C++高级语言程序设计

王晨宇

北京邮电大学网络空间安全学院

第7章 继承和派生

- 通过继承已有类的部分或全部成员，创建新类的过程称为派生。
- 继承既简化程序设计，显著提高软件的重用性，又使软件维护更容易。

7.1 继承
7.2 初始化基类成员
7.3 冲突、支配规则和赋值兼容性
7.4 虚基类
7.5 虚函数

7.1 继承

- 在定义类A时，若使用了已有类B，则称类A继承了类B，并称类B为基类或父类，称类A为派生类或子类。
- 一个派生类又可作为另一个类的基类，一个基类可派生出若干个派生类，这样就构成类树或类族。
- 继承分为
  - 单一继承：一个派生类仅有一个基类；
  - 多重继承：一个派生类有两个或两个以上的基类。

- 举例：见图13-1，图中的箭头是从派生类指向基类。
  - 在校人员类：描述在校人员的共性信息(如姓名、年龄、身高和性别等)。
  - 学生类：单一继承“在校人员类”，并增加描述学生的个性信息(如学号、所学专业和课程等)。
  - 职工类：单一继承“在校人员类”，并增加描述职工的个性信息(如工资、工作部门、职工号、所教课程等)。
  - 在职研究生类：多重继承“学生类”和“职工类”。
  - 类族：在校人员类、学生类、职工类、在职研究生类。

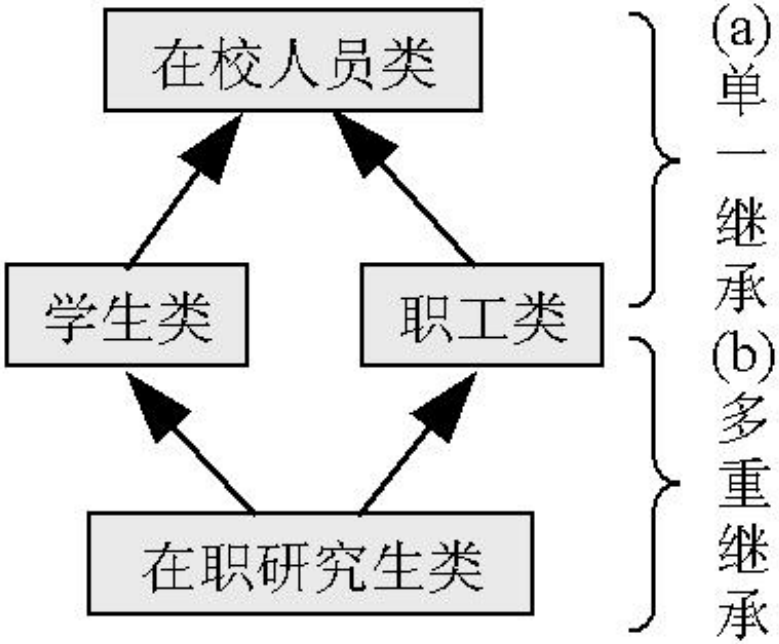


图 13.1　继承

单一继承

- 格式:

```
class ClassName:<Access> BaseClassName
{
...   //派生类中新增成员，可为空
};
```

已定义的类名

派生类的类名

规定基类成员在派生类中的访问权限，取public、private和protected三者之一或缺省。

- 当Access为public时，为公有派生；当Access为private时，为私有派生；当Access为protected时，为保护派生。
- Access缺省时，类为private；结构体为public。

公有派生

- 基类成员在公有派生类中保持原有访问权限。
  - 基类的public成员，在派生类中仍为public成员。
  - 基类的private成员，在派生类中仍为private成员。注意，派生类不能直接使用基类中的私有成员。
  - 基类的protected成员，在派生类中仍为protected成员，在派生类中可直接访问，但在派生类外，不可直接访问。
- 例7.1 公有派生。

```
#include<iostream.h>

class A
{  int x;
protected:
   int y;
```

```
public:
   int z;
   A(int a,int b,int c){ x=a;y=b;z=c; }
   void Setx(int a){ x=a; }
   void Sety(int a){ y=a; }
   int Getx( ){ return x; }
   int Gety( ){ return y; }
   void ShowB( )
   {cout<<"x="<<x<<"\ty="<<y<<"\tz="<<z<<'\n';}
};

class B:public A   //公有继承基类A，派生类B
{  int Length,Width;
public:
   B(int a,int b,int c,int d,int e): A(a,b,c)
   {   Length=d;Width=e;  }
```

在派生类的构造函数中，初始化基类的成员数据：用成员初始化列表调用基类的构造函数，详见7.2节。

```
    void Show( )
    { cout<<"Length="<<Length<<"\tWidth="<<Width<<'\n';
      cout<<"x="<<Getx( )<<"\ty="<<y<<"\tz="<<z<<'\n';
    }
    int Sum( ){  return Getx( )+y+z+Length+Width; }
};

void main(void)
{   B b1(1,2,3,4,5);
    b1.ShowB( );
    b1.Show( );
    cout<<"Sum="<<b1.Sum( );
    cout<<"\ny="<<b1.Gety( );
    cout<<"\tz="<<b1.z<<'\n';
}
```

在派生类B内：直接使用基类中的保护成员y和公有成员z；但不能直接使用基类的私有成员x，例如，若将Getx( )改为x，则会出现编译错误。

在派生类B外：派生类B的对象b1直接使用基类的公有成员函数ShowB( )、Gety( )和基类的公有成员数据z。

问：能否改为b1.y?

