第4篇

面向对象的程序设计

第11章 继承与派生
第12章 多态性与虚函数
第13章 输入输出流
第14章 C++工具

第11章 继承与派生

面向对象程序设计有**4**个主要特点**:** 抽象、封装、继承和多态性。

要较好地进行面向对象程序设计，还必须了解面向对象程序设计另外两个重要特征——继承性和多态性。在本章中主要介绍有关继承的知识，在第**12**章中将介绍多态性。

面向对象技术强调软件的可重用性**(software reusability)**。**C++**语言提供了类的继承机制，解决了软件重用问题。

11.1 继承与派生的概念

在C++中可重用性是通过继承(inheritance)这一机制来实现的。继承是C++的一个重要组成部分。

一个类中包含了若干数据成员和成员函数。在不同的类中，数据成员和成员函数是不相同的。但有时两个类的内容基本相同或有一部分相同。

利用原来声明的类Student作为基础，再加上新的内容即可，以减少重复的工作量。 C++提供的继承机制就是为了解决这个问题。

在第8章已举了马的例子来说明继承的概念。见图11.1示意。

在C++中，所谓“继承”就是在一个已存在的类的基础上建立一个新的类。已存在的类(例如“马”)称为“基类(base class)”或“父类(father class)”。新建立的类(例如“公马”)称为“派生类(derived class)”或“子类(son class)”。见图11.2示意。

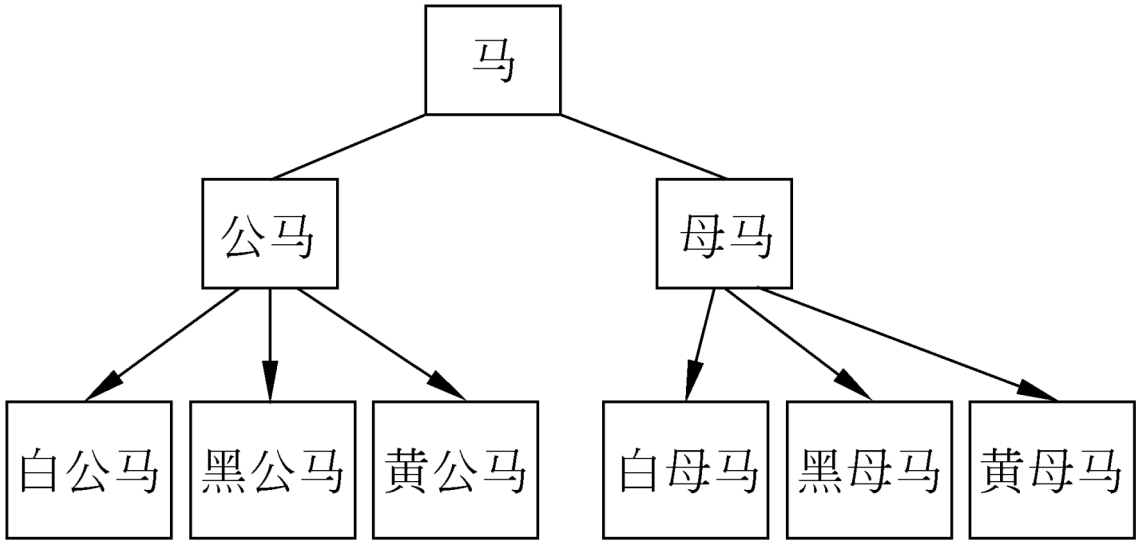


图11.1

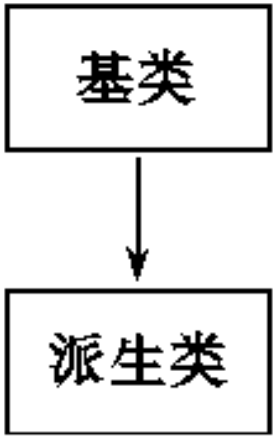


图11.2

一个新类从已有的类那里获得其已有特性，这种现象称为类的继承。通过继承，一个新建子类从已有的父类那里获得父类的特性。从另一角度说，从已有的类(父类)产生一个新的子类，称为类的派生。类的继承是用已有的类来建立专用类的编程技术。派生类继承了基类的所有数据成员和成员函数，并可以对成员作必要的增加或调整。一个基类可以派生出多个派生类，每一个派生类又可以作为基类再派生出新的派生类，因此基类和派生类是相对而言的。

以上介绍的是最简单的情况：一个派生类只从一个基类派生，这称为单继承**(single inheritance)**，这种继承关系所形成的层次是一个树形结构，可以用图**11.3**表示。

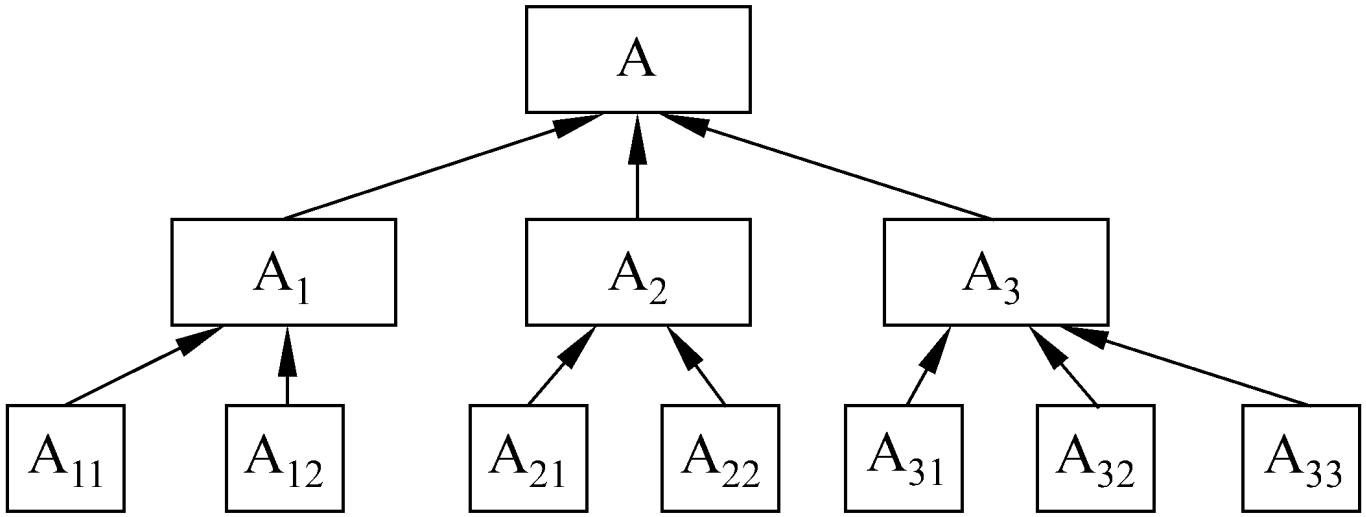


图11.3

请注意图中箭头的方向，在本书中约定，箭头表示继承的方向，从派生类指向基类。

一个派生类不仅可以从一个基类派生，也可以从多个基类派生。一个派生类有两个或多个基类的称为多重继承**(multiple inheritance)**，这种继承关系所形成的结构如图**11.4**所示。

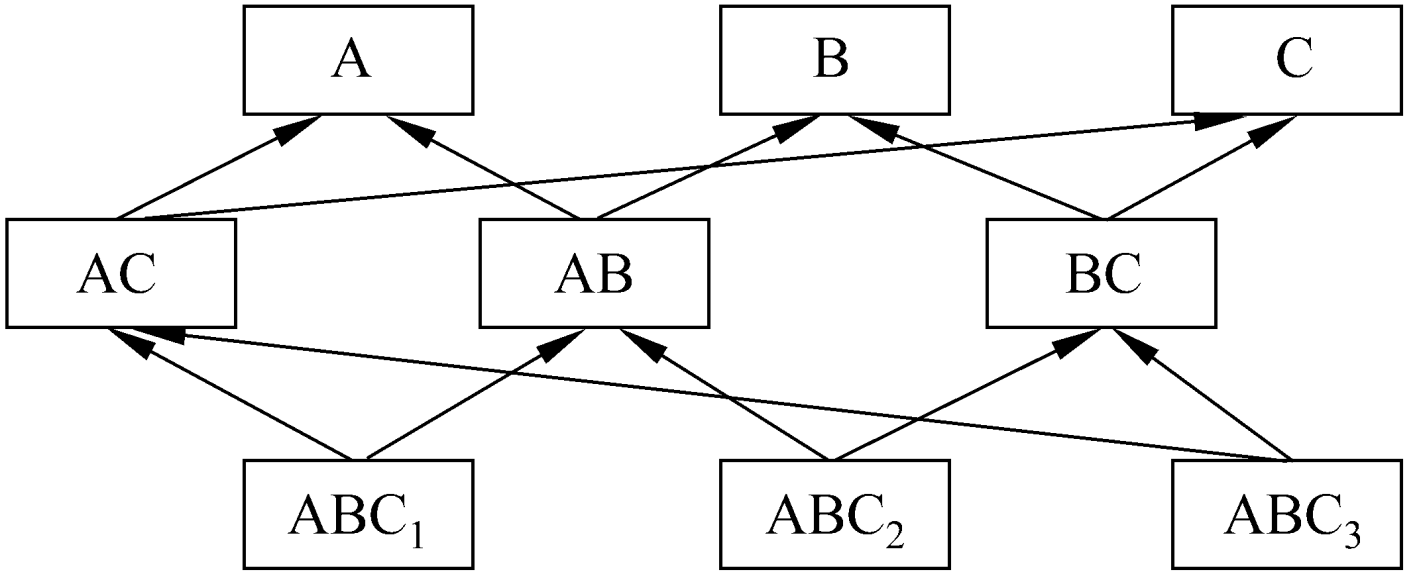


图11.4

关于基类和派生类的关系，可以表述为：派生类是基类的具体化，而基类则是派生类的抽象。

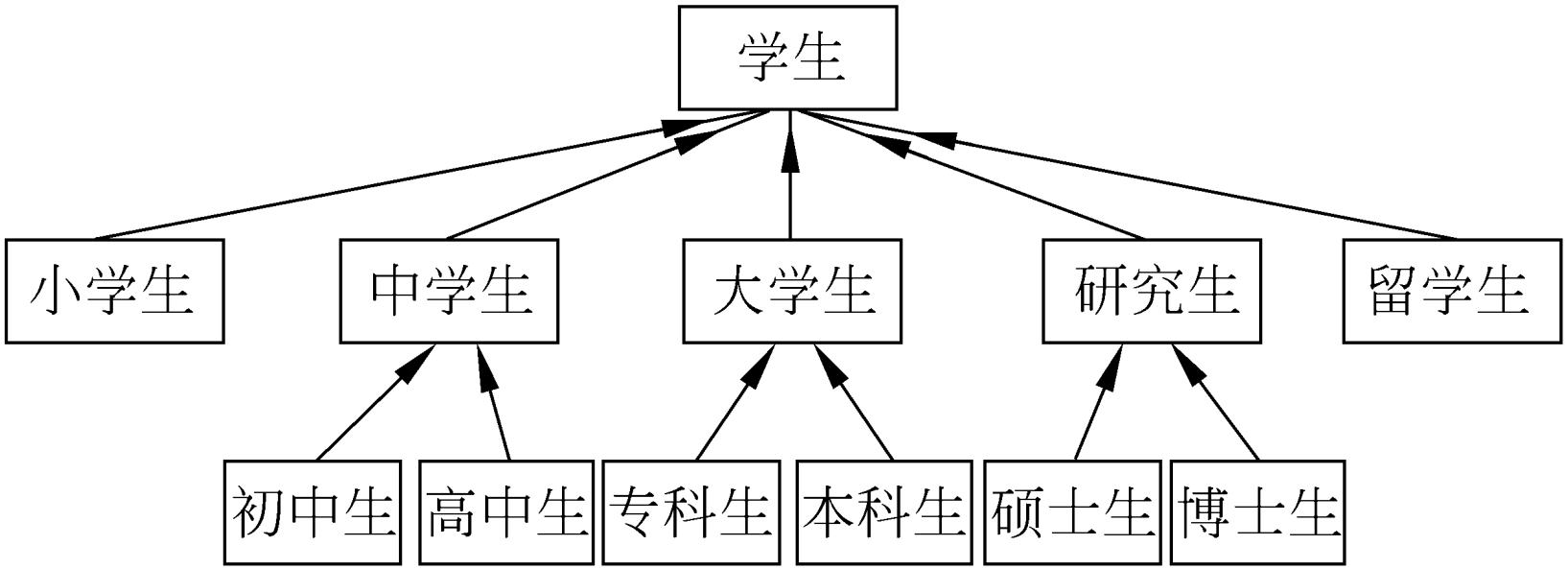


图11.5

11.2 派生类的声明方式

假设已经声明了一个基类**Student**，在此基础上通过单继承建立一个派生类**Student1:**

```
class Student1: public Student//声明基类是Student
{public:
void display_1( )                    //新增加的成员函数
{cout<<"age: "<<age<<endl;
cout<<"address: "<<addr<<endl;}
private:
int age;                             //新增加的数据成员
string addr;                         //新增加的数据成员
};
```

基类名前面有**public**的称为“公用继承**(public inheritance)**”。

声明派生类的一般形式为

class 派生类名**:**［继承方式］ 基类名

{

派生类新增加的成员

};

继承方式包括**: public(**公用的**),private(**私有的**)**和**protected(**受保护的**)**，此项是可选的，如果不写此项，则默认为**private(**私有的**)**。

11.3 派生类的构成

派生类中的成员包括从基类继承过来的成员和自己增加的成员两大部分。在基类中包括数据成员和成员函数(或称数据与方法)两部分，派生类分为两大部分: 一部分是从基类继承来的成员，另一部分是在声明派生类时增加的部分。每一部分均分别包括数据成员和成员函数。

