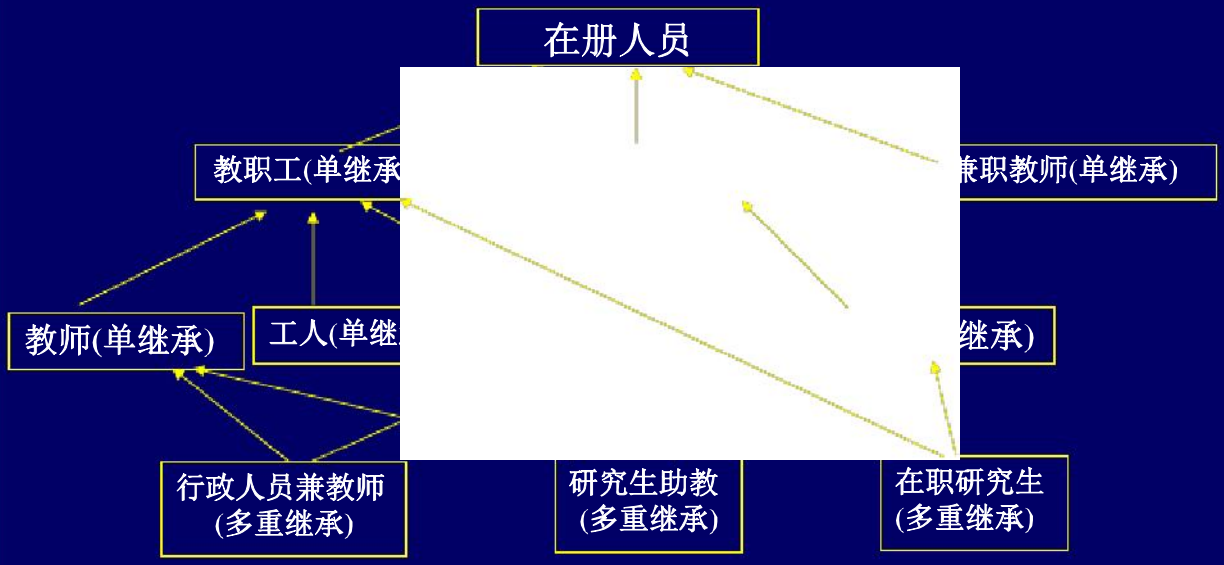


§2.1 基类和派生类

- 继承与派生问题举例



继承与派生问题举例


在册人员
教职工(单继承
兼职教师(单继承)
教师(单继承)
工人(单继
继承)
行政人员兼教师
(多重继承)
研究生助教
(多重继承)
在职研究生
(多重继承)

§2.1 基类和派生类

- 继承与派生的目的
- 继承的目的：实现代码重用。
- 派生的目的：当新的问题出现，原有程序无法解决（或不能完全解决）时，需要对原有程序进行改造。

编制派生类时可分四步

- 吸收基类的成员
- 改造基类成员
- 发展新成员
- 重写构造函数与析构函数

§2.1 基类和派生类

回顾：类的定义格式

```
class 类名称
  {
      public:
          公有成员（外部接口）
      private:
          私有成员
      protected:
          保护型成员
  }
```

本类内的所以成员在本类内均可访问，但该类的对象只可访问公有成员

§2.1 基类和派生类

一、派生类的定义格式

1、单继承的定义格式：

```
class <派生类名>：<继承方式> <基类名>
{
    <派生类新定义成员>
};
```

其中，<继承方式> 为public，private和protected

2、多继承的定义格式：

```
class <派生类名>：<继承方式1> <基类名1>，<继承方式2> <基类名2>，…
{
    <派生类新定义成员>
};
```

§2.1 基类和派生类

二、派生类的三种继承方式

- 不同继承方式的影响主要体现在：
  1、派生类成员对基类成员的访问控制。
  2、派生类对象对基类成员的访问控制。
- 三种继承方式
  - 公有继承public
  - 私有继承private
  - 保护继承protected

§2.1 基类和派生类

二、派生类的三种继承方式

1、公有继承(public)