私有派生

- 基类中公有成员和保护成员在私有派生类中均变为私有的，在派生类中仍可直接访问，但在派生类之外均不可直接访问。
- 基类中的私有成员在私有派生类中不可直接访问，当然在派生类之外，更不直接访问。

- 例7.2 私有派生。

```
#lnclude<iostream.h>

class A
{  int x;
protected:
   int y;
```

```
public:
   int z;
   A(int a,int b,int c){  x=a;y=b;z=c;  }
   void Setx(int a){  x=a;  }
   int Getx( ){  return x;  }
   int Gety( ){  return y;  }
};

class B:private A  //私有继承基类A,  派生类B
{  int Length,Width;
public:
   B(int a,int b,int c,int d,int e) : A(a,b,c)
   {  Length=d;Width=e;  }
   void Show( )
   { cout<<"Length="<<Length<<"\tWidth="<<Width<<'\n';
```

```
    cout<<"x="<<Getx( )<<"\ty="<<y<<"\tz="<<z<<'\n';
  }
  int Sum(void)
  {  return Getx( )+y+z+Length+Width; }
};

void main(void)
{  B b1(1,2,3,4,5);
   b1.Show( );
   cout<<"Sum="<<b1.Sum( )<<'\n';
}
```

在私有派生类B内：基类的公有和保护成员在派生类中均变为私有，但在派生类中仍可直接使用，如y和z。

在私有派生类B外：派生类B的对象b1只能直接访问添加的公有成员Show( )和Sum( )，而不可直接访问基类的公有成员Getx( )、Gety( )和z。

- 实际编程中，公有派生用得最普遍，私有派生用得较少，而保护派生极少使用，这里不作介绍。
- 在派生类内外，对继承基类成员的访问权限，如表7-1所示。

表7.1 公有和私有派生

派生方式	基类成员的访问权限	派生类中对基类成员的访问权限	派生类外对基类成员的访问权限
public	public	public	可访问
public	protected	protected	不可访问
public	private	不可访问	不可访问
private	public	private（可访问）	不可访问
private	protected	private（可访问）	不可访问
private	private	不可访问	不可访问

抽象类

- 抽象类：不能定义对象而只能做基类派生新类的类。
- 抽象类举例：
  - 构造函数的访问权限为protected的类。

    若定义此类的对象，则无法在对象外调用它的私有构造函数。但用此类做基类产生公有派生类时，在派生类中可调用其基类的保护成员，即在产生派生类的对象时，允许在派生类的构造函数中调用基类的构造函数。
  - 析构函数的访问权限为protected的类。

    因在撤消该类的对象时，无法在对象外调用析构函数。 但用此类做基类产生公有派生类时，在派生类中可调用其基类的保护成员，即在撤消派生类的对象时，允许在派生类的析构函数中调用基类的析构函数。
  - 含有纯虚函数的类。详见7.5节。
- 抽象类的作用：用于建立类的层次结构。详见7.5节。

多重继承

- 格式:

  class 类名:<Access> 类名1,…,<Access> 类名n
  {
  … //派生类中新增成员，可为空
  };

  其中，派生类“类名”继承了类名1～类名n的所有成员，每个基类的类名前的Access用以限定该基类中的成员在派生类中的访问权限，其规则与单一继承相同。

- 例7.3 多重继承。

```
#include<iostream.h>
```

```
class A    //描述圆：(x,y)为圆心，r为半径
{  float x,y,r;
public:
   A(float a,float b,float c){ x=a;y=b;r=c; }
   float& AccessX( ){ return x; }
   float& AccessY( ){ return y; }
   float& AccessR( ){ return r; }
   float Area( ){ return r*r*3.14159f; }
};

class B
{  float High;
public:
   B(float a){ High=a; }
   float& AccessHigh( ){ return High; }
};
```

派生类C的构造函数调用基类A和基类B的构造函数。

```
class C:public A,private B //描述圆柱体
{  float Volume;//圆柱体的体积
public:
   C(float a,float b,float c,float d):A(a,b,c),B(d)
   {  Volume=Area( )*AccessHigh( ); }
   float GetVolume( ){ return Volume; }
};
```