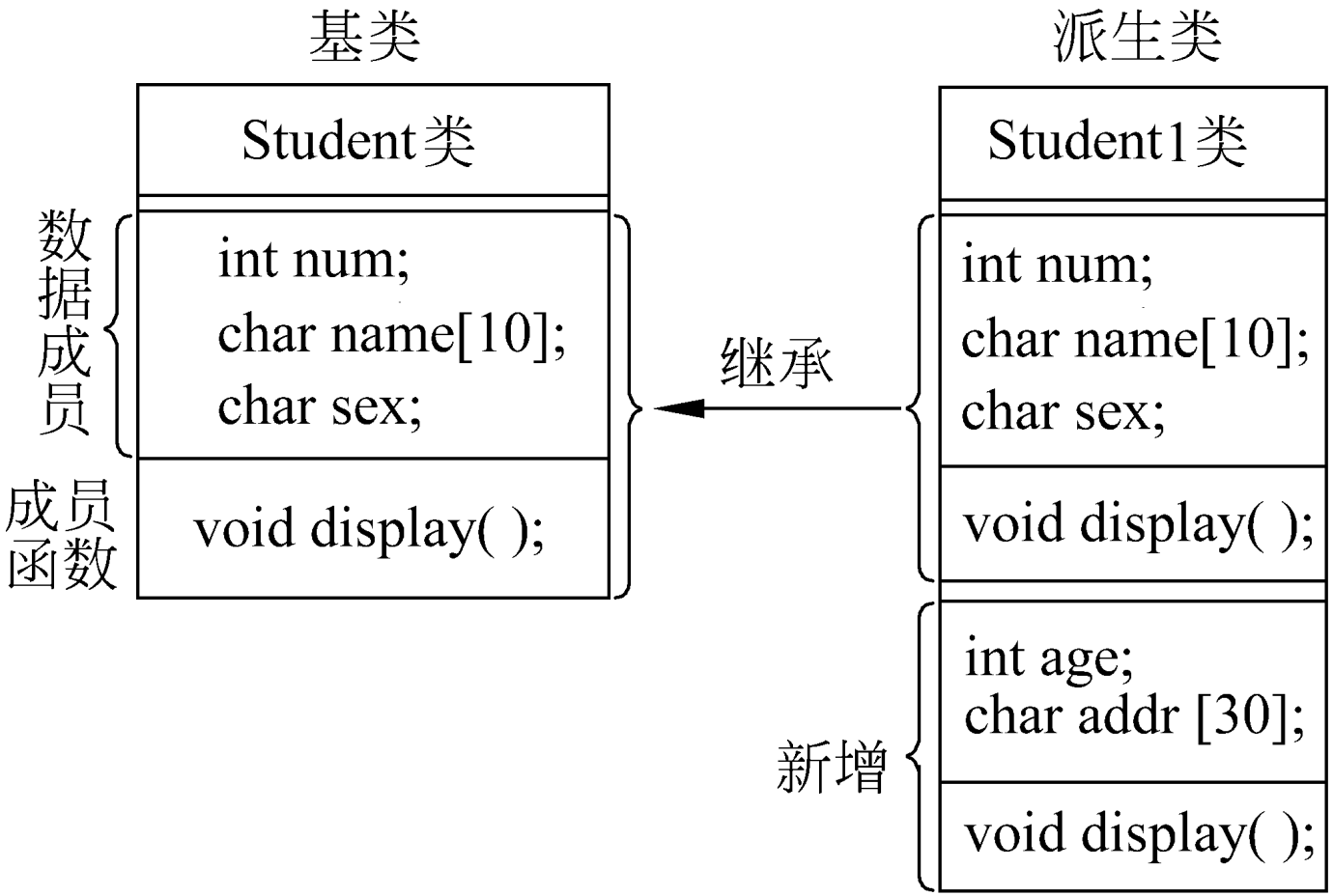

基类
派生类
Student类
Student1类
数据成员
int num;
char name[10];
char sex;
成员函数
void display( );
继承
int num;
char name[10];
char sex;
void display( );
新增
int age;
char addr [30];
void display( );

图11.6

实际上，并不是把基类的成员和派生类自己增加的成员简单地加在一起就成为派生类。构造一个派生类包括以下3部分工作：

(1) 从基类接收成员。派生类把基类全部的成员(不包括构造函数和析构函数)接收过来，也就是说是没有选择的，不能选择接收其中一部分成员，而舍弃另一部分成员。

要求我们根据派生类的需要慎重选择基类，使冗余量最小。事实上，有些类是专门作为基类而设计的，在设计时充分考虑到派生类的要求。

(2) 调整从基类接收的成员。接收基类成员是程序人员不能选择的，但是程序人员可以对这些成员作某些调整。

(3) 在声明派生类时增加的成员。这部分内容是很重要的，它体现了派生类对基类功能的扩展。要根据需要仔细考虑应当增加哪些成员，精心设计。

此外，在声明派生类时，一般还应当自己定义派生类的构造函数和析构函数，因为构造函数和析构函数是不能从基类继承的。

派生类是基类定义的延续。可以先声明一个基类，在此基类中只提供某些最基本的功能，而另外有些功能并未实现，然后在声明派生类时加入某些具体的功能，形成适用于某一特定应用的派生类。通过对基类声明的延续，将一个抽象的基类转化成具体的派生类。因此，派生类是抽象基类的具体实现。

11.4 派生类成员的访问属性

既然派生类中包含基类成员和派生类自己增加的成员，就产生了这两部分成员的关系和访问属性的问题。在建立派生类的时候，并不是简单地把基类的私有成员直接作为派生类的私有成员，把基类的公用成员直接作为派生类的公用成员。实际上，对基类成员和派生类自己增加的成员是按不同的原则处理的。

具体说，在讨论访问属性时，要考虑以下几种情况：

(1) 基类的成员函数访问基类成员。

(2) 派生类的成员函数访问派生类自己增加的成员。

(3) 基类的成员函数访问派生类的成员。

(4) 派生类的成员函数访问基类的成员。

(5) 在派生类外访问派生类的成员。

(6) 在派生类外访问基类的成员。

对于第(1)和第(2)种情况，比较简单，按第8章介绍过的规则处理，即：基类的成员函数可以访问基类成员，派生类的成员函数可以访问派生类成员。私有数据成员只能被同一类中的成员函数访问，公用成员可以被外界访问。

第(3)种情况也比较明确，基类的成员函数只能访问基类的成员，而不能访问派生类的成员。第(5)种情况也比较明确，在派生类外可以访问派生类的公用成员，而不能访问派生类的私有成员。

对于第(4)和第(6)种情况，就稍微复杂一些，也容易混淆。

这些牵涉到如何确定基类的成员在派生类中的访问属性的问题，不仅要考虑对基类成员所声明的访问属性，还要考虑派生类所声明的对基类的继承方式，根据这两个因素共同决定基类成员在派生类中的访问属性。

前面已提到**:** 在派生类中，对基类的继承方式可以有**public(**公用的**)**，**private(**私有的**)**和**protected(**保护的**)3**种。不同的继承方式决定了基类成员在派生类中的访问属性。

简单地说**:**

（1） 公用继承(public inheritance)

基类的公用成员和保护成员在派生类中保持原有访问属性，其私有成员仍为基类私有。

（2） 私有继承(private inheritance)

基类的公用成员和保护成员在派生类中成了私有成员。其私有成员仍为基类私有。

（3） 受保护的继承(protected inheritance)

基类的公用成员和保护成员在派生类中成了保护成员，其私有成员仍为基类私有。

保护成员的意思是: 不能被外界引用，但可以被派生类的成员引用，具体的用法将在稍后介绍。

11.4.1 公用继承

在定义一个派生类时将基类的继承方式指定为**public**的，称为公用继承，用公用继承方式建立的派生类称为公用派生类**(public derived class)**，其基类称为公用基类**(public base class)**。

采用公用继承方式时，基类的公用成员和保护成员在派生类中仍然保持其公用成员和保护成员的属性，而基类的私有成员在派生类中并没有成为派生类的私有成员，它仍然是基类的私有成员，只有基类的成员函数可以引用它，而不能被派生类的成员函数引用，因此就成为派生类中的不可访问的成员。

公用基类的成员在派生类中的访问属性见书中表**11.1**。

例**11.1** 访问公有基类的成员。

下面写出类的声明部分：

```
Class Student//声明基类
{public:                                  /基类公用成员
void get_value( )
{cin>>num>>name>>sex;}
void display( )
{cout<<" num: "<<num<<endl;
cout<<" name: "<<name<<endl;
cout<<" sex: "<<sex<<endl;}
private :                                 /基类私有成员
  int num;
  string name;
  char sex;
};

class Student1: public Student            //以public方式声明派生类Student1
```

```
 {public:
void display_1( )
{cout<<" num: "<<num<<endl;          //企图引用基类的私有成员，错误
     cout<<" name: "<<name<<endl;        //企图引用基类的私有成员，错误
     cout<<" sex: "<<sex<<endl;          //企图引用基类的私有成员，错误
     cout<<" age: "<<age<<endl;          //引用派生类的私有成员，正确
     cout<<" address: "<<addr<<endl;}    //引用派生类的私有成员，正确
private:
     int age;
     string addr;
 };
```

由于基类的私有成员对派生类来说是不可访问的，因此在派生类中的**display_1**函数中直接引用基类的私有数据成员**num**，**name**和**sex**是不允许的。只能通过基类的公用成员函数来引用基类的私有数据成员。

可以将派生类**Student1**的声明改为

```
class Student1: public Student//以public方式声明派生类Student1
{public:
void display_1( )
{cout<<" age: "<<age<<endl;          //引用派生类的私有成员，正确
cout<<" address: "<<addr<<endl;     //引用派生类的私有成员，正确
}
private:
int age;
string addr;
};
```

然后在**main**函数中分别调用基类的**display**函数和派生类中的**display_1**函数，先后输出**5**个数据。

可以这样写**main**函数（假设对象**stud**中已有数据）：

```
int main( )
{Student1  stud;//定义派生类Student1的对象stud
```

```
┆
stud.display( );        //调用基类的公用成员函数，输出基类中3个数据成员的值
stud.display_1();       //调用派生类的公用成员函数，输出派生类中两个数据成员的值
return 0;
}
```

请根据上面的分析，写出完整的程序，程序中应包括输入数据的函数。

实际上，程序还可以改进，在派生类的**display_1**函数中调用基类的**display**函数，在主函数中只要写一行：

```
stud.display_1();
```

即可输出**5**个数据。

11.4.2 私有继承

在声明一个派生类时将基类的继承方式指定为**private**的，称为私有继承，用私有继承方式建立的派生类称为私有派生类**(private derived class)**，其基类称为私有基类**(private base class)**。

私有基类的公用成员和保护成员在派生类中的访问属性相当于派生类中的私有成员，即派生类的成员函数能访问它们，而在派生类外不能访问它们。私有基类的私有成员在派生类中成为不可访问的成员，只有基类的成员函数可以引用它们。一个基类成员在基类中的访问属性和在派生类中的访问属性可能是不同的。私有基类的成员在私有派生类中的访问属性见书中表**11.2**。

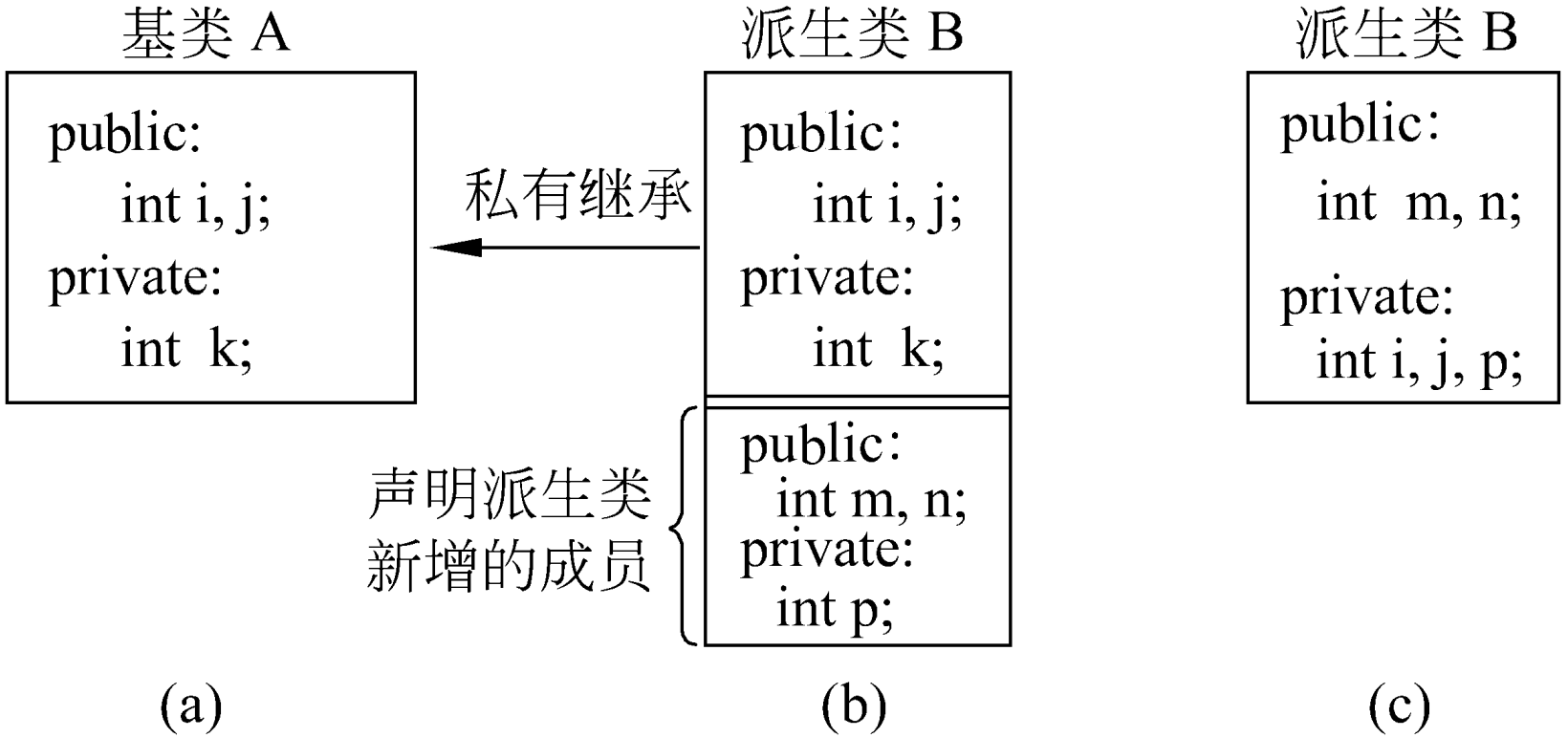


图11.7

图11.7表示了各成员在派生类中的访问属性。

对表**11.2**的规定不必死记，只需理解**:** 既然声明为私有继承，就表示将原来能被外界引用的成员隐藏起来，不让外界引用，因此私有基类的公用成员和保护成员理所当然地成为派生类中的私有成员。私有基类的私有成员按规定只能被基类的成员函数引用，在基类外当然不能访问他们，因此它们在派生类中是隐蔽的，不可访问的。