- 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。

例1 公有继承举例

```
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;
};
```

例1 公有继承举例

```
class Rectangle: public Point  //派生类声明
{
public://新增公有函数成员
    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:          //新增私有数据成员
    float W,H;
};
```

例1 公有继承举例

```
#include<iostream>
#include<cmath>
using namespace std;
int main（ ）
{  Rectangle rect;
    rect.InitR(2,3,20,10);
    //通过派生类对象访问基类公有成员
    rect.Move(3,2);
    cout<<rect.GetX（ ）<<',' <<rect.GetY（ ）<<','
        <<rect.GetH（ ）<<',' <<rect.GetW（ ）<<endl;
    return 0;
}
```

§2.1 基类和派生类

二、派生类的三种继承方式

2、私有继承(private)

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

例2 私有继承举例

```
class Rectangle: private Point      //派生类声明
{public:          //新增外部接口
    void InitR(float x, float y, float w, float h)
    {InitP(x,y);W=w;H=h;}     //访问基类公有成员
    void Move(float xOff, float yOff)
    {Point::Move(xOff,yOff);}
    float GetX（ ） {return Point::GetX（ ）;}
    float GetY（ ） {return Point::GetY（ ）;}
    float GetH（ ） {return H;}
    float GetW（ ） {return W;}
private:          //新增私有数据
    float W,H;
};
```

```
#include<iostream>
#include<cmath>
 using namespace std;
int main（ ）
{  //通过派生类对象只能访问本类成员
    Rectangle rect;
    rect.InitR(2,3,20,10);
    rect.Move(3,2);
    cout<<rect.GetX（ ）<<','
       <<rect.GetY（ ）<<','
         <<rect.GetH（ ）<<','
         <<rect.GetW（ ）<<endl;
    return 0;
}
```

§2.1 基类和派生类

二、派生类的三种继承方式

3、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，但不能访问基类的private成员。
- 通过派生类的对象不能访问基类中的任何成员

§2.1 基类和派生类

二、派生类的三种继承方式

protected 成员的特点与作用

- 对建立其所在类对象的模块来说（水平访问时），它与 private 成员的性质相同。
- 对于其派生类来说（垂直访问时），它与 public 成员的性质相同。
- 既实现了数据隐藏，又方便继承，实现代码重用。

例3 protected 成员举例

```
class A {
    protected:
                int x;
}
int main（ ）
{
     A a;
     a.x=5;    //错误
}
```

```
class A {
    protected:
                int x;
}
class B: public A{
    public:
        void Function（ ）;
};
void B:Function（ ）
{
     x=5;   //正确
}
```