类C的成员函数不能直接访问类A和B的私有成员数据，只能通过基类A和B的公有成员函数间接访问。

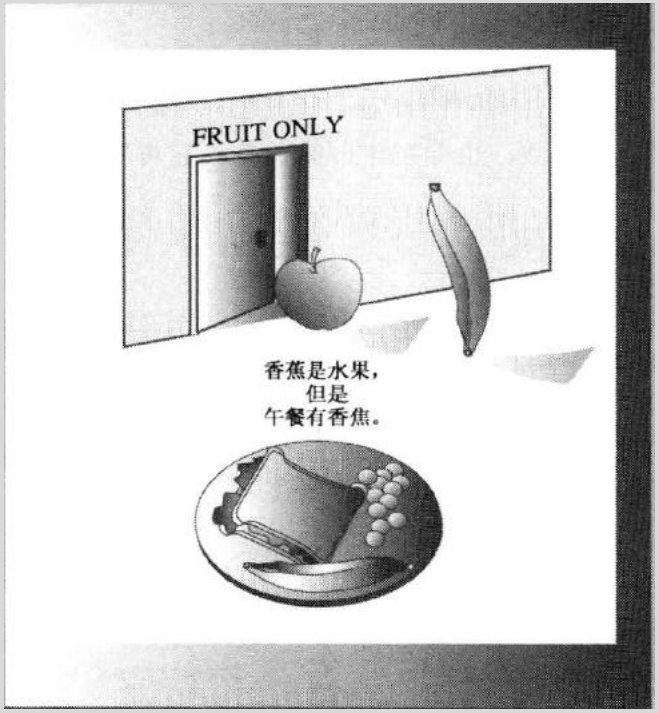
```
void main(void)
{  A a(6,8,9);  B b=23;  C c(1,2,3,4);
   cout<<"\nx="<<a.AccessX()<<"\ty="<<a.AccessY()
       <<"\nr="<<a.AccessR()<<"\tAREA="<<a.Area()
       <<"\nHigh="<<b.AccessHigh()
       <<"\nVolume="<<c.GetVolume()<<'\n';
}
```

继承：is-a关系

派生类和基类之间的特殊关系是基于 C++继承的底层模型的。实际上，C++有 3 种继承方式：公有继承、保护继承和私有继承。公有继承是最常用的方式，它建立一种 is-a 关系，即派生类对象也是一个基类对象，可以对基类对象执行的任何操作，也可以对派生类对象执行。例如，假设有一个 Fruit 类，可以保存水果的重量和热量。因为香蕉是一种特殊的水果，所以可以从 Fruit 类派生出 Banana 类。新类将继承原始类的所有数据成员，因此，Banana 对象将包含表示香蕉重量和热量的成员。新的 Banana 类还添加了专门用于香蕉的成员，这些成员通常不用于水果，例如 Banana Institute Peel Index（香蕉机构果皮索引）。因为派生类可以添加特性，所以，将这种关系称为 is-a-kind-of（是一种）关系可能更准确，但是通常使用术语 is-a。

继承：is-a关系

为阐明 is-a 关系，来看一些与该模型不符的例子。公有继承不建立 has-a 关系。例如，午餐可能包括水果，但通常午餐并不是水果。所以，不能通过从 Fruit 类派生出 Lunch 类来在午餐中添加水果。在午餐中加入水果的正确方法是将其作为一种 has-a 关系：午餐有水果。正如将在第 14 章介绍的，最容易的建模方式是，将 Fruit 对象作为 Lunch 类的数据成员（参见图 13.3）。



继承：is-a关系

公有继承不能建立 is-like-a 关系，也就是说，它不采用明喻。人们通常说律师就像鲨鱼，但律师并不是鲨鱼。例如，鲨鱼可以在水下生活。所以，不应从 Shark 类派生出 Lawyer 类。继承可以在基类的基础上添加属性，但不能删除基类的属性。在有些情况下，可以设计一个包含共有特征的类，然后以 is-a 或 has-a 关系，在这个类的基础上定义相关的类。

公有继承不建立 is-implemented-as-a（作为……来实现）关系。例如，可以使用数组来实现栈，但从 Array 类派生出 Stack 类是不合适的，因为栈不是数组。例如，数组索引不是栈的属性。另外，可以以其他方式实现栈，如链表。正确的方法是，通过让栈包含一个私有 Array 对象成员来隐藏数组实现。

公有继承不建立 uses-a 关系。例如，计算机可以使用激光打印机，但从 Computer 类派生出 Printer 类（或反过来）是没有意义的。然而，可以使用友元函数或类来处理 Printer 对象和 Computer 对象之间的通信。

在 C++中，完全可以使用公有继承来建立 has-a、is-implemented-as-a 或 uses-a 关系；然而，这样做通常会导致编程方面的问题。因此，还是坚持使用 is-a 关系吧。

7.2 初始化基类成员

- 派生类的构造函数完成：
  - 初始化派生类中的基类成员：调用基类的构造函数
  - 初始化派生类中新增成员数据：调用派生类的构造函数
- 派生类的构造函数的格式：