对于不需要再往下继承的类的功能可以用私有继承方式把它隐蔽起来，这样，下一层的派生类无法访问它的任何成员。

可以知道**:** 一个成员在不同的派生层次中的访问属性可能是不同的。它与继承方式有关。

例11.2 将例11.1中的公用继承方式改为用私有继承方式(基类Student不改)。

可以写出私有派生类如下：

```
class Student1: private Student//用私有继承方式声明派生类Student1
{public:
void display_1( )                    //输出两个数据成员的值
{cout<<"age: "<<age<<endl;          //引用派生类的私有成员，正确
cout<<"address: "<<addr<<endl;}    //引用派生类的私有成员，正确
private:
int age;
string addr;
};请分析下面的主函数:int main( )
{Student1  stud1;//定义一个Student1类的对象stud1
 stud1.display();            //错误，私有基类的公用成员函数在派生类中是私有函数
stud1.display_1( );          //正确。Display_1函数是Student1类的公用函数
stud1.age=18;               //错误。外界不能引用派生类的私有成员
return 0;
 }
```

可以看到:

(1) 不能通过派生类对象(如**stud1**)引用从私有基类继承过来的任何成员(如**stud1.display()**或**stud1.num**)。

(2) 派生类的成员函数不能访问私有基类的私有成员，但可以访问私有基类的公用成员(如**stud1.display_1**函数可以调用基类的公用成员函数**display**，但不能引用基类的私有成员**num**)。

有没有办法调用私有基类的公用成员函数，从而引用私有基类的私有成员呢？有。应当注意到: 虽然在派生类外不能通过派生类对象调用私有基类的公用成员函数，但可以通过派生类的成员函数调用私有基类的公用成员函数(此时它是派生类中的私有成员函数，可以被派生类的任何成员函数调用)。

可将上面的私有派生类的成员函数定义改写为

```
void display_1( )//输出5个数据成员的值
{display():                    //调用基类的公用成员函数，输出3个数据成员的值
 cout<<"age: "<<age<<endl;      //输出派生类的私有数据成员
cout<<"address: "<<addr<<endl;}  //输出派生类的私有数据成员
```

main函数可改写为

```
int main( )
{Student1 stud1;
 stud1.display_1( );//display_1函数是派生类Student1类的公用函数
 return 0;
}
```

这样就能正确地引用私有基类的私有成员。可以看到，本例采用的方法是：

① 在**main**函数中调用派生类中的公用成员函数**stud1.display_1**；

② 通过该公用成员函数调用基类的公用成员函数**display(**它在派生类中是私有函数，可以被派生类中的任何成员函数调用**)**;

③ 通过基类的公用成员函数**display**引用基类中的数据成员。

请根据上面的要求，补充和完善上面的程序，使之成为完整、正确的程序。程序中应包括输入数据的函数。

由于私有派生类限制太多，使用不方便，一般不常使用。

11.4.3 保护成员和保护继承

由**protected**声明的成员称为“受保护的成员”，或简称“保护成员”。从类的用户角度来看，保护成员等价于私有成员。但有一点与私有成员不同，保护成员可以被派生类的成员函数引用。见图**11.8**示意。

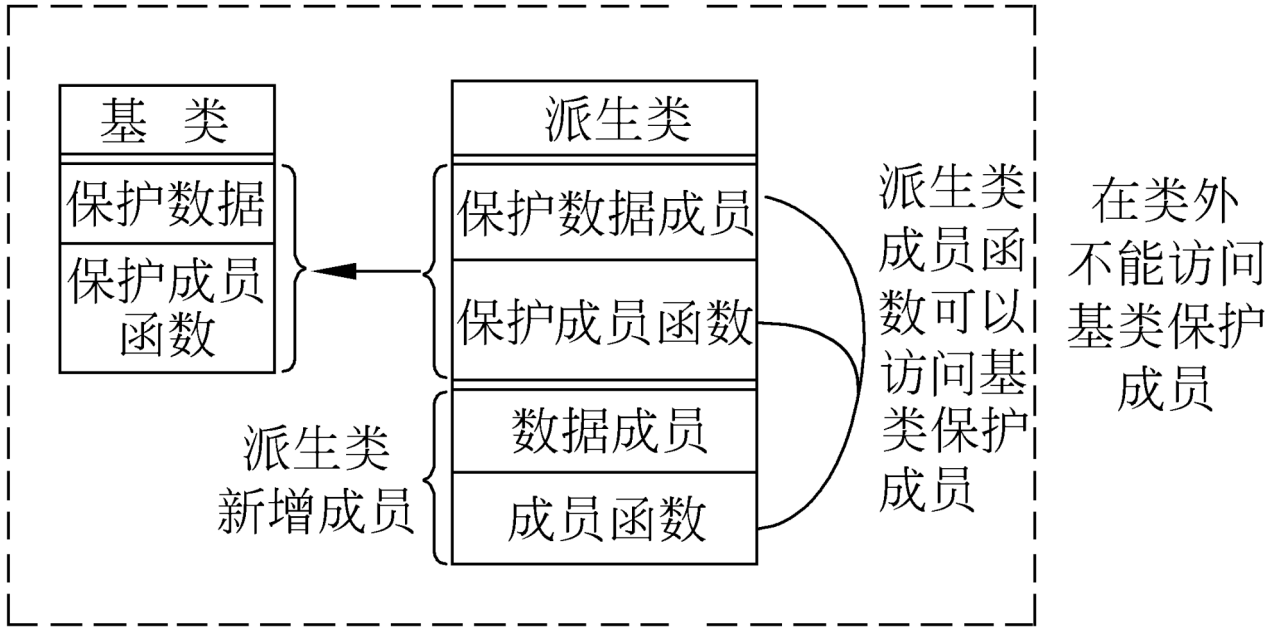


图11.8

如果基类声明了私有成员，那么任何派生类都是不能访问它们的，若希望在派生类中能访问它们，应当把它们声明为保护成员。如果在一个类中声明了保护成员，就意味着该类可能要用作基类，在它的派生类中会访问这些成员。

在定义一个派生类时将基类的继承方式指定为**protected**的，称为保护继承，用保护继承方式建立的派生类称为保护派生类**(protected derived class)**，其基类称为受保护的基类**(protected base class)**，简称保护基类。

保护继承的特点是**:** 保护基类的公用成员和保护成员在派生类中都成了保护成员，其私有成员仍为基类私有。也就是把基类原有的公用成员也保护起来，不让类外任意访问。

将表**11.1**和表**11.2**综合起来，并增加保护继承的内容，见书中表**11.3**。

保护基类的所有成员在派生类中都被保护起来，类外不能访问，其公用成员和保护成员可以被其派生类的成员函数访问。

比较一下私有继承和保护继承(也就是比较在私有派生类中和在保护派生类中的访问属性)，可以发现，在直接派生类中，以上两种继承方式的作用实际上是相同的: 在类外不能访问任何成员，而在派生类中可以通过成员函数访问基类中的公用成员和保护成员。但是如果继续派生，在新的派生类中，两种继承方式的作用就不同了。

例如，如果以公用继承方式派生出一个新派生类，原来私有基类中的成员在新派生类中都成为不可访问的成员，无论在派生类内或外都不能访问，而原来保护基类中的公用成员和保护成员在新派生类中为保护成员，可以被新派生类的成员函数访问。

从表**11.3**可知**:** 基类的私有成员被派生类继承后变为不可访问的成员，派生类中的一切成员均无法访问它们。如果需要在派生类中引用基类的某些成员，应当将基类的这些成员声明为**protected**，而不要声明为**private**。

如果善于利用保护成员，可以在类的层次结构中找到数据共享与成员隐蔽之间的结合点。既可实现某些成员的隐蔽，又可方便地继承，能实现代码重用与扩充。

通过以上的介绍，可以知道：

(1) 在派生类中，成员有**4**种不同的访问属性：

① 公用的，派生类内和派生类外都可以访问。

② 受保护的，派生类内可以访问，派生类外不能访问，其下一层的派生类可以访问。

③ 私有的，派生类内可以访问，派生类外不能访问。

④ 不可访问的，派生类内和派生类外都不能访问。

可以用书中表**11.4**表示。

需要说明的是：

① 这里所列出的成员的访问属性是指在派生类中所获得的访问属性。

② 所谓在派生类外部，是指在建立派生类对象的模块中，在派生类范围之外。

③ 如果本派生类继续派生，则在不同的继承方式下，成员所获得的访问属性是不同的，在本表中只列出在下一层公用派生类中的情况，如果是私有继承或保护继承，读者可以从表**11.3**中找到答案。

(2) 类的成员在不同作用域中有不同的访问属性，对这一点要十分清楚。

下面通过一个例子说明怎样访问保护成员。

例11.3 在派生类中引用保护成员。

```
#include <iostream>
#include <string>
using namespace std;
class Student//声明基类
{public:                          //基类公用成员
  void display( );
 protected :                       //基类保护成员
int num;
string name;
char sex;
};

void Student::display( )            //定义基类成员函数
   {cout<<"num: "<<num<<endl;
    cout<<"name: "<<name<<endl;
    cout<<"sex: "<<sex<<endl;
   }

class Student1: protected Student       //用protected方式声明派生类Student1
{public:
void display1( );                    //派生类公用成员函数
```

```
private:
int age;                          //派生类私有数据成员
string addr;                      //派生类私有数据成员
};

void Student1::display1( )        //定义派生类公用成员函数
   {cout<<"num: "<<num<<endl;     //引用基类的保护成员，合法
    cout<<"name: "<<name<<endl;   //引用基类的保护成员，合法
  cout<<"sex: "<<sex<<endl;       //引用基类的保护成员，合法
    cout<<"age: "<<age<<endl;     //引用派生类的私有成员，合法
    cout<<"address: "<<addr<<endl;//引用派生类的私有成员，合法
  }

int main( )
 {Student1 stud1;                 //stud1是派生类Student1类的对象
  stud1.display1( );              //合法，display1是派生类中的公用成员函数
  stud1.num=10023;                //错误，外界不能访问保护成员
return 0;
 }
```

在派生类的成员函数中引用基类的保护成员是合法的。保护成员和私有成员不同之处，在于把保护成员的访问范围扩展到派生类中。

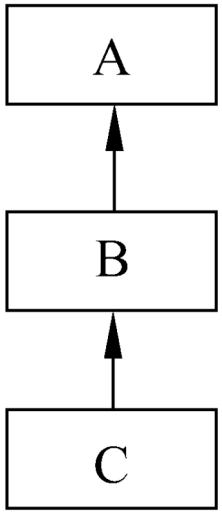
注意: 在程序中通过派生类**Student1**的对象**stud1**的公用成员函数**display1**去访问基类的保护成员**num.name**和**sex**，不要误认为可以通过派生类对象名去访问基类的保护成员。

请补充、修改上面的程序，使之能正常运行。程序中应包括输入数据的函数。

私有继承和保护继承方式在使用时需要十分小心，很容易搞错，一般不常用。

11.4.4 多级派生时的访问属性

图11.9



如果有图11.9所示的派生关系: 类A为基类，类B是类A的派生类，类C是类B的派生类，则类C也是类A的派生类。类B称为类A的直接派生类，类C称为类A的间接派生类。类A是类B的直接基类，是类C的间接基类。在多级派生的情况下，各成员的访问属性仍按以上原则确定。

例**11.4** 多级派生的访问属性。

如果声明了以下的类**:**

```
class A                          /基类
{public:
  int i;
 protected:
  void f2( );
  int j;
 private:
  int k;
};

class B: public A                //public方式
{public:
void f3( );
 protected:
void f4( );
 private:
  int m;
};
```

```
class C: protected B          //protected方式
{public:
void f5( );
 private:
  int n;
};
```

类A是类B的公用基类，类B是类C的保护基类。各成员在不同类中的访问属性如下：

	i	f2	j	k	f3	f4	m	f5	n
基类A	公用	保护	保护	私有					
公用派生类B	公用	保护	保护	不可访问	公用	保护	私有		
保护派生类C	保护	保护	保护	不可访问	保护	保护	不可访问	公用	私有

无论哪一种继承方式，在派生类中是不能访问基类的私有成员的，私有成员只能被本类的成员函数所访问，毕竟派生类与基类不是同一个类。如果在多级派生时都采用公用继承方式，那么直到最后一级派生类都能访问基类的公用成员和保护成员。如果采用私有继承方式，经过若干次派生之后，基类的所有的成员已经变成不可访问的了。如果采用保护继承方式，在派生类外是无法访问派生类中的任何成员的。而且经过多次派生后，人们很难清楚地记住哪些成员可以访问，哪些成员不能访问，很容易出错。因此，在实际中，常用的是公用继承。