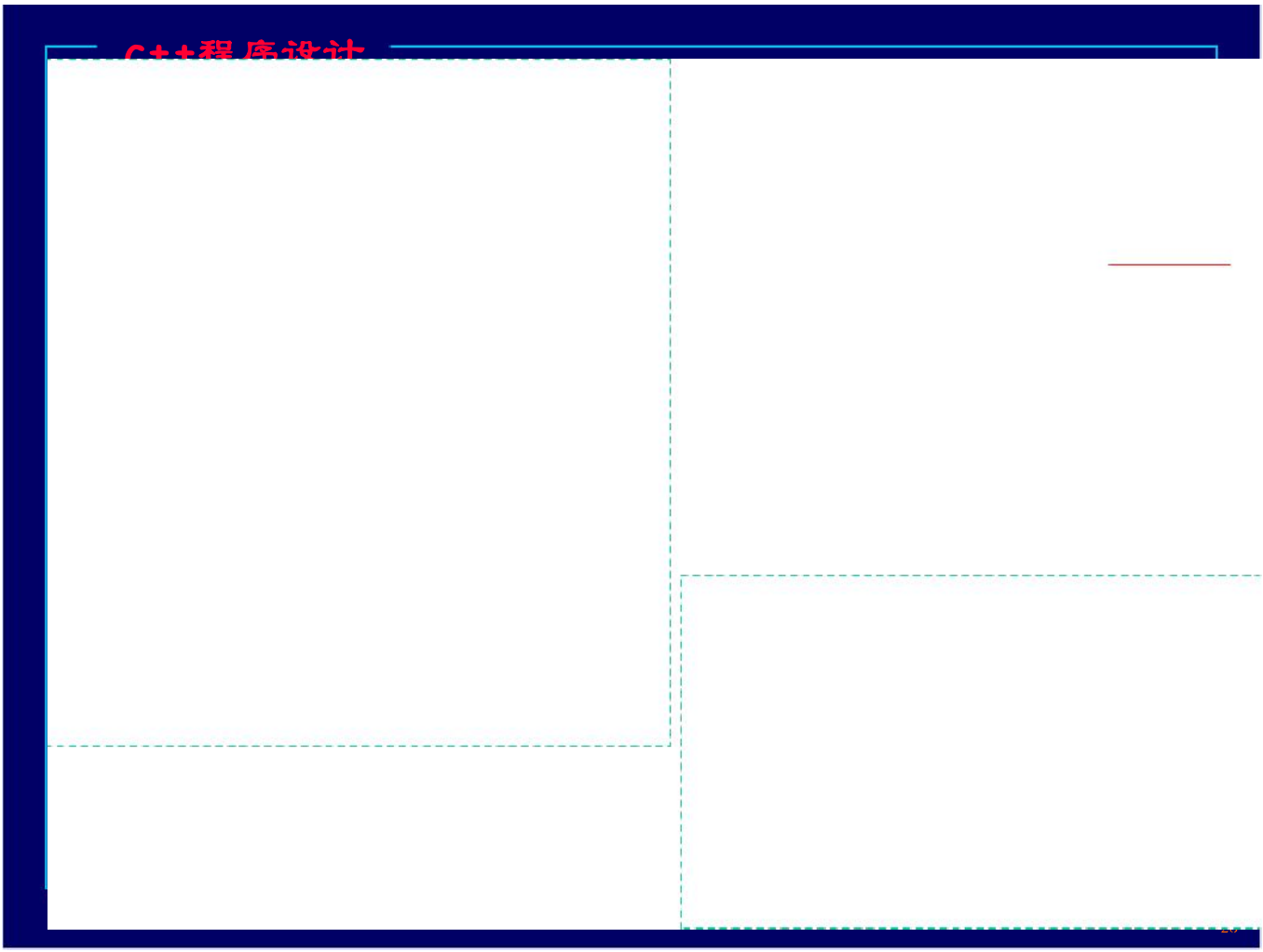
例3 protected 成员举例

```
class A {
    private:
                int x;
}
class B: public A{
    public:
        void Function（ ）;
};
void B:Function（ ）
{
     x=5;   //错误
}
```

§2.1 基类和派生类

三、基类与派生类的对应关系

- 单继承
  - 派生类只从一个基类派生。
- 多继承
  - 派生类从多个基类派生。
- 多重派生
  - 由一个基类派生出多个不同的派生类。
- 多层派生
  - 派生类又作为基类，继续派生新的类。


C++程序设计

继承方式	基类特性	派生类特性
public	public	public
	protected	protected
	private	不可访问
private	public	private
	protected	private
	private	不可访问
protected	public	protected
	protected	protected
	private	不可访问

§2.2 单继承

一、成员访问权限的控制

缺省继承方式是：private

§2.2 单继承

二、构造函数和析构函数

1、 构造函数

- 基类的构造函数不被继承，需要在派生类中自行定义。
- 定义构造函数时，只需要对本类中新增成员进行初始化，对继承来的基类成员的初始化由基类完成。
- 若派生类中存在基类对象（即子对象），则在派生类构造函数中应包含对子对象的初始化。
- 定义格式：

派生类名::派生类名(基类所需的形参，子对象所需的形参，本类成员所需的形参):基类名(参数表1)，子对象名(参数表2)