其中，ClassName是派生类名；B1、…、Bn是基类构造函数名；a是形参表，a1、…、an是实参表。

```
ClassName::ClassName(a) : B1(a1),…,Bn(an)
{ … }
```

函数体初始化派生类中的其他成员数据。

- 初始化成员列表：冒号后列举基类成员的构造函数和对象成员的构造函数，项与项之间用逗号分隔。
- 初始化成员列表用基类名调用基类的构造函数，可省略基类成员的缺省构造函数(即无参或所有参数都带缺省值的构造函数)。

- 例7.4 输出派生类中构造函数与析构函数的调用关系。

```
#include<iostream.h>

class B1
{  int x;
public:
   B1(int a){ x=a; cout<<"基类B1的构造函数!\n"; }
   ~B1(){ cout<<"基类B1的析构函数!\n"; }
};

class B2
{  int y;
public:
   B2(int a=20){ y=a; cout<<"基类B2的构造函数!\n"; }
   ~B2(){ cout<<"基类B2的析构函数!\n"; }
};
```

基类构造函数的调用顺序与继承基类的顺序有关。

基类构造函数的调用顺序与其在初始化成员列表中的顺序无关。

```
class D : public B1,public B2
{ int z;
public:
  D(int a,int b) : B1(a),B2(20)
  { z=b; cout<<"派生类D的构造函数!\n";}
  ~D(){ cout<<"派生类D的析构函数!\n";}
};
void main(void){  D d(100,200); }
```

可将B2(20)省略或改为B2()，因基类B2带默认值为20的缺省构造函数。

- 说明派生类的对象：先调用各基类的构造函数，后执行派生类的构造函数。若某个基类仍是派生类，则这种调用基类构造函数的过程递归进行。
- 撤消派生类的对象：析构函数的调用顺序正好与构造函数的顺序相反。

程序运行结果：

```
基类B1的构造函数!
基类B2的构造函数!
派生类D的构造函数!
派生类D的析构函数!
基类B2的析构函数!
基类B1的析构函数!
```

- 派生类含对象成员：其构造函数的初始化成员列表既要列举基类成员的构造函数，又要列举对象成员的构造函数。
- 例7.5 派生类中包含对象成员。

```
#include<iostream.h>

class B1
{  int x;
public:
   B1(int a){ x=a; cout<<"基类B1的构造函数!\n"; }
   ~B1(){ cout<<"基类B1的析构函数!\n"; }
};

class B2
{  int y;
public:
```

```
    B2(int a){ y=a; cout<<"基类B2的构造函数!\n"; }
    ~B2(){ cout<<"基类B2的析构函数!\n"; }
};
class D:public B1,public B2
{  int z;
   B1 b1,b2;
public:
   D(int a,int b):B1(a),B2(20),b1(200),b2(a+b)
   {  z=b; cout<<"派生类D的构造函数!\n"; }
   ~D(){ cout<<"派生类D的析构函数!\n"; }
};

void main(void){ D d(100,200); }
```

对象成员的构造函数的调用顺序与对象成员的说明顺序有关。

基类成员的初始化必须使用基类名

- 对象成员的初始化必须使用对象名。
- 对象成员的构造函数的调用顺序与其在初始化成员列表中的顺序无关。

程序运行结果：

```
基类B1的构造函数!
基类B2的构造函数!
基类B1的构造函数!
基类B1的构造函数!
派生类D的构造函数!
派生类D的析构函数!
基类B1的析构函数!
基类B1的析构函数!
基类B2的析构函数!
基类B1的析构函数!
```

问题：请写出产生输出结果的基类成员名或对象成员名。

- 由例7.5程序输出结果可见，
  - 在创建派生类D的对象d时，先调用基类的构造函数，再调用对象成员的构造函数，最后执行派生类的构造函数。若有多个对象成员，则调用这些对象成员的构造函数的顺序取决于它们在派生类中说明的顺序。
  - 在派生类的构造函数的初始化成员列表中，对象成员的初始化必须使用对象名，基类成员的初始化必须使用基类名。

作业（8）

编写一个汽车类，派生货车类和客车类，再基于这两个类派生皮卡车类，体现并注释说明虚基类、虚函数和纯虚函数的功能特点

7.3 冲突、支配规则和赋值兼容性

- 冲突：在多重继承的派生类中使用不同基类中的同名成员时出现的二义性。
- 例7.7 冲突。

```
#include<iostream.h>

class A{
protected:
   int x;
public:
   void Show(){ cout<<"x="<<x<<'\n'; }
   A(int a=0){ x=a; }
};
```

```
class B{
protected:
   int x;
public:
   void Show(){ cout<<"x="<<x<<'\n'; }
   B(int a=0){ x=a; }
};

class C:public A,public B{
protected:
   int y;
public:
   int& AccessX(){ return x; }
   int& AccessY(){ return y; }
};

void main(void){  C c; c.Show(); }
```