11.5 派生类的构造函数和析构函数

用户在声明类时可以不定义构造函数，系统会自动设置一个默认的构造函数，在定义类对象时会自动调用这个默认的构造函数。这个构造函数实际上是一个空函数，不执行任何操作。如果需要对类中的数据成员初始化，应自己定义构造函数。

构造函数的主要作用是对数据成员初始化。在设计派生类的构造函数时，不仅要考虑派生类所增加的数据成员的初始化，还应当考虑基类的数据成员初始化。也就是说，希望在执行派生类的构造函数时，使派生类的数据成员和基类的数据成员同时都被初始化。解决这个问题的思路是：在执行派生类的构造函数时，调用基类的构造函数。

11.5.1 简单的派生类的构造函数

任何派生类都包含基类的成员，简单的派生类只有一个基类，而且只有一级派生(只有直接派生类，没有间接派生类)，在派生类的数据成员中不包含基类的对象(即子对象)。

例11.5 简单的派生类的构造函数。

```
#include <iostream>
#include<string>
using namespace std;
class Student//声明基类Student
 {public:
   Student(int n,string nam,char s)          //基类构造函数
    {num=n;
     name=nam;
     sex=s; }
~Student( ){ }                               //基类析构函数
```

```
  protected:                          //保护部分
   int num;
   string name;
   char sex ;
};

class Student1: public Student      //声明派生类Student1
 {public:                           //派生类的公用部分
  Student1(int n,string nam,char s,int a,string ad):Student(n,nam,s)
 //派生类构造函数
{age=a;                             //在函数体中只对派生类新增的数据成员初始化
    addr=ad;
}
  void show( )
   {cout<<"num: "<<num<<endl;
    cout<<"name: "<<name<<endl;
    cout<<"sex: "<<sex<<endl;
    cout<<"age: "<<age<<endl;
    cout<<"address: "<<addr<<endl<<endl;
   }
  ~Student1( ){ }                   //派生类析构函数
 private:                           //派生类的私有部分
```

```
    int age;
    string addr;
};

int main( )
 {Student1 stud1(10010,"Wang-li",'f',19,"115 Beijing Road,Shanghai");
  Student1 stud2(10011,"Zhang-fun",'m',21,"213 Shanghai Road,Beijing");
  stud1.show( );                    //输出第一个学生的数据
  stud2.show( );                    //输出第二个学生的数据
  return 0;
}
```

运行结果为

```
num:10010
name:Wang-li
sex:f
address: 115 Beijing Road,Shanghai

num:10011
name:Zhang-fun
sex:m
address: 213 Shanghai Road,Beijing
```

请注意派生类构造函数首行的写法：

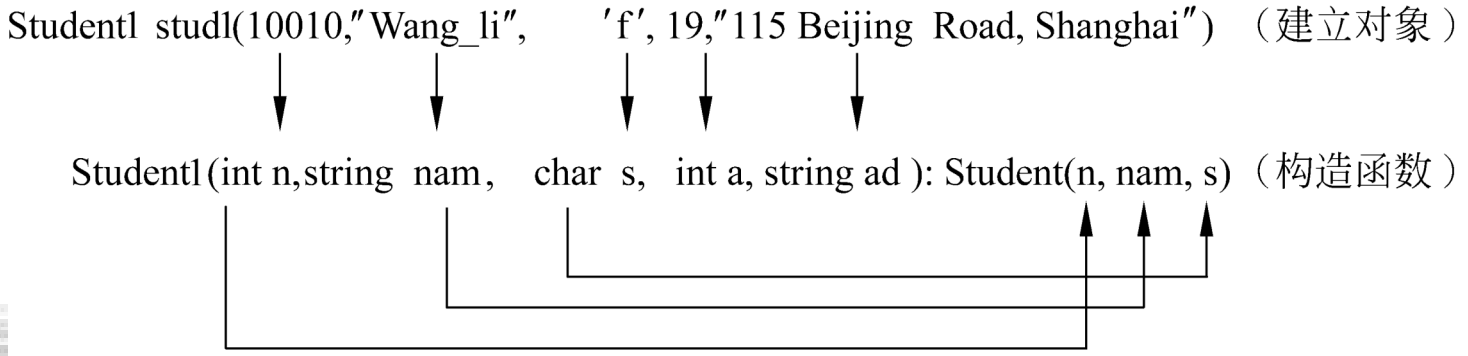
Student1(int n，string nam，char s，int a，string ad):Student(n，nam，s)

其一般形式为

派生类构造函数名（总参数表列）：基类构造函数名（参数表列）

{派生类中新增数据成员初始化语句}

在**main**函数中，建立对象**stud1**时指定了**5**个实参。它们按顺序传递给派生类构造函数**Student1**的形参。然后，派生类构造函数将前面**3**个传递给基类构造函数的形参。见图**11.10**。　　图**11.10**



通过**Student (n， nam， s)**把**3**个值再传给基类构造函数的形参，见图**11.11**。

```
Student(      n，      nam，          s)
                ↓       ↓              ↓
Student(int   n，      string nam，char s)//这是基类构造函数的首部
```

图**11.11**

在上例中也可以将派生类构造函数在类外面定义，而在类体中只写该函数的声明：

```
Student1(int n，string nam，char s，int a，string ad);
```

在类的外面定义派生类构造函数：

```
Student1∶∶Student1(int n，string nam,char s,int a，string
ad):Student(n，nam，s)
 {age=a;
   addr=ad;
}
```

请注意: 在类中对派生类构造函数作声明时，不包括基类构造函数名及其参数表列（即**Student(n，nam，s)**）。只在定义函数时才将它列出。

在以上的例子中，调用基类构造函数时的实参是从派生类构造函数的总参数表中得到的，也可以不从派生类构造函数的总参数表中传递过来，而直接使用常量或全局变量。例如，派生类构造函数首行可以写成以下形式:

Student1(string nam，char s，int a，string ad):Student(10010，nam，s)

即基类构造函数**3**个实参中，有一个是常量**10010**，另外两个从派生类构造函数的总参数表传递过来。

请回顾一下在第**9**章介绍过的构造函数初始化表的例子:

```
Box::Box(int h,int w,int len):height(h)，width(w)，length(len)
{}
```

它也有一个冒号，在冒号后面的是对数据成员的初始化表。实际上，本章介绍的在派生类构造函数中对基类成员初始化，就是在第9章介绍的构造函数初始化表。也就是说，不仅可以利用初始化表对构造函数的数据成员初始化，而且可以利用初始化表调用派生类的基类构造函数，实现对基类数据成员的初始化。也可以在同一个构造函数的定义中同时实现这两种功能。例如，例11.5中派生类的基类构造函数的定义采用了下面的形式：

```
Student1(int n, string nam,char s,int a, string ad):Student(n,nam,s)
   {age=a;//在函数体中对派生类数据成员初始化
    addr=ad;
   }
```

可以将对**age**和**addr**的初始化也用初始化表处理，将构造函数改写为以下形式**:**

Student1(int n, string nam,char s,int a, string ad):Student(n,nam,s),age(a),addr(ad){}

这样函数体为空，更显得简单和方便。

在建立一个对象时，执行构造函数的顺序是**:** ①派生类构造函数先调用基类构造函数；②再执行派生类构造函数本身**(**即派生类构造函数的函数体**)**。对上例来说，先初始化**num**，**name**，**sex**，然后再初始化**age**和**addr**。

在派生类对象释放时，先执行派生类析构函数**~Student1( )**，再执行其基类析构函数**~Student( )**。

11.5.2 有子对象的派生类的构造函数

类的数据成员中还可以包含类对象，如可以在声明一个类时包含这样的数据成员：

Student s1;// Student是已声明的类名，**s1**是**Student**类的对象

这时，**s1**就是类对象中的内嵌对象，称为子对象**(subobject)**，即对象中的对象。

通过例子来说明问题。

例**11.6** 包含子对象的派生类的构造函数。

为了简化程序以易于阅读，这里设基类**Student**的数据成员只有两个，即**num**和**name**。

```
#include <iostream>
#include <string>
using namespace std;
class Student//声明基类
 {public:                              //公用部分
Student(int n, string nam )           //基类构造函数，与例11.5相同
    {num=n;
     name=nam;
    }
   void display( )                    //成员函数，输出基类数据成员
    {cout<<"num:"<<num<<endl<<"name:"<<name<<endl;}
protected:                            //保护部分
int num;
    string name;
};

class Student1: public Student        //声明公用派生类Student1
```

```
{public:
 Student1(int n, string nam,int n1, string nam1,int a, string ad)
    :Student(n,nam),monitor(n1,nam1)                    //派生类构造函数
  {age=a;
   addr=ad;
  }
 void show( )
  {cout<<"This student is:"<<endl;
   display();                              //输出num和name
   cout<<"age: "<<age<<endl;                    //输出age
   cout<<"address: "<<addr<<endl<<endl;     //输出addr
  }

    void show_monitor( )                        //成员函数，输出子对象
  {cout<<endl<<"Class monitor is:"<<endl;
   monitor.display( );                          //调用基类成员函数
}
 private:                                  //派生类的私有数据
  Student monitor;                             //定义子对象(班长)
  int age;
  string addr;
};
```

```
int main( )
{Student1 stud1(10010,"Wang-li",10001,"Li-sun",19,"115 Beijing
Road,Shanghai");
  stud1.show( );                    //输出学生的数据
  stud1.show_monitor();             //输出子对象的数据
return 0;
}
```

运行时的输出如下：

```
This student is:
num: 10010
name: Wang-li
age: 19
address:115 Beijing Road,Shanghai

Class monitor is:
num:10001
name:Li-sun
```

派生类构造函数的任务应该包括3个部分：

（1） 对基类数据成员初始化；

（2） 对子对象数据成员初始化；

（3） 对派生类数据成员初始化。

程序中派生类构造函数首部如下：

```
Student1(int n, string nam,int n1, string nam1,int a, string ad):
Student(n,nam),monitor(n1,nam1)
```

在上面的构造函数中有6个形参，前两个作为基类构造函数的参数，第3、第4个作为子对象构造函数的参数，第5、第6个是用作派生类数据成员初始化的。见图11.12。

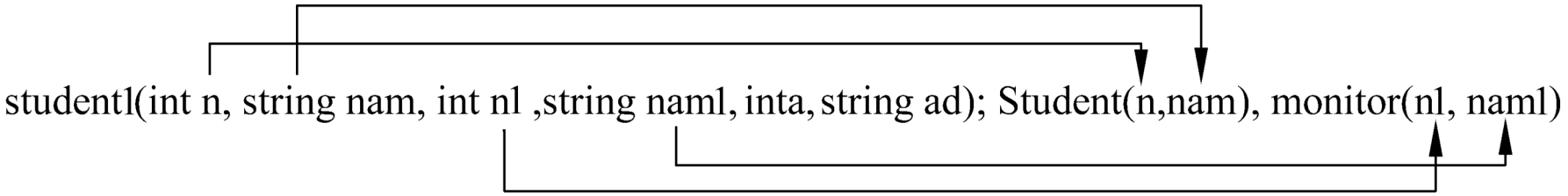


图11.12

归纳起来，定义派生类构造函数的一般形式为

派生类构造函数名（总参数表列）：基类构造函数名（参数表列），子对象名(参数表列)

{派生类中新增数成员据成员初始化语句}

执行派生类构造函数的顺序是：

① 调用基类构造函数，对基类数据成员初始化；

② 调用子对象构造函数，对子对象数据成员初始化；

③ 再执行派生类构造函数本身，对派生类数据成员初始化。

派生类构造函数的总参数表列中的参数，应当包括基类构造函数和子对象的参数表列中的参数。基类构造函数和子对象的次序可以是任意的，如上面的派生类构造函数首部可以写成

Student1(int n, string nam,int n1, string nam1,int a, string ad):

monitor(n1,nam1),Student(n,nam)

编译系统是根据相同的参数名(而不是根据参数的顺序)来确立它们的传递关系的。但是习惯上一般先写基类构造函数。

如果有多个子对象，派生类构造函数的写法依此类

11.5.3 多层派生时的构造函数

一个类不仅可以派生出一个派生类，派生类还可以继续派生，形成派生的层次结构。在上面叙述的基础上，不难写出在多级派生情况下派生类的构造函数。

例11.7 多级派生情况下派生类的构造函数。

```
#include <iostream>
#include<string>
using namespace std;
class Student//声明基类
 {public:                              //公用部分
   Student(int n, string nam )          //基类构造函数
    {num=n;
     name=nam;
    }
   void display( )                      //输出基类数据成员
```

```
    {cout<<"num:"<<num<<endl;
     cout<<"name:"<<name<<endl;
    }
  protected:                          //保护部分
    int num;                          //基类有两个数据成员
    string name;
};

class Student1: public Student          //声明公用派生类Student1
 {public:
   Student1(int n,char nam[10],int a):Student(n,nam)//派生类构造函数
    {age=a; }                    //在此处只对派生类新增的数据成员初始化
   void show( )                        //输出num，name和age
    {display( );                       //输出num和name
     cout<<"age: "<<age<<endl;
    }
   private:                           //派生类的私有数据
    int age;                          //增加一个数据成员
  };

class Student2:public Student1          //声明间接公用派生类Student2
 {public:
```

```
  //下面是间接派生类构造函数
  Student2(int n, string nam,int a,int s):Student1(n,nam,a)
{score=s;}
  void show_all( )                    //输出全部数据成员
{show( );                             //输出num和name
    cout<<"score:"<<score<<endl;      //输出age
   }
 private:
  int score;                          //增加一个数据成员
};

int main( )
 {Student2 stud(10010,"Li",17,89);
  stud.show_all( );                   //输出学生的全部数据
return 0;
}
```