{　　派生类中数据成员初始化　}

§2.2 单继承

二、构造函数和析构函数

1、 构造函数

- 派生类构造函数的调用顺序：
  - 基类的构造函数
  - 子对象类的构造函数（如果有的话）
  - 派生类构造函数

单继承构造函数例题

```
#include<iostream>
using namespace std;
class A
{
    public:
        A() { a=0;cout<<"A's default constructor called.\n"; }
        A(int i) { a=i;cout<<"A's constructor called.\n"; }
        ~A() { cout<<"A's destructor called."<<endl; }
        void Print() const{cout<<a<<",";}
        int Geta( ) { return a; }
    private:
        int a;
};
```

单继承构造函数例题

```
class B:public A
{
    public:
        B() { b=0; cout<<"B's default constructor called.\n"; }
        B(int i,int j,int k)
        ~B() { cout<<"B's destructor called."<<endl; }
        void Print();
    private:
        int b;
        A  aa;
};
```

单继承构造函数例题

```
B(int i,int j,int k) :A(i),aa(j)
{    b=k;
     cout<<"B's constructor called.\n";
}

void B::Print() const
{
     A:: Print() ;
     cout<<b<<", "<<aa.Geta()<<endl;
}
```

```
void main()
{
    B bb[2];
    bb[1]=B(3,4,8);
    for(int i=0; i<2; i++)
        bb[i].print();
}
```

运行结果：

A's default constructor called.
A's default constructor called.
B's default constructor called.
A's default constructor called.
A's default constructor called.
B's default constructor called.

A's constructor called.
A's constructor called.
B's constructor called.
B's deconstructor called.
A's deconstructor called.
A's deconstructor called.

A's constructor called.
A's constructor called.
B's constructor called.
B's deconstructor called.
A's deconstructor called.
A's deconstructor called.

1,5,2
3,8,4
B's deconstructor called.
A's deconstructor called.
A's deconstructor called.
B's deconstructor called.
A's deconstructor called.
A's deconstructor called.

§2.2 单继承

二、构造函数和析构函数

2、 析构函数

- 析构函数也不被继承，派生类自行定义。
- 定义方法与一般（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。

§2.2 单继承

二、构造函数和析构函数

3、派生类构造函数使用时应注意的问题

- 当基类中定义有缺省形式的构造函数或未定义构造函数时，派生类构造函数的定义中可以省略对基类构造函数的调用。
- 若基类中未定义构造函数，派生类中也可以不定义，全采用缺省形式构造函数。
- 当基类定义有带形参的构造函数时，派生类也应定义带形参的构造函数，提供将参数传递给基类构造函数的途径。

例题参见P_{224}[例6]（请自己分析）

§2.2 单继承

三、子类型化和类型适应

1、 子类型化

- 子类型概念涉及到行为共享，与继承有着密切关系。
- 有一个特定的类型S，当且仅当它至少提供了类型T的行为，则称类型S是类型T的子类型。
- 子类型是类型之间的一般和特殊的关系。
- 在继承中，公有继承可以实现子类型。例如，

§2.2 单继承

```
class A
{
    public:
        void Print() const{cout<<"A::print( ) called.\n";}
};

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

类B继承了类A，并且是公有继承方式。因此说类B是类A的一个子类。类B具有类A中的操作。

§2.2 单继承

```
void f1(const A& r)
 {
      r. Print() ;
}
void main()
{
    B b;
    f1(b);
}
```

运行结果：

A::print( ) called.

说明：子类型关系不可逆；公有继承可以实现子类型化。

§2.2 单继承

三、子类型化和类型适应

2、类型适应

- 类型适应是指两种类型之间的关系。例如，B类型适应A类型是指B类型的对象的对象能够用于A类型的对象所能使用的场合。
- 若派生类对象可以用于基类对象所能使用的场合，则说派生类适应于基类。
- 子类型化与类型适应是一致的。
- 利用子类型可减轻程序员工作量。

§2.2 单继承

三、子类型化和类型适应

- 赋值兼容原则
  - 在需要基类对象的任何地方都可以使用公有派生类的对象来替代
    - 一个公有派生类的对象在使用上可以被当作基类的对象，反之则禁止。具体表现在：
      - 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。
      - 派生类的地址可以赋值给指向基类的指针，即指向基类的指针也可以指向派生类。

§2.2 单继承

三、子类型化和类型适应

- 赋值兼容说明
- 使用同样的函数对一个继承系中的各种对象进行操作——多态性的基础
- 形参为基类对象，实参为派生类对象
- 举例：