用VC6编译时，对成员数据x有如下错误和警告：

error C2385: 'C::x' is ambiguous

warning C4385: could be the 'x' in base 'A' of class 'C'

warning C4385: or the 'x' in base 'B' of class 'C'

用VC6编译时，对成员函数Show( )有如下错误和警告：

error C2385: 'C::Show' is ambiguous

warning C4385: could be the 'Show' in base 'A' of class 'C'

warning C4385: or the 'Show' in base 'B' of class 'C'

- 解决冲突的方法：
  - 使各基类中的成员名各不相同；
  - 将基类的成员数据的访问权限说明为private，并在相应的基类中提供成员函数访问这些成员数据，但并不实用。原因是，在实际编程时，通常将基类成员数据的访问权限定义为protected，以保障类的封装性和便于派生类访问。
  - 用类名限定来指明所访问的成员。格式为：

    类名::成员名

- 例7.8 用类名限定来指明所访问的成员。

```
#include<iostream.h>

class A{
protected:
   int x;
public:
   void Show(){ cout<<"x="<<x<<'\n'; }
   A(int a=0){ x=a; }
};

class B{
protected:
   int x;
public:
   void Show(){ cout<<"x="<<x<<'\n'; }
 B(int a=0){ x=a; }
};
```

```
 class C:public A,public B{
protected:
   int y;
public:
   int& AccessAX(){ return  A:: x; }
   int& AccessBX(){ return  B:: x; }
   int& AccessY(){ return y; }
};

void main(void)
{   C c;
    c.AccessAX()=35;
    c.AccessBX()=100;
    c.AccessY()=300;
    c. A:: Show();    //调用类A中的成员函数
    c. B:: Show();    //调用类B中的成员函数
    cout<<"y="<<c.AccessY()<<'\n';
}
```

用类名限定来指明所访问的成员，避免冲突。

• 例7.9 多重继承中的冲突。

```
#include<iostream.h>

class A{
protected:
   int x;
public:
   void Show(){ cout<<"x="<<x<<'\n'; }
   A(int a=0){ x=a; }
};

class B{
protected:
   int x;
public:
   void Show(){ cout<<"x="<<x<<'\n'; }
   B(int a=0){ x=a; }
};
```

```
class C:public A,public B{
protected:
   int y;
public:
   int& AccessAX(){ return A::x; }
   int& AccessBX(){ return B::x; }
   int& AccessY(){ return y; }
};

class D:public C{
protected:
   int z;
public:
   int& AccessZ(){ return z; }
};
```

```
void main(void)
{  D d;
   d.AccessAX()=35;
   d.AccessBX()=100;
   d.AccessY()=300;

   d. C::A:: Show();

   d. C::B:: Show();

   d.AccessZ()=500;
   cout<<"y="<<d.AccessY()<<'\n';
   cout<<"z="<<d.AccessZ()<<'\n';
}
```

- 现象：用VC6编译时，指出此处有错误和警告

error C2039: 'A' : is not a member of 'C'
error C2385: 'D::Show' is ambiguous
warning C4385: could be the 'Show' in base 'A' of base 'C' of class 'D'
warning C4385: or the 'Show' in base 'B' of base 'C' of class 'D'

- 原因：作用域运算符不能连续使用。
- 改法：

d.A::Show( );

用VC6编译时，此处亦有上述问题，
最简单的解决办法是将该行改为：
d.B::Show();

支配规则

- **支配规则：**未用类名限定时，派生类定义的成员优先于基类中的同名成员，并不产生冲突。

- 例7.10 支配规则。

```
#include<iostream.h>

class A{
protected:
   int x;
public:
   void Show(){ cout<<"x="<<x<<'\n'; }
};
```

```
class C:public A{
protected:
   int x;
public:
   int& AccessX(){ return  x ;}
   int& AccessAX(){ return  A::x ;}
};

void main(void)
{   C c;
    c.AccessX()=100;
    c.AccessAX()=200;
    cout<<"C中的x="<<c.AccessX()<<'\n';
    cout<<"A中的x="<<c.AccessAX()<<'\n';
}
```

按支配规则，此处访问的是派生类C中的新增成员x。

用类名限定，访问的是基类A中的成员x。

用基类成员还是对象成员？

- 基类成员举例

```
class A{
public: float x;
    …
};

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

问题：因派生类B中含有两个直接继承来的成员x，在使用类B的对象访问基类的成员x时会产生无法避免的冲突。

- C++规定：任一基类在派生类中只能直接继承一次，以避免上述无法避免的冲突。

- 对象成员举例：若在B类中，确实需要两个类A的成员，则可用类A的两个对象来实现。例如：