运行时的输出如下：

```
num:10010
name:Li
age:17
score:89
```

其派生关系如图**11.13**所示。

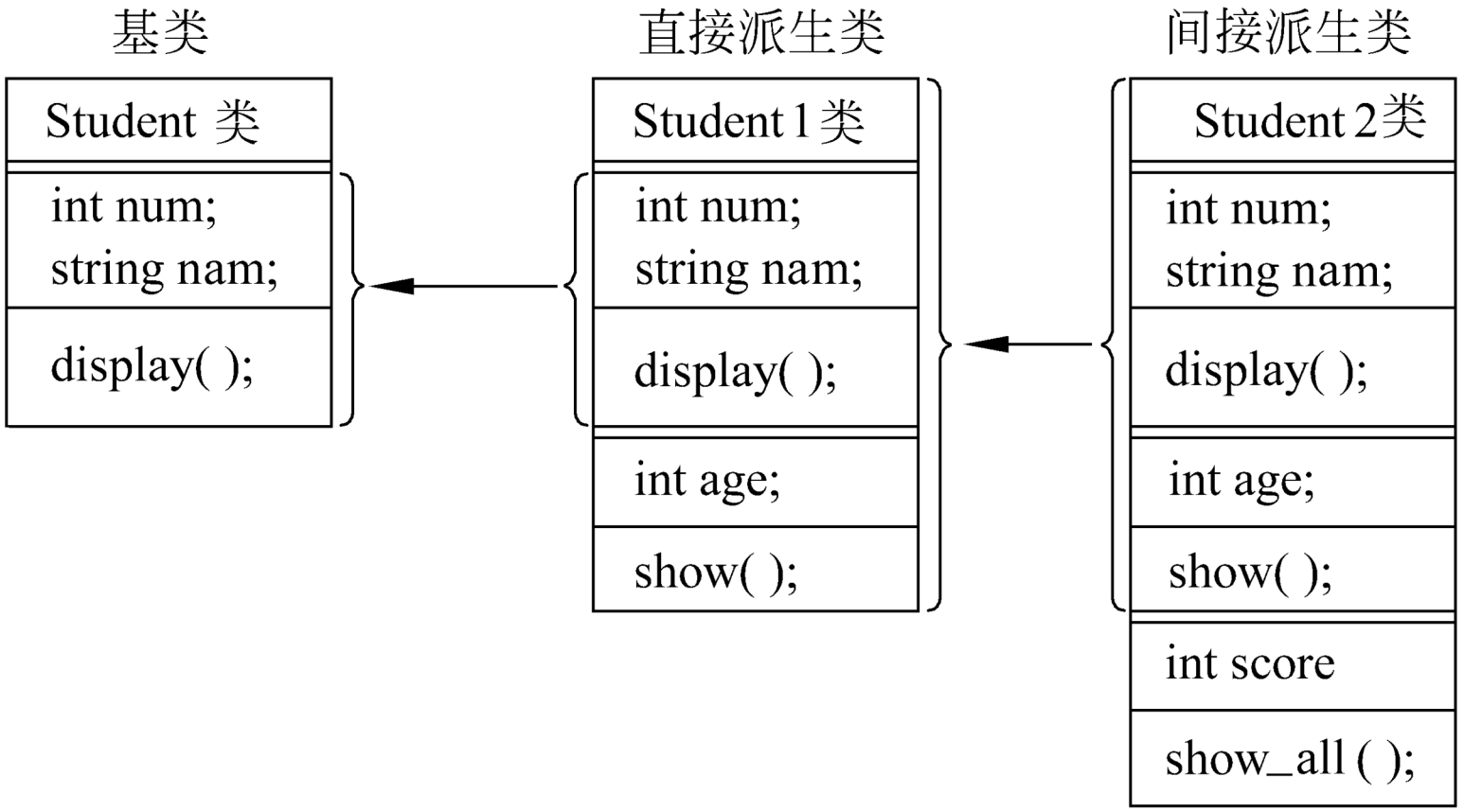


图11.13

请注意基类和两个派生类的构造函数的写法：

基类的构造函数首部：

Student(int n, string nam)

派生类**Student1**的构造函数首部：

Student1(int n, string nam],int a):Student(n,nam)

派生类**Student2**的构造函数首部：

Student2(int n, string nam,int a,int s):Student1(n,nam,a)

在声明**Student2**类对象时，调用**Student2**构造函数；在执行**Student2**构造函数时，先调用**Student1**构造函数；在执行**Student1**构造函数时，先调用基类**Student**构造函数。初始化的顺序是：

① 先初始化基类的数据成员**num**和**name**。

② 再初始化**Student1**的数据成员**age**。

③ 最后再初始化**Student2**的数据成员**score**。

11.5.4 派生类构造函数的特殊形式

在使用派生类构造函数时，有以下特殊的形式：

(1) 当不需要对派生类新增的成员进行任何初始化操作时，派生类构造函数的函数体可以为空，即构造函数是空函数，如例**11.6**程序中派生类**Student1**构造函数可以改写为

```
Student1(int n, strin nam,int n1, strin nam1):Student(n,nam),
monitor(n1,nam1) { }
```

此派生类构造函数的作用只是为了将参数传递给基类构造函数和子对象，并在执行派生类构造函数时调用基类构造函数和子对象构造函数。在实际工作中常见这种用法。

(2) 如果在基类中没有定义构造函数，或定义了没有参数的构造函数，那么在定义派生类构造函数时可不写基类构造函数。因为此时派生类构造函数没有向基类构造函数传递参数的任务。调用派生类构造函数时系统会自动首先调用基类的默认构造函数。

如果在基类和子对象类型的声明中都没有定义带参数的构造函数，而且也不需对派生类自己的数据成员初始化，则可以不必显式地定义派生类构造函数。因为此时派生类构造函数既没有向基类构造函数和子对象构造函数传递参数的任务，也没有对派生类数据成员初始化的任务。在建立派生类对象时，系统会自动调用系统提供的派生类的默认构造函数，并在执行派生类默认构造函数的过程中，调用基类的默认构造函数和子对象类型默认构造函数。

如果在基类或子对象类型的声明中定义了带参数的构造函数，那么就必须显式地定义派生类构造函数，并在派生类构造函数中写出基类或子对象类型的构造函数及其参数表。

如果在基类中既定义无参的构造函数，又定义了有参的构造函数(构造函数重载)，则在定义派生类构造函数时，既可以包含基类构造函数及其参数，也可以不包含基类构造函数。在调用派生类构造函数时，根据构造函数的内容决定调用基类的有参的构造函数还是无参的构造函数。编程者可以根据派生类的需要决定采用哪一种方式。

11.5.5 派生类的析构函数

在派生时，派生类是不能继承基类的析构函数的，也需要通过派生类的析构函数去调用基类的析构函数。在派生类中可以根据需要定义自己的析构函数，用来对派生类中所增加的成员进行清理工作。基类的清理工作仍然由基类的析构函数负责。在执行派生类的析构函数时，系统会自动调用基类的析构函数和子对象的析构函数，对基类和子对象进行清理。

调用的顺序与构造函数正好相反：先执行派生类自己的析构函数，对派生类新增加的成员进行清理，然后调用子对象的析构函数，对子对象进行清理，最后调用基类的析构函数，对基类进行清理。

11.6 多重继承

前面讨论的是单继承，即一个类是从一个基类派生而来的。实际上，常常有这样的情况：一个派生类有两个或多个基类，派生类从两个或多个基类中继承所需的属性。**C++**为了适应这种情况，允许一个派生类同时继承多个基类。这种行为称为多重继承**(multiple inheritance)**。

11.6.1 声明多重继承的方法

如果已声明了类**A**、类**B**和类**C**，可以声明多重继承的派生类**D:**

class D:public A,private B,protected C

{类**D**新增加的成员**}**

D是多重继承的派生类，它以公用继承方式继承**A**类，以私有继承方式继承**B**类，以保护继承方式继承**C**类。**D**按不同的继承方式的规则继承**A,B,C**的属性，确定各基类的成员在派生类中的访问权限。

11.6.2 多重继承派生类的构造函数

多重继承派生类的构造函数形式与单继承时的构造函数形式基本相同，只是在初始表中包含多个基类构造函数。如

派生类构造函数名(总参数表列): 基类1构造函数(参数表列), 基类2构造函数(参数表列), 基类3构造函数(参数表列)

{派生类中新增数成员据成员初始化语句}

各基类的排列顺序任意。派生类构造函数的执行顺序同样为: 先调用基类的构造函数，再执行派生类构造函数的函数体。调用基类构造函数的顺序是按照声明派生类时基类出现的顺序。

例11.8 声明一个教师(Teacher)类和一个学生(Student)类，用多重继承的方式声明一个研究生(Graduate)派生类。教师类中包括数据成员name(姓名)、age(年龄)、title(职称)。学生类中包括数据成员name1(姓名)、age(性别)、score(成绩)。在定义派生类对象时给出初始化的数据，然后输出这些数据。

```
#include <iostream>
#include <string>
using namespace std;
class Teacher//声明类Teacher(教师)
 {public:                              //公用部分
   Teacher(string nam,int a, string t)      //构造函数
    {name=nam;
     age=a;
     title=t;}
   void display( )                          //输出教师有关数据
     {cout<<"name:"<<name<<endl;
```

```
    cout<<"age"<<age<<endl;
    cout<<"title:"<<title<<endl;
   }
 protected:                          //保护部分
  string name;
  int age;
  string title;                      //职称
};

class Student                         //定义类Student(学生)
 {public:
  Student(char nam[],char s,float sco)
   {strcpy(name1,nam);
    sex=s;
    score=sco;}                      //构造函数
  void display1( )                    //输出学生有关数据
   {cout<<"name:"<<name1<<endl;
    cout<<"sex:"<<sex<<endl;
    cout<<"score:"<<score<<endl;
   }
 protected:                          //保护部分
  string name1;
  char sex;
```

```
  float score;                    //成绩
};

class Graduate:public Teacher,public Student   //声明多重继承的派生类
Graduate
{public:
  Graduate(string nam,int a,char s, string t,float sco,float w):
      Teacher(nam,a,t),Student(nam,s,sco),wage(w) { }
  void show( )                              /输出研究生的有关数据
   {cout<<"name:"<<name<<endl;
    cout<<"age:"<<age<<endl;
    cout<<"sex:"<<sex<<endl;
    cout<<"score:"<<score<<endl;
    cout<<"title:"<<title<<endl;
    cout<<"wages:"<<wage<<endl;
    }
 private:
   float wage;                  //工资
};

int main( )
 {Graduate grad1("Wang-li",24,'f',"assistant",89.5,1234.5);
```

```
  grad1.show( );
  return 0;
}
```

程序运行结果如下：

```
name: Wang-li
age: 24
sex:f
score: 89.5
title: assistance
wages: 1234.5
```

在两个基类中分别用**name**和**name1**来代表姓名，其实这是同一个人的名字，从**Graduate**类的构造函数中可以看到总参数表中的参数**nam**分别传递给两个基类的构造函数，作为基类构造函数的实参。

解决这个问题有一个好方法：在两个基类中可以都使用同一个数据成员名**name**，而在**show**函数中引用数据成员时指明其作用域，如

```
cout<<"name:"<<Teacher::name<<endl;
```

这就是惟一的，不致引起二义性，能通过编译，正常运行。

通过这个程序还可以发现一个问题：在多重继承时，从不同的基类中会继承一些重复的数据。如果有多个基类，问题会更突出。在设计派生类时要细致考虑其数据成员，尽量减少数据冗余。

11.6.3 多重继承引起的二义性问题

多重继承可以反映现实生活中的情况，能够有效地处理一些较复杂的问题，使编写程序具有灵活性，但是多重继承也引起了一些值得注意的问题，它增加了程序的复杂度，使程序的编写和维护变得相对困难，容易出错。其中最常见的问题就是继承的成员同名而产生的二义性**(ambiguous)**问题。

在上一节中已经初步地接触到这个问题了。现在作进一步的讨论。

如果类**A**和类**B**中都有成员函数**display**和数据成员**a**，类**C**是类**A**和类**B**的直接派生类。分别讨论下列**3**种情况**:**

(1) 两个基类有同名成员。如图**11.14**所示。

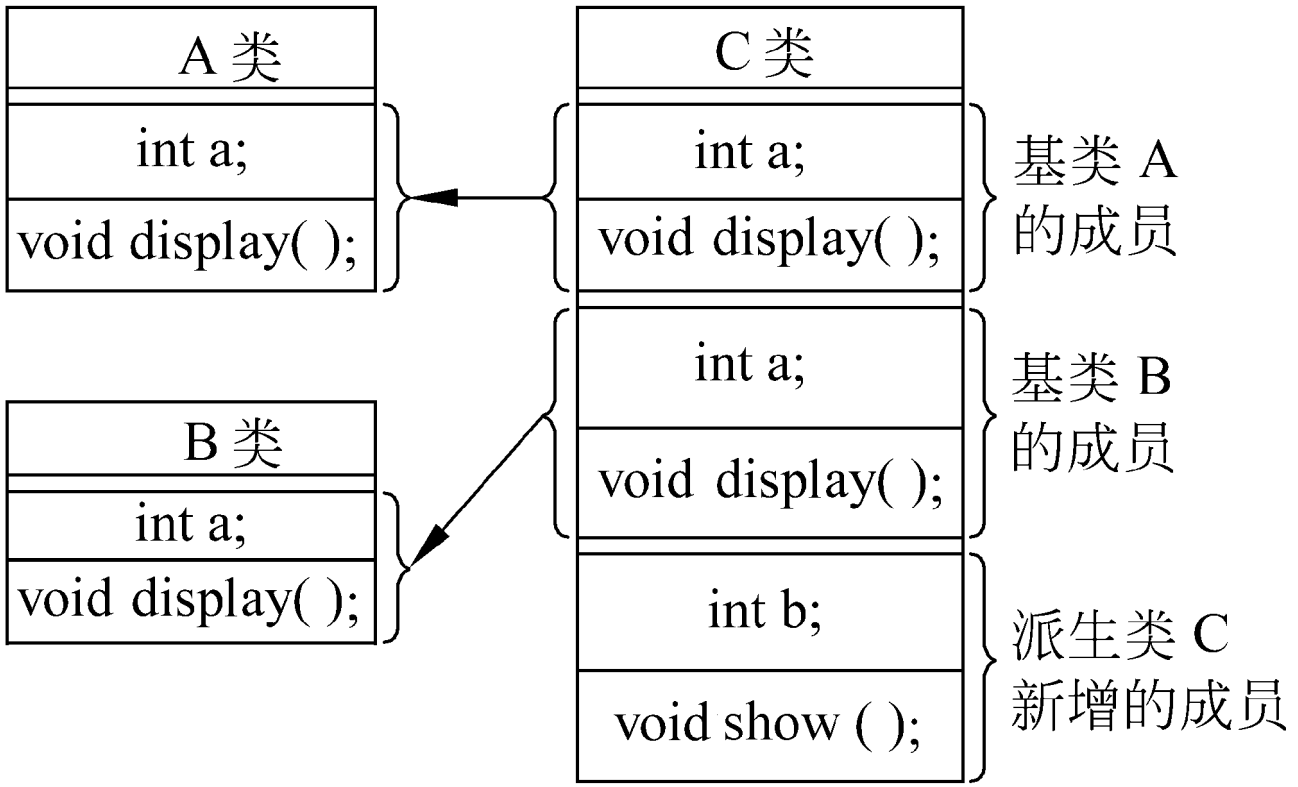


图**11.14**

```
class A
{public:
  int a;
void display( );
};
class B
{public:
  int a;
  void display( );
};
class C :public A,public B
{public :
  int b;
  void show();
};
```