```
#include<iostream>  using namespace std;
class B0
{
    public:
      void display()
      {cout<< “B0::display()” <<endl;}
};
```

```
class B1:public B0
{
    public:
        void display()
        {cout<<“B1::display()”<<endl;}
};
class D1:public B1
{
    public:
          void display()
         {cout<<“D1::display()”<<endl;}
};
```

```
void  fun(B0 * ptr)
{
    ptr->display();
}
void main()
{
   B0  b0; B1  b1; D1  d1; B0 *p;
  p=&b0;  fun(p);
  p=&b1;  fun(p);
  p=&d1;  fun(p);
}
```

§2.3 多继承

一、多继承的概念

- 所谓多继承是指派生类具有多个基类，派生类与每个基类之间的关系忍可看作是一个单继承。
- 定义格式：

```
class <派生类名>：<继承方式1> <基类名1>，<继承方式2> <基类名2>，…
{
    成员定义;
};
```

例如，

```
class A
{
    …
};
```

```
class B
{
    …
};
```

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

§2.3 多继承

二、多继承的构造函数

<派生类名>(<总参数表>):<基类名1>(<参数1>), <基类名2>(<参数2>), …,
<子对象名1>(<参数表n+1>)

{

<派生类构造函数体>

};

构造函数的调用次序

1. 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. 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. 派生类的构造函数体中的内容。

多继承派生类构造函数例题

```
#include <iostream.h>
class B1
{
   public:
        B1(int i)
          {    b1=i;
             cout<< “constructor B1."<< i <<endl;
          }
        void Print ( )  {cout<<b1<<endl;}
   private:
       int  b1;
};
```

多继承派生类构造函数例题

```
class B2
{
    public:
          B2(int i)
            {    b2=i;
                cout<<“constructor B2."<< i <<endl;
            }
          void Print ( )  {cout<<b2<<endl;}
    private:
         int  b2;
};
```

多继承派生类构造函数例题

```
class B3
{
    public:
          B3(int i)
            {    b3=i;
                cout<<“constructor B3."<< i <<endl;
            }
         int getb3 ( )  {return b3;}
    private:
        int  b3;
};
```

```
class A: public B2, public B1 //注意基类名的顺序
{
    public:
     A(int i,int j, int k,int l,int m) : B1(i), B2(j), bb(k),ba(m)
      {       a=l;
              cout<< “constructor A."<< l <<endl;
     }
     void Print ( )
      {      B1::Print ( );              B2::Print ( );
             cout<<a<< ","<< bb.getb3( )<<endl;
      }
    private：
       int  a;
       B1   ba; //注意对象定义顺序
       B3 bb;
};
```

```
void main()
{
    A  aa(1,2,3,4,5);
    aa::Print ( );
}
```

运行结果：

```
constructor B2.2
constructor B1.1
constructor B1.5
constructor B3.3
constructor A.4
1
2
4，3
```

§2.3 多继承

三、二义性问题

- 在多继承时，基类与派生类之间，或基类之间出现同名成员时，将出现访问时的二义性（不确定性）——采用虚函数（第9章）或支配（同名覆盖）原则来解决。
- 当派生类从多个基类派生，而这些基类又从同一个基类派生，则在访问此共同基类中的成员时，将产生二义性——采用虚基类来解决。

二义性问题举例

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

```
class C: public A, piblic B
{       public:
        void g();
        void h();
};
```

如果定义：C c1;
则 c1.f(); 具有二义性
而 c1.g(); 无二义性（同名覆盖）

二义性的解决方法

- 解决方法一：用类名来限定
  c1.A::f()　或　c1.B::f()
- 解决方法二：同名覆盖
  在C 中定义一个同名成员函数f()，f()再根据需要调用 A::f()　或　B::f()

同名覆盖原则

当派生类与基类中有相同成员时：

- 若未强行指名，则通过派生类对象使用的是派生类中的同名成员。
- 如要通过派生类对象访问基类中被覆盖的同名成员，应使用基类名限定。

二义性问题举例