```
class B{
   A a1,a2; //或A a[2];
   …
};
```

- 基类成员和对象成员，在功能上是相同的，但在使用上是有区别的：在派生类中可直接使用基类的成员(访问权限允许的话)，但要使用对象成员的成员时，必须在对象名后加上成员运算符“.”和成员名。参见例7.5。

赋值兼容规则

- 赋值兼容规则：规定派生类的对象与其基类的对象之间互相赋值的规则。设：

```
class A{
public:
  int x;
   …
};
class C:public A{
public:
  int y;
   …
};
C c1,c2;
A a,*pa;
```

则有以下规则：

– 派生类的对象可赋给基类的对象，如：

a=c1; //将c1中从类A继承的部分赋给a

– 不允许将基类的对象赋给派生类对象，如：

c2=a; //不允许

– 可将派生类对象的指针赋给基类型的指针变量，如：

pa=&c2;

– 派生类对象可以初始化基类型的引用，如：

A&ra=c1;

- 注意：在后两种情况下，使用基类的指针或引用时，只能访问从相应基类中继承来的成员，而不允许访问其他基类的成员或在派生类中增加的成员。

7.4 虚基类

- 尽管一个基类在派生类中只能直接继承一次，但并没有限制一个基类在派生类中间接继承的次数。
- 例7.12 一个基类在派生类中产生两个拷贝。

```
#include<iostream.h>
class A{
public:
   int x;
   A(int a=0){ x=a; }
};
class B:public A{
public:
   int y;
   B(int a=0,int b=0):A(b){ y=a; }
   void PB(){ cout<<"x="<<x<<"\ty="<<y<<'\n'; }
};
```

```
class C:public A{
public:
   int z;
   C(int a=0,int b=0):A(b){  z=a; }
   void PC(){  cout<<"x="<<x<<"\tz="<<z<<'\n';  }
};

class D:public B,public C{
public:
   int m;
   D(int a,int b,int d,int e,int f):B(a,b),C(d,e)
   {  m=f;  }
   void Print(void)
   {  PB();  PC();  cout<<"m="<<m<<'\n'; }
};
```

```
void main(void)
{   D d(100,200,300,400,500);
    d.Print();
}
```

注意到：

- 派生类D包含两份基类A成员。
- 两份基类A成员的成员数据x的值。
- 问题：若希望类D只包含一份基类A成员(实际编程时较常见，如图13-1所示)，则用例7.12来实现不仅多占内存，且可能产生冲突，难以保证多份拷贝中的数据一致。如何解决？

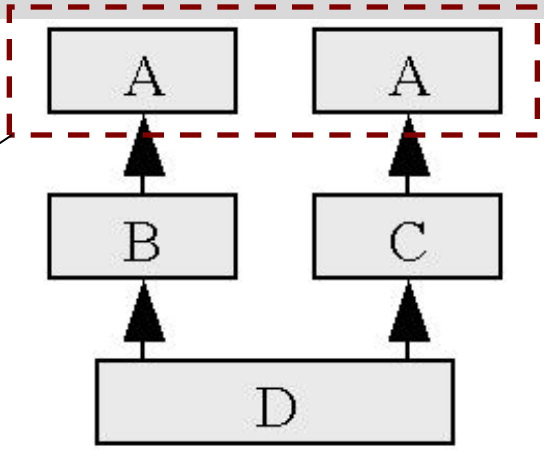


图13-2 派生类D包含两份基类A成员

程序运行结果：

x=200 y=100
x=400 z=300
m=500

- 例7.13 派生类包含两份基类成员，产生冲突。

```
#include<iostream.h>

class A{
public:
   int x;
   A(int a=0){ x=a; }
};

class B:public A{
public:
   int y;
   B(int a=0,int b=0):A(b){ y=a; }
};

class C:public A{
public:
   int z;
   C(int a=0,int b=0):A(b){ z=a; }
};
```

```
class D:public B,public C{
public:
   int m;
   D(int a,int b,int d,int e,int f):B(a,b),C(d,e)
   { m=f; }
   void Print(void)
   { cout<< x <<'\t'<<y<<'\n'; //E
     cout<< x <<'\t'<<z<<'\n'; //F
     cout<<m<<'\t';
   }
};

void main(void)
{  D d1(100,200,300,400,500);
   d1.Print();
}
```

冲突：此时编译器无法确定成员x是继承于类B，还是继承于类C。

消除：用类名限定来指明成员x源自类B或类C。即用B::x代替E行中的x，用C::x代替F行中的x。

- 虚基类：在定义派生类时，在继承的公共基类的类名前加关键字virtual，使得公共基类在派生类中只有一份拷贝。
- 虚基类的格式：

```
class ClassName:virtual <access> ClassName1
{ … };
```

或

```
class ClassName:<access> virtual ClassName1
{ … };
```

- 例7.13 定义虚基类，使派生类中只有基类的一份拷贝。