如果在**main**函数中定义**C**类对象**c1**，并调用数据成员**a**和成员函数**display:**

```
C c1;
c1.a=3;
c1.display();
```

由于基类**A**和基类**B**都有数据成员**a**和成员函数**display**，编译系统无法判别要访问的是哪一基类的成员，因此，程序编译出错。可以用基类名来限定：

```
c1.A::a=3;//引用c1对象中的基类A的数据成员a
c1.A::display();          //调用c1对象中的基类A的成员函数display
```

如果是在派生类**C**中通过派生类成员函数**show**访问基类**A**的**display**和**a**，可以不必写对象名而直接写

```
A::a=3;//指当前对象
A::display( );
```

如同上一节最后所介绍的那样。为清楚起见，图**11.14**应改用图**11.15**的形式表示。

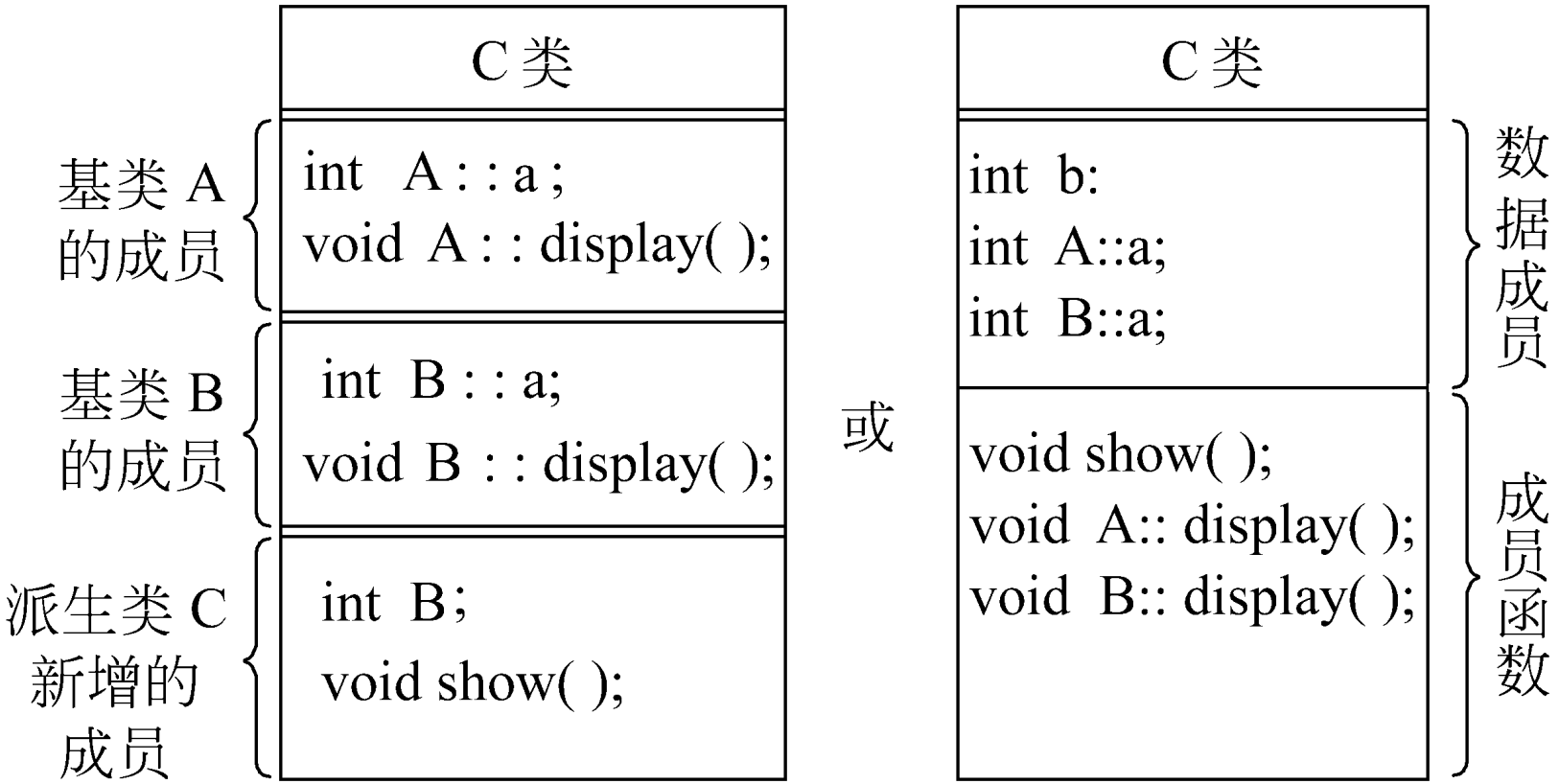

C 类
基类 A 的成员
int A : : a ;
void A : : display( );
基类 B 的成员
int B : : a;
void B : : display( );
派生类 C 新增的成员
int B;
void show( );
或
C 类
int b:
int A::a;
int B::a;
void show( );
void A:: display( );
void B:: display( );
数据成员
成员函数

图11.15

(2) 两个基类和派生类三者都有同名成员。将上面的**C**类声明改为

```
class C :public A,public B
{int a;
  void display();
};
```

如图**11.16**所示。即有**3**个**a**，**3**个**display**函数。

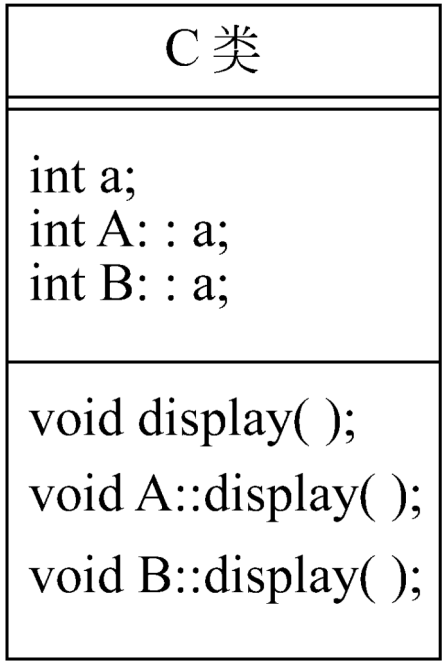


图**11.16**

如果在**main**函数中定义**C**类对象**c1**，并调用数据成员**a**和成员函数**display:**

```
C c1;
c1.a=3;
c1.display( );
```

程序能通过编译，也可正常运行。访问的是派生类**C**中的成员。规则是**:** 基类的同名成员在派生类中被屏蔽，成为“不可见”的，或者说，派生类新增加的同名成员覆盖了基类中的同名成员。因此如果在定义派生类对象的模块中通过对象名访问同名的成员，则访问的是派生类的成员。请注意**:** 不同的成员函数，只有在函数名和参数个数相同、类型相匹配的情况下才发生同名覆盖，如果只有函数名相同而参数不同，不会发生同名覆盖，而属于函数重载。

要在派生类外访问基类A中的成员，应指明作用域A，写成以下形式：

```
c1.A::a=3;//表示是派生类对象c1中的基类A中的数据成员a
c1.A::display();    //表示是派生类对象c1中的基类A中的成员函数display
```

(3) 如果类A和类B是从同一个基类派生的，如图11.18所示。

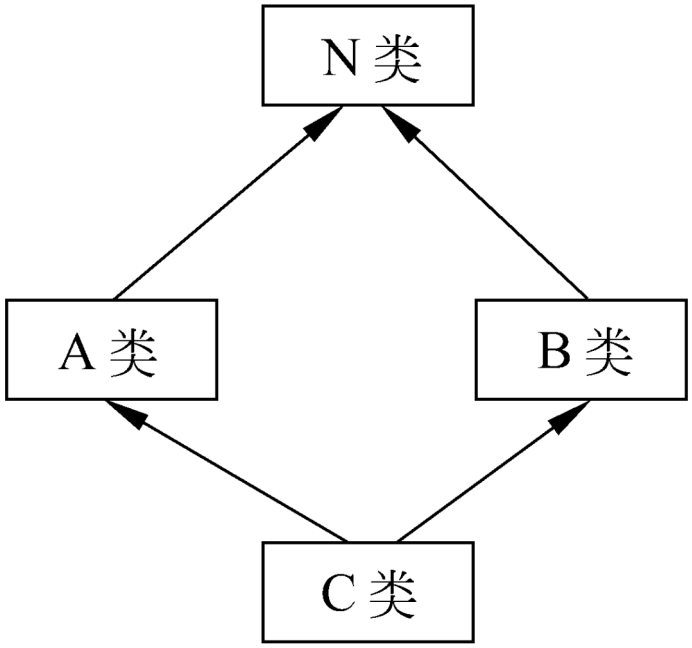


图11.18

```
class N
{public:
int a;
void display(){cout<<"A::a="<<a<<endl;}
};
class A:public N
{public:
int a1;
};
class B:public N
{public:
int a2;
 };
class C :public A,public B
{public :
int a3;
void show( ){cout<<"a3="<<a3<<endl;}
};
int main( )
{C c1;//定义C类对象c1
 ┆
}
```

图11.19和图11.20表示了派生类C中成员的情况。

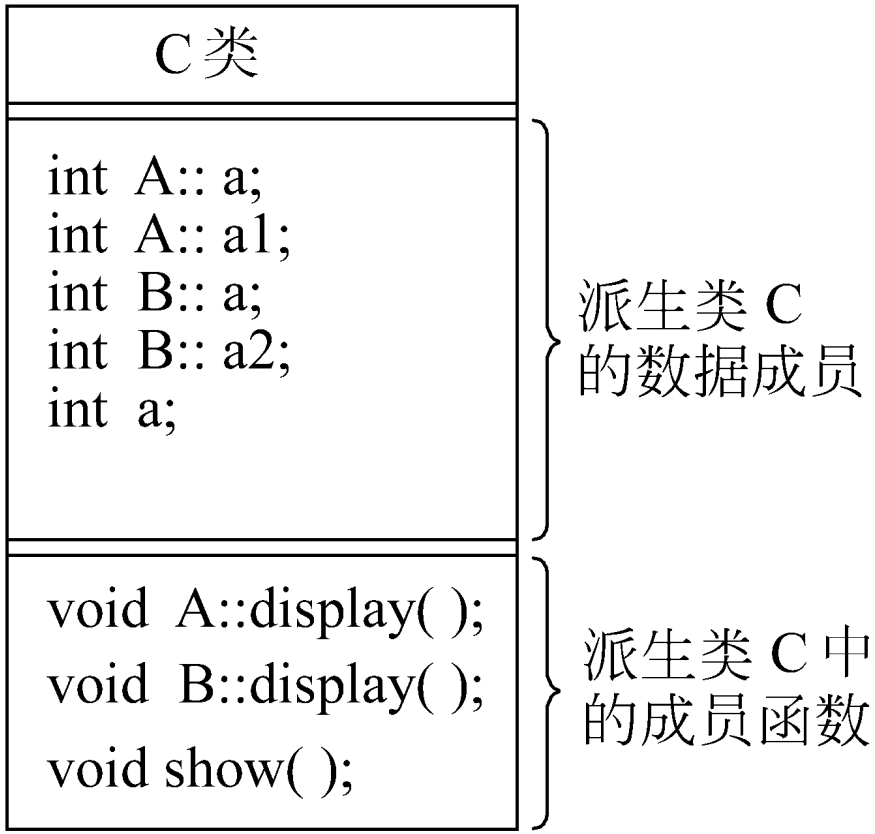


图11.19

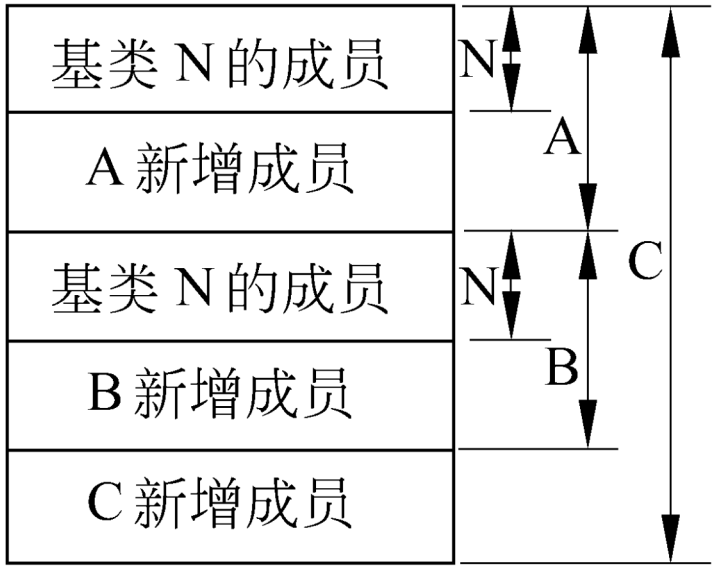


图11. 20

怎样才能访问类**A**中从基类**N**继承下来的成员呢？显然不能用

c1.a=3;c1.display( );或**c1.N::a=3; c1.N::display();**

因为这样依然无法区别是类**A**中从基类**N**继承下来的成员，还是类**B**中从基类**N**继承下来的成员。应当通过类**N**的直接派生类名来指出要访问的是类**N**的哪一个派生类中的基类成员。如

c1.A::a=3; c1.A::display();//要访问的是类**N**的派生类**A**中的基类成员

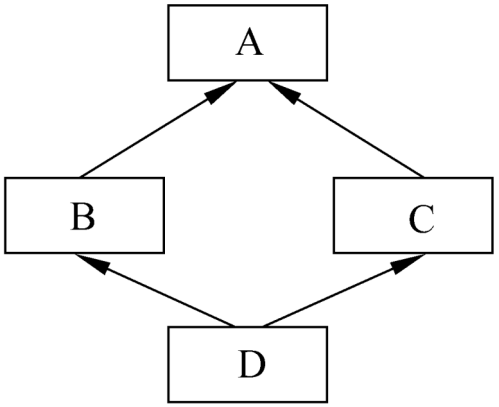
11.6.4 虚基类

1. 虚基类的作用

从上面的介绍可知: 如果一个派生类有多个直接基类，而这些直接基类又有一个共同的基类，则在最终的派生类中会保留该间接共同基类数据成员的多份同名成员。如图**11.19**和图**11.20**所示。在引用这些同名的成员时，必须在派生类对象名后增加直接基类名，以避免产生二义性，使其惟一地标识一个成员，如**c1.A::display( )**。