```
class B
{       public:
          int b;
};
class B1 : public B
{
      private:
          int b1;
};
class B2 : public B
{
      private:
          int b2;
};
```

```
class C : public B1,public B2
{
      public:
          int f();
      private:
          int d;
};
```

下面的访问是二义性的：

C c;

c.b

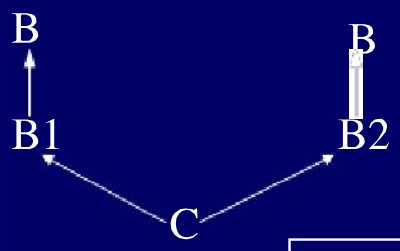
c.B::b

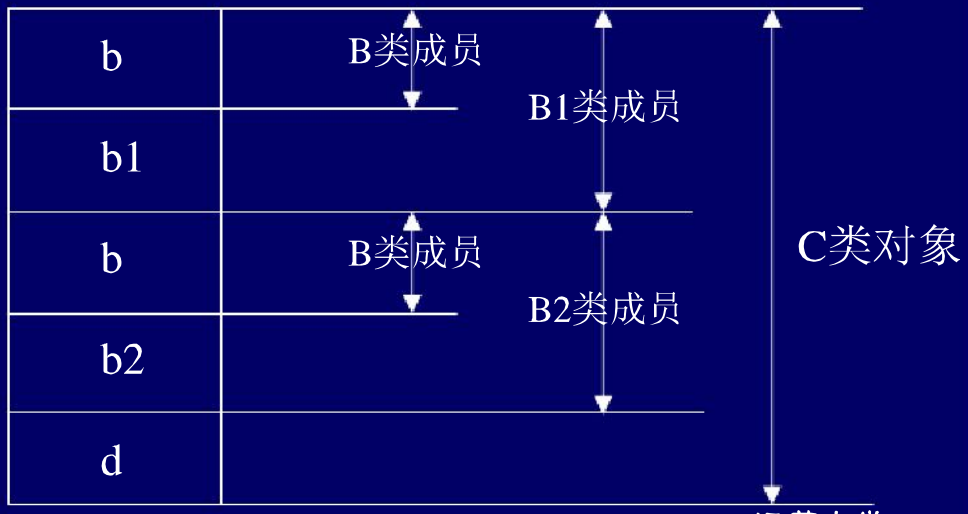
下面是正确的：

c.B1::b

c.B2::b

派生类C的对象的存储结构示意图





§2.4 虚基类

一、虚基类的引入和说明

- 虚基类的引入
  - 用于有共同基类的场合，解决二义性问题
- 定义
  - 虚基类说明格式： virtual <继承方式> <基类名>
    例：class B1:virtual public B
- 作用
  - 主要用来解决多继承时可能发生的对同一基类继承多次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，而不重复产生多次拷贝
- 注意：
  - 在第一级继承时就要将共同基类设计为虚基类。

虚基类举例

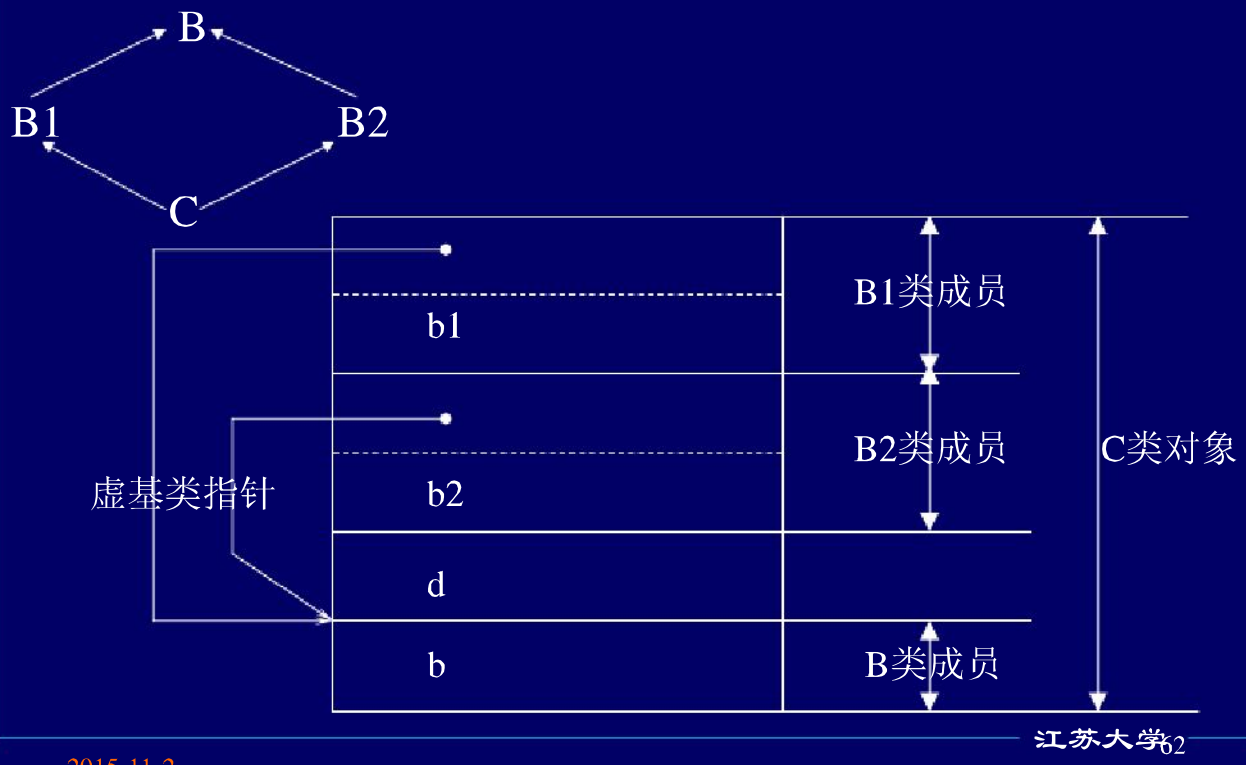
```
class B{public: int b;};
class B1 : virtual public B { private: int b1;};
class B2 : virtual public B { private: int b2;};
class C : public B1, public B2{ private: float d;}
```

下面的访问是正确的：

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

- 引入虚基本类后，派生类(即子类)的对象中只存在一个虚基类的子对象。当一个类有虚基类时，编译系统将为该类的对象定义一个指针成员，让它指向虚基类的子对象。该指针称为虚基类指针。

虚基类的派生类对象存储结构示意图



§2.4 虚基类

二、虚基类的构造函数

- 建立对象时所指定的类称为最（远）派生类。
- 虚基类的成员是由最派生类的构造函数通过调用虚基类的构造函数进行初始化的。
- 在整个继承结构中，直接或间接继承虚基类的所有派生类，都必须在构造函数的成员初始化表中给出对虚基类的构造函数的调用。如果未列出，则表示调用该虚基类的缺省构造函数。
- 在建立对象时，只有最派生类的构造函数调用虚基类的构造函数，该派生类的其它基类对虚基类构造函数的调用被忽略。

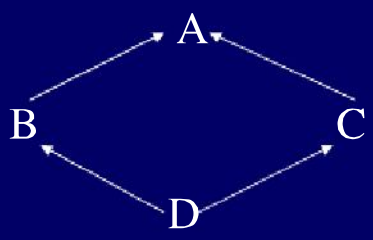
```
#include <iostream.h>// P238[例8.10]
class A          //声明基类A
{ public:        //外部接口
    A(const char *s){cout<<s<<endl;}
     ~A() { }
};
class B: virtual public A
{
 public:
    B(const char *s1, const char *s2):A(s1)
     {
         cout<<s2<<endl;
      }
};
```