在一个类中保留间接共同基类的多份同名成员，这种现象是人们不希望出现的。

C++提供虚基类**(virtual base class)**的方法，使得在继承间接共同基类时只保留一份成员。


A
B
C
D

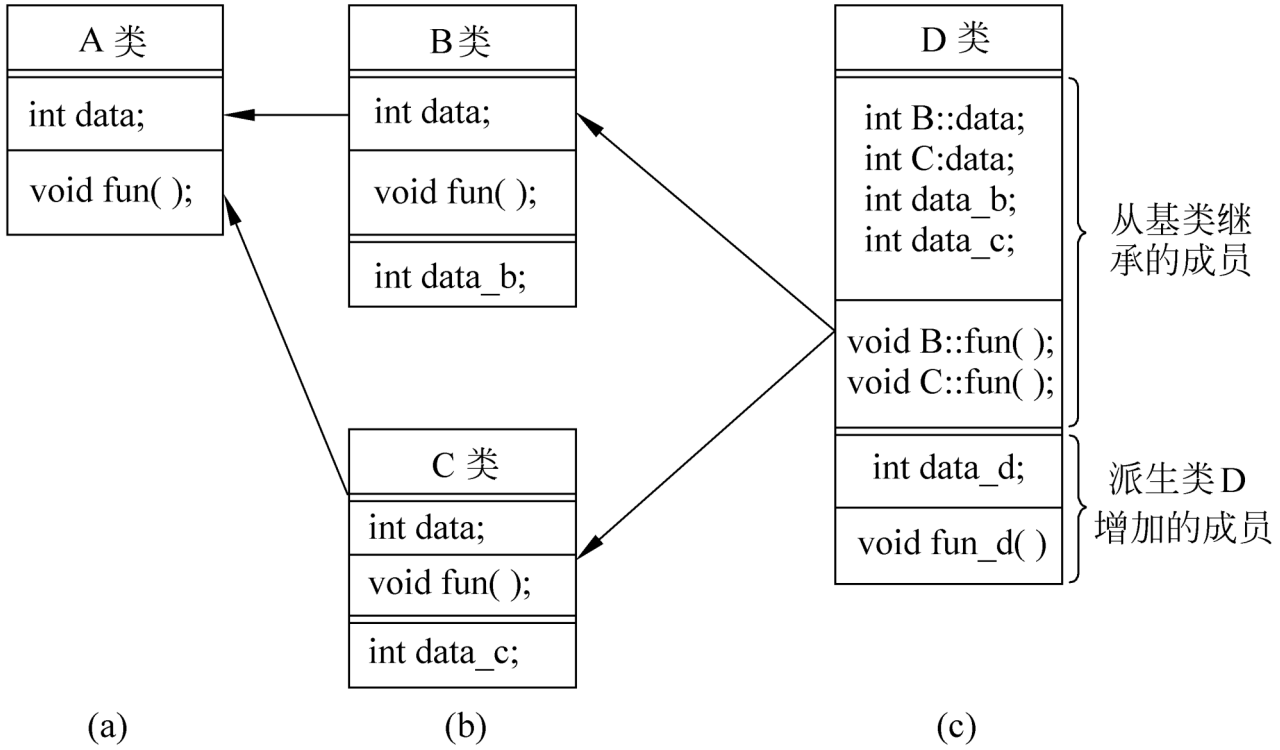

A 类
int data;
void fun( );
B类
int data;
void fun( );
int data_b;
C 类
int data;
void fun( );
int data_c;
D 类
int B::data;
int C:data;
int data_b;
int data_c;
void B::fun( );
void C::fun( );
int data_d;
void fun_d( )
从基类继承的成员
派生类D增加的成员
(a)
(b)
(c)

图11.21

图11.22

现在，将类**A**声明为虚基类，方法如下：

```
class A//声明基类A
 {…};
class B :virtual public A        //声明类B是类A的公用派生类，A是B的虚基类
 {…};
class C :virtual public A        //声明类C是类A的公用派生类，A是C的虚基类
 {…};
```

注意：虚基类并不是在声明基类时声明的，而是在声明派生类时，指定继承方式时声明的。因为一个基类可以在生成一个派生类时作为虚基类，而在生成另一个派生类时不作为虚基类。声明虚基类的一般形式为

class 派生类名**: virtual** 继承方式 基类名

经过这样的声明后，当基类通过多条派生路径被一个派生类继承时，该派生类只继承该基类一次。

在派生类**B**和**C**中作了上面的虚基类声明后，派生类**D**中的成员如图**11.23**所示。

需要注意**:** 为了保证虚基类在派生类中只继承一次，应当在该基类的所有直接派生类中声明为虚基类。否则仍然会出现对基类的多次继承。如果像图**11.24**所示的那样，在派生类**B**和**C**中将类**A**声明为虚基类，而在派生类**D**中没有将类**A**声明为虚基类，则在派生类**E**中，虽然从类**B**和**C**路径派生的部分只保留一份基类成员，但从类**D**路径派生的部分还保留一份基类成员。

D类
int data; int data_b; int data_c
void fun( );
int data_d;
void fun_d( );

图11.23

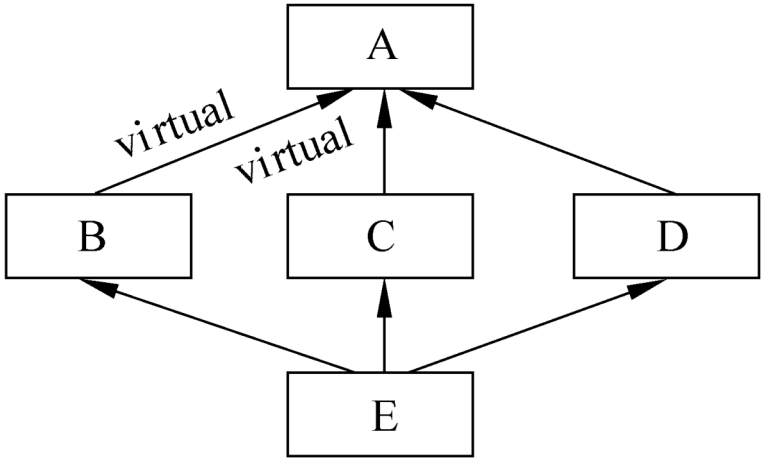


图11.24

2. 虚基类的初始化

如果在虚基类中定义了带参数的构造函数，而且没有定义默认构造函数，则在其所有派生类(包括直接派生或间接派生的派生类)中，通过构造函数的初始化表对虚基类进行初始化。例如

```
class A//定义基类A
 {A(int i){ }                    //基类构造函数，有一个参数
…};
class B :virtual public A        //A作为B的虚基类
 {B(int n):A(n){ }               //B类构造函数，在初始化表中对虚基类初始化
…};
class C :virtual public A        //A作为C的虚基类
 {C(int n):A(n){ }               //C类构造函数，在初始化表中对虚基类初始化
…};
class D :public B,public C     //类D的构造函数，在初始化表中对所有基类初始化
 {D(int n):A(n),B(n),C(n){ }
…};
```

注意: 在定义类D的构造函数时，与以往使用的方法有所不同。规定: 在最后的派生类中不仅要负责对其直接基类进行初始化，还要负责对虚基类初始化。

C++编译系统只执行最后的派生类对虚基类的构造函数的调用，而忽略虚基类的其他派生类(如类B和类C) 对虚基类的构造函数的调用，这就保证了虚基类的数据成员不会被多次初始化。

3. 虚基类的简单应用举例

例11.9 在例11.8的基础上，在Teacher类和Student类之上增加一个共同的基类Person，如图11.25所示。作为人员的一些基本数据都放在Person中，在Teacher类和Student类中再增加一些必要的数据。

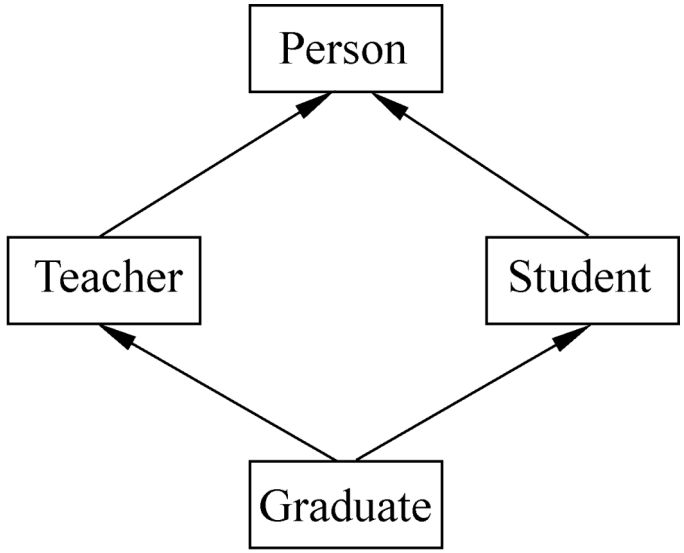


图11.25

```
#include <iostream>
#include <string>
using namespace std;
//声明公共基类Person
class Person
{public:
Person(string nam,char s,int a)//构造函数
{name=nam;sex=s;age=a;}
protected:                              //保护成员
  string name;
  char sex;
  int age;
};

//声明Person的直接派生类Teacher
class Teacher:virtual public Person          //声明Person为公用继承的虚基类
{public:
  Teacher(string nam,char s,int a, string t):Person(nam,s,a)//构造函数
   {title=t;
   }
protected:                              //保护成员
   string title;                        //职称
```

```
};

//声明Person的直接派生类Student
class Student:virtual public Person              //声明Person为公用继承的虚基类
{public:
Student(string nam,char s,int a,float sco)      //构造函数
     :Person(nam,s,a),score(sco){ }              //初始化表
protected:                                       //保护成员
   float score;                                  //成绩
 };

//声明多重继承的派生类Graduate
class Graduate:public Teacher,public Student   //Teacher和Student为直接基类
{public:
Graduate(string nam,char s,int a, string t,float sco,float w)//构造函数
:Person(nam,s,a),Teacher(nam,s,a,t),Student(nam,s,a,sco),wage(w){}
//初始化表
   void show( )                                  /输出研究生的有关数据
   {cout<<"name:"<<name<<endl;
    cout<<"age:"<<age<<endl;
    cout<<"sex:"<<sex<<endl;
    cout<<"score:"<<score<<endl;
```

```
    cout<<"title:"<<title<<endl;
    cout<<"wages:"<<wage<<endl;
    }
private:
   float wage;                  //工资
 };

//主函数
int main( )
{Graduate grad1("Wang-li",'f',24,"assistant",89.5,1234.5);
grad1.show( );
return 0;
}
```

运行结果为

```
name: Wang-li
age:24
sex:f
score:89.5
title:assistant
wages:1234.5
```

可以看到: 使用多重继承时要十分小心，经常会出现二义性问题。许多专业人员认为: 不要提倡在程序中使用多重继承，只有在比较简单和不易出现二义性的情况或实在必要时才使用多重继承，能用单一继承解决的问题就不要使用多重继承。也是由于这个原因，有些面向对象的程序设计语言(如**Java，Smalltalk)**并不支持多重继承。

11.7 基类与派生类的转换

只有公用派生类才是基类真正的子类型，它完整地继承了基类的功能。

基类与派生类对象之间有赋值兼容关系，由于派生类中包含从基类继承的成员，因此可以将派生类的值赋给基类对象，在用到基类对象的时候可以用其子类对象代替。具体表现在以下几个方面：

(1) 派生类对象可以向基类对象赋值。

可以用子类(即公用派生类)对象对其基类对象赋值。如

```
A a1;          //定义基类A对象a1
B b1;          //定义类A的公用派生类B的对象b1
a1=b1;         //用派生类B对象b1对基类对象a1赋值
```

在赋值时舍弃派生类自己的成员。如图**11.26**示意。实际上，所谓赋值只是对数据成员赋值，对成员函数不存在赋值问题。

请注意**:** 赋值后不能企图通过对象**a1**去访问派生类对象**b1**的成员，因为**b1**的成员与**a1**的成员是不同的。假设**age**是派生类**B**中增加的公用数据成员，分析下面的用法**:**

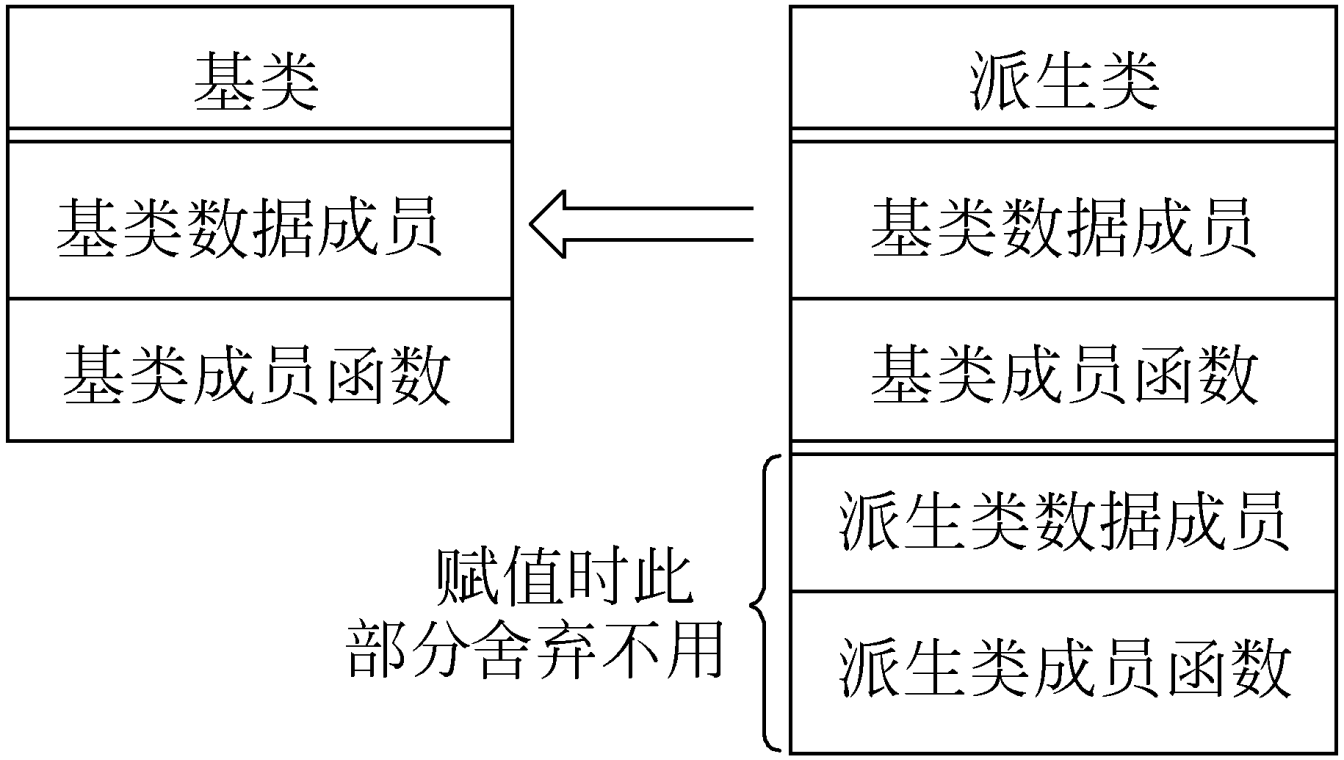

基类
基类数据成员
基类成员函数
派生类
基类数据成员
基类成员函数
派生类数据成员
派生类成员函数
赋值时此
部分舍弃不用

图11.26

a1.age=23;//错误，**a1**中不包含派生类中增加的成员

b1.age=21; //正确，**b1**中包含派生类中增加的成员

应当注意，子类型关系是单向的、不可逆的。**B**是**A**的子类型，不能说**A**是**B**的子类型。只能用子类对象对其基类对象赋值，而不能用基类对象对其子类对象赋值，理由是显然的，因为基类对象不包含派生类的成员，无法对派生类的成员赋值。同理，同一基类的不同派生类对象之间也不能赋值。

(2) 派生类对象可以替代基类对象向基类对象的引用进行赋值或初始化。

如已定义了基类**A**对象**a1**，可以定义**a1**的引用变量**:**

```
A a1;          //定义基类A对象a1
B b1;          //定义公用派生类B对象b1
A& r=a1;       //定义基类A对象的引用变量r，并用a1对其初始化
```

这时，引用变量**r**是**a1**的别名，**r**和**a1**共享同一段存储单元。也可以用子类对象初始化引用变量**r**，将上面最后一行改为

```
A& r=b1;//定义基类A对象的引用变量r，并用派生类B对象b1
//对其初始化
```

或者保留上面第**3**行**“A& r=a1;”**，而对**r**重新赋值**:**

```
r=b1;//用派生类B对象b1对a1的引用变量r赋值
```

注意: 此时r并不是b1的别名，也不与b1共享同一段存储单元。它只是b1中基类部分的别名，r与b1中基类部分共享同一段存储单元，r与b1具有相同的起始地址。见图11.27。

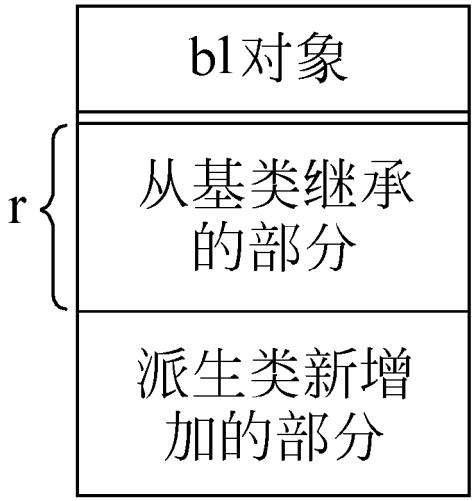


图11.27

(3) 如果函数的参数是基类对象或基类对象的引用，相应的实参可以用子类对象。

如有一函数**fun:**

```
void fun(A& r)//形参是类A的对象的引用变量
 {cout<<r.num<<endl;}          //输出该引用变量的数据成员num
```

函数的形参是类**A**的对象的引用变量，本来实参应该为**A**类的对象。由于子类对象与派生类对象赋值兼容，派生类对象能自动转换类型，在调用**fun**函数时可以用派生类**B**的对象**b1**作实参**:**

```
fun(b1);
```

输出类**B**的对象**b1**的基类数据成员**num**的值。

与前相同，在**fun**函数中只能输出派生类中基类成员的值。

(4) 派生类对象的地址可以赋给指向基类对象的指针变量，也就是说，指向基类对象的指针变量也可以指向派生类对象。

例**11.10** 定义一个基类**Student**(学生)，再定义**Student**类的公用派生类**Graduate**(研究生)，用指向基类对象的指针输出数据。

本例主要是说明用指向基类对象的指针指向派生类对象，为了减少程序长度，在每个类中只设很少成员。学生类只设**num**(学号),**name**(名字)和**score**(成绩)**3**个数据成员，**Graduate**类只增加一个数据成员**pay**(工资)。程序如下**:**

```
#include <iostream>
#include <string>
using namespace std;
class Student//声明Student类
{public:
   Student(int, string,float);                    //声明构造函数
   void display( );                               //声明输出函数
private:
   int num;
   string name;
   float score;
};

Student::Student(int n, string nam,float s)        //定义构造函数
{num=n;
  name=nam;
  score=s;
}

void Student::display( )                           //定义输出函数
{cout<<endl<<"num:"<<num<<endl;
  cout<<"name:"<<name<<endl;
```

```
  cout<<"score:"<<score<<endl;
}

class Graduate:public Student                    //声明公用派生类Graduate
{public:
  Graduate(int, string ,float,float);            //声明构造函数
  void display( );                               //声明输出函数
private:
  float pay;                                     //工资
};

Graduate::Graduate(int n, string nam,float s,float p):Student(n,nam,s),pay(p){ }
                                  //定义构造函数
void Graduate::display()                         //定义输出函数
{Student::display();                             //调用Student类的display函数
  cout<<"pay="<<pay<<endl;
}

int main()
 {Student stud1(1001,"Li",87.5);                 //定义Student类对象stud1
  Graduate grad1(2001,"Wang",98.5,563.5);        //定义Graduate类对象grad1
```

```
 Student *pt=&stud1;//定义指向Student类对象的指针并指向stud1
 pt->display( );              //调用stud1.display函数
 pt=&grad1;                   //指针指向grad1
 pt->display( );              //调用grad1.display函数
}
```

很多读者会认为：在派生类中有两个同名的**display**成员函数，根据同名覆盖的规则，被调用的应当是派生类**Graduate**对象的**display**函数，在执行**Graduate::display**函数过程中调用**Student::display**函数，输出**num,name,score**，然后再输出**pay**的值。事实上这种推论是错误的，先看看程序的输出结果：

```
num:1001
name:Li
score:87.5

num:2001
name:wang
score:98.5
```

并没有输出**pay**的值。问题在于**pt**是指向**Student**类对象的指针变量，即使让它指向了**grad1**，但实际上**pt**指向的是**grad1**中从基类继承的部分。通过指向基类对象的指针，只能访问派生类中的基类成员，而不能访问派生类增加的成员。所以**pt->display()**调用的不是派生类**Graduate**对象所增加的**display**函数，而是基类的**display**函数，所以只输出研究生**grad1**的**num,name,score3**个数据。如果想通过指针输出研究生**grad1**的**pay**，可以另设一个指向派生类对象的指针变量**ptr**，使它指向**grad1**，然后用**ptr->display()**调用派生类对象的**display**函数。但这不大方便。

通过本例可以看到:用指向基类对象的指针变量指向子类对象是合法的、安全的，不会出现编译上的错误。但在应用上却不能完全满足人们的希望，人们有时希望通过使用基类指针能够调用基类和子类对象的成员。在下一章就要解决这个问题。办法是使用虚函数和多态性。

11.8 继承与组合

在本章**11.5.2**节中已经说明: 在一个类中可以用类对象作为数据成员，即子对象。在**11.5.2**节中的例**11.6**中，对象成员的类型是基类。实际上，对象成员的类型可以是本派生类的基类，也可以是另外一个已定义的类。在一个类中以另一个类的对象作为数据成员的，称为类的组合**(composition)**。

例如，声明**Professor**(教授)类是**Teacher**(教师)类的派生类，另有一个类**BirthDate**(生日)，包含**year,month,day**等数据成员。可以将教授生日的信息加入到**Professor**类的声明中。如

```
class Teacher//教师类
{public:
 ┆
private:
   int num;
   string name;
   char sex;
  };

class BirthDate          //生日类
{ public:
 ┆
private:
int year;
int month;
int day;
};

class Professor:public Teacher       //教授类
{public:
 ┆
  private:
```

```
  BirthDate birthday;          //BirthDate类的对象作为数据成员
};
```

类的组合和继承一样，是软件重用的重要方式。组合和继承都是有效地利用已有类的资源。但二者的概念和用法不同。

Professor类通过继承，从**Teacher**类得到了**num,name,age,sex**等数据成员，通过组合，从**BirthDate**类得到了**year,month,day**等数据成员。继承是纵向的，组合是横向的。

如果定义了**Professor**对象**prof1**，显然**prof1**包含了生日的信息。通过这种方法有效地组织和利用现有的类，大大减少了工作量。

如果有

void fun1(Teacher &);

void fun2(BirthDate &);

在**main**函数中调用这两个函数:

fun1(prof1);//正确，形参为**Teacher**类对象的引用，实参为**Teacher**类的子类对象，与之赋值兼容

fun2(prof1.birthday);//正确，实参与形参类型相同，都是**BirthDate**类对象

fun2(prof1);//错误，形参要求是**BirthDate**类对象，而**prof1**是**Professor**类型，不匹配

对象成员的初始化的方法已在**11.5.2**节中作过介绍。如果修改了成员类的部分内容，只要成员类的公用接口(如头文件名)不变，如无必要，组合类可以不修改。但组合类需要重新编译。

11.9 继承在软件开发中的重要意义

有了继承，使软件的重用成为可能。继承是C++和C的最重要的区别之一。

由于C++提供了继承的机制，这就吸引了许多厂商开发各类实用的类库。用户将它们作为基类去建立适合于自己的类(即派生类)，并在此基础上设计自己的应用程序。类库的出现使得软件的重用更加方便，现在有一些类库是随着C++编译系统卖给用户的。读者不要认为类库是C++编译系统的一部分。不同的C++编译系统提供的由不同厂商开发的类库一般是不同的。

对类库中类的声明一般放在头文件中，类的实现(函数的定义部分)是单独编译的，以目标代码形式存放在系统某一目录下。用户使用类库时，不需要了解源代码，但必须知道头文件的使用方法和怎样去连接这些目标代码(在哪个子目录下)，以便源程序在编译后与之连接。

由于基类是单独编译的，在程序编译时只需对派生类新增的功能进行编译，这就大大提高了调试程序的效率。如果在必要时修改了基类，只要基类的公用接口不变，派生类不必修改，但基类需要重新编译，派生类也必须重新编译，否则不起作用。

人们为什么这么看重继承，要求在软件开发中使用继承机制，尽可能地通过继承建立一批新的类？为什么不是将已有的类加以修改，使之满足自己应用的要求呢？

(1) 有许多基类是被程序的其他部分或其他程序使用的，这些程序要求保留原有的基类不受破坏。

(2) 用户往往得不到基类的源代码。

(3) 在类库中，一个基类可能已被指定与用户所需的多种组件建立了某种关系，因此在类库中的基类是不容许修改的。

(4) 实际上，许多基类并不是从已有的其他程序中选取来的，而是专门作为基类设计的。

(5) 在面向对象程序设计中，需要设计类的层次结构，从最初的抽象类出发，每一层派生类的建立都逐步地向着目标的具体实现前进。