```
class C: virtual public A
{
 public:
    C(const char *s1, const char *s2):A(s1)
    {
        cout<<s2<<endl;
    }
};
class D: public B, public C
{
 public:
    B(const char *s1, const char *s2, const char *s3, const char *s4):
    B(s1,s2), C(s1,s3), A(s1)
    {
        cout<<s4<<endl;
    }
};
```

```
void main（ ）
{
    D *ptr=new D("Class A","Class B", "Class C","Class D");
    delete ptr;
}
```

执行结果：

Class A
Class B
Class C
Class D

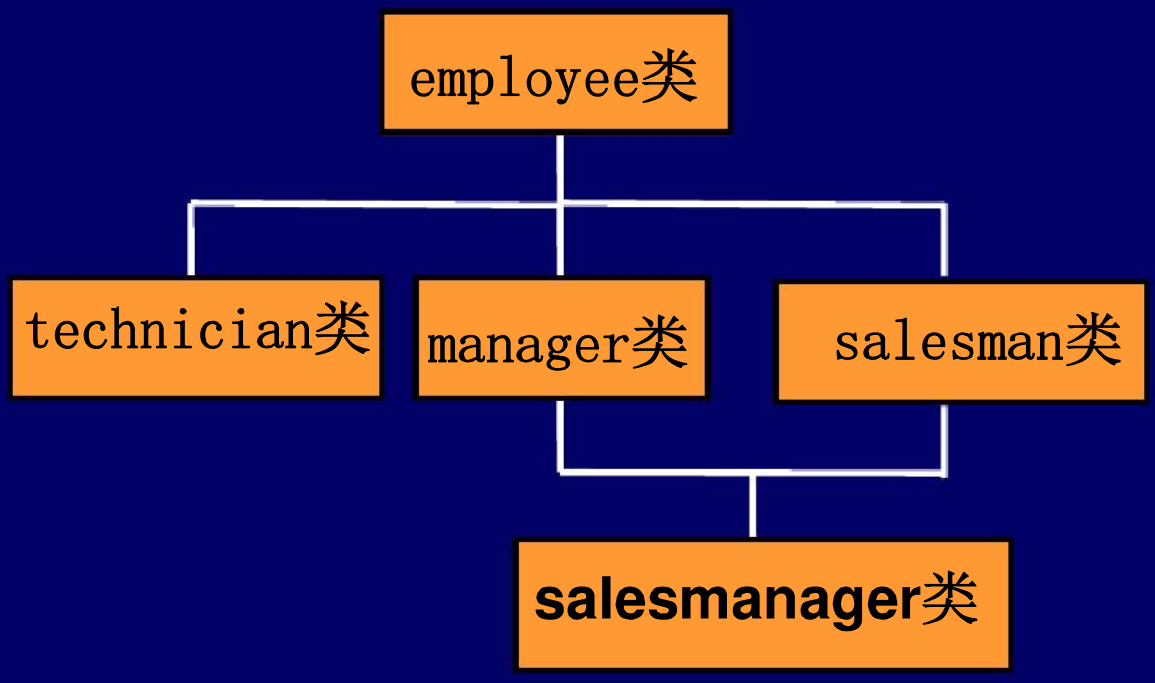


在面向对象语言中实现多继承是复杂的，建议能够使用单继承的时候，避免使用多继承。商品化的程序代码是经过安全测试的，可以学习使用。

综合举例

- 要求程序具用升级功能：初始级别为1级，然后利用升级功能将经理升为4级，兼职技术人员和销售经理升为3级
- 月薪计算方法：经理固定8000元/月；兼职技术人员100元/小时；兼职销售员按本人当月销售额的4%提成，没有固定月薪；销售经理固定月薪为5000元，销售提成为管辖部门内当月销售额的5‰。

综合举例


employee类
technician类
manager类
salesman类
salesmanager类

本章小结

1、基类和派生类；
2、单继承和多继承；
3、二义性与虚基类；

重点：
利用类的继承进行程序设计