```
#include<iostream.h>

class A{
public:
  int x;
```

```
   A(int a=0){  x=a;  }          //L1
};

class B:virtual public A{
public:
   int y;
   B(int a,int b):A(b){  y=a;  }
   void PB(){  cout<<"x="<<x<<"\ty="<<y<<'\n';  }
};

class C:public virtual A{
public:
   int z;
   C(int a,int b):A(b){  z=a;  }
   void PC(){  cout<<"x="<<x<<"\tz="<<z<<'\n';  }
};
```

```
class D:public B,public C{
public:
   int m;
   D(int a,int b,int d,int e,int f):B(a,b),C(d,e)//L2
   {  m=f;  }
   void Print(void)
   {  PB(); PC(); cout<<"m="<<m<<'\n';  }
};

void main(void)
{   D d(100,200,300,400,500);
    d.Print();
    d.x=400;
    d.Print();
}
```

- 当改变成员x的值时，由基类B和C中的成员函数输出的x的值是相同的。即派生类D的对象d只有基类A的一份拷贝，如图13-3所示。
- x的初值为何为0?
  - 分析：从构造函数的调用过程看，成员x的初值应为0。
  - 原因：虚基类A在类D中只有一份拷贝，系统无法确定是由类B的构造函数还是由类C的构造函数来调用类A的构造函数。此时，系统约定，在执行类B和类C的构造函数时都不调用虚基类A的构造函数，而是在类D的构造函数中直接调用虚基类A的缺省的构造函数，即x的值置为0。

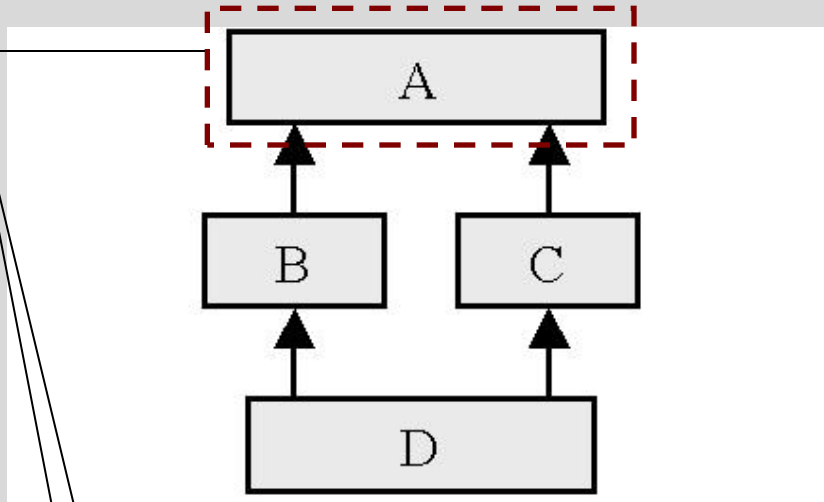


图13-3 派生类D包含一个基类A成员

程序运行结果:

```
x=0     y=100
x=0     z=300
m=500
x=400   y=100
x=400   z=300
m=500
```

- 进一步讨论：

若将L1行改为

```
A(int a){ x=a; }            //L1
```

再用VS编译时，将指出L2行有错：“error C2512: ‘A::A’ : no appropriate default constructor available”，即指出类A没有合适的缺省的构造函数可用；此时可将L2行改为：

```
D(int a,int b,int d,int e,int f)
  :B(a,b),C(d,e), A(10)     //L2
```

即在类D的构造函数的初始化成员列表中增加调用虚基类A的构造函数，则将类D中的x成员的初值置为10。

注意：用虚基类进行多重派生时，若虚基类没有缺省的构造函数，则在每一个派生类的构造函数的初始化成员列表中都应有对虚基类构造函数的调用。如上面的A(10)。

- 如果派生类继承了多个基类，基类中有虚基类和非虚基类，那么在创建该派生类的对象时，首先调用虚基类的构造函数，然后调用非虚基类的构造函数，最后调用派生类的构造函数。若虚基类又有多个，则虚基类构造函数的调用顺序取决于它们继承时的说明顺序。
- 例7.14 虚基类与非虚基类构造函数的调用顺序。

```
#include<iostream.h>

class A{
   int x;
public:
   A(int a=0){ x=a; cout<<"call A(int=0)\n";}
};

class B{
   int y;
```

```
public:
   B(int a=0){ y=a; cout<<"call B(int=0)\n";}
};

class C:public B,virtual public A{
   int z;
public:
   C(int a=0,int b=0):B(b+20),A(b)
   { z=a; cout<<"call C(int=0,int=0)\n"; }
};

void main(void)
{  C c(100,200); }
```

程序运行结果:

```
call A(int=0)
call B(int=0)
call C(int=0,int=0)
```

7.5 虚函数

- 多态性：通过调用同名函数来实现不同功能。分为：
  - 编译时的多态性：通过函数重载或运算符重载实现。重载的函数根据调用时给出的实参类型或个数，在程序编译时就可确定调用哪个函数。
  - 运行时的多态性：在程序执行前，根据函数名和参数无法确定应该调用哪个函数，必须在程序的执行过程中，根据具体的执行情况来动态确定。它通过类的继承关系和虚函数来实现，主要用来建立实用的类层次体系结构、设计通用程序。

虚函数的定义和使用

- 虚函数为类的非静态成员，定义格式为：

  virtual <type> FuncName(<ArgList>);

  其中，virtual指明该成员函数为虚函数。

- 虚函数的特性：
  - 继承性。若某类有某个虚函数，则在它的派生类中，该虚函数均保持虚函数特性。
  - 可重定义。若某类有某个虚函数，则在它的派生类中还可重定义该虚函数，此时不用virtual修饰，仍保持虚函数特性，但为了提高程序的可读性，通常再用virtual修饰。应强调，在派生类中重定义虚函数时，必须与基类的同名虚函数的参数个数、参数类型及返回值类型完全一致，否则属重载。

例7.15 虚函数。

```
#include<iostream.h>
class A{
protected:
   int x;
public:
   A(){ x=1000; }
   virtual void print(){ cout<<"x="<<x<<'\t'; }
};
class B:public A{
   int y;
public:
   B(){ y=2000; }
   void print()
   {  cout<<"y="<<y<<'\n'; }
};
```

问：若将此处的virtual省去，程序运行结果如何？

因派生类B的print()成员函数在参数类型、参数个数及函数的返回值类型方面与基类A的虚函数print()完全相同，故属于对基类A虚函数print()的重定义，即使不用virtual修饰，也具有虚函数特性。

编译时多态性： b是派生类对象，对于b.print()调用，在编译时，根据对象名和支配规则即可确定所调用的print()为派生类B中重定义的print()，与print()是否是虚函数无关。

```
void main(void)
{   A a,*pa;
    B b;
    a.print(); b.print();
    pa=&a; pa->print();
    pa=&b; pa->print();
}
```

程序运行结果：

x=1000 y=2000
x=1000 y=2000

问： 将“**pa->print();**”改为“**pa->A::print();**”，结果如何？

运行时多态性： 将派生类的对象b的指针赋给基类的指针变量pa，符合赋值兼容性规则。执行“pa->print();”，因print()为基类中的虚函数并在派生类中重定义，此时实际调用的是派生类中重定义的虚函数print()，而不是基类中的虚函数print()。

- 关于运行时多态性与虚函数的说明：
  - 使用基类类型的指针变量(或基类类型的引用)，使该指针指向派生类的对象(或该引用是派生类的对象的别名)，并通过指针(或引用)调用指针(或引用)所指对象的虚函数才能实现运行时的多态性。
  - 若派生类中没有重定义基类的虚函数时，当调用这种派生类对象的虚函数时，则调用其基类中的虚函数。
  - 不能将构造函数定义为虚函数，但通常把析构函数定义为虚函数，以便通过运行时多态性，正确释放基类及其派生类申请的动态内存。
  - 与一般成员函数相比，虚函数调用时的执行速度要慢一些。原因是，为了实现运行时多态性，在每个派生类中均要保存相应虚函数的入口地址表，函数的调用机制也是间接实现的。

虚函数的特殊性

例7.16 成员函数调用虚函数。

```
#include<iostream.h>
class A{
public:
   void f1(){  cout<<"A::f1\t"; f2(); }
   virtual void f2(){  cout<<"A::f2\t"; f3(); }
   void f3(){  cout<<"A::f3\n"; }
};
class B:public A{
public:
   void f2(){  cout<<"B::f2\t"; f3(); }
   void f3(){  cout<<"B::f3\n"; }
};
void main(void){  B b;  b.f1(); }
```

程序运行结果：

A::f1　B::f2　B::f3

- 问题：在main()函数中，执行b.f1()，即调用基类A中的f1(),然后在f1()中又调用f2()，此时调用的是类A的f2()还是类B重定义的f2()?
- 分析：类A的f1()的定义可用含this指针的等价形式表示为：

```
void A::f1()
{  cout<<"A::f1\t"; this->f2(); }
```

  - 此处this指针指向基类对象，其类型为：

```
A* const this;
```

即this指针是基类类型。

  - 此外，f2()是基类A的虚函数又在派生类B中重定义，因此，执行“this->f2();”语句必然引发运行时多态性，即调用的是B::f2()，而不是A::f2()。

例7.17 在构造函数中调用虚函数。

```
#include<iostream.h>
class A{
public:
   A(){ f(); }
   virtual void f()
   { cout<<"A::f\t"; }
};
class B:public A{
public:
   B(){ f(); }
   void f()
   { cout<<"B::f\t"; }
};
void main(void){  B b; }
```

程序运行结果:

A::f B::f

- 注意到，创建派生类的对象时，先调用基类的构造函数初始化基类成员，再调用派生类的构造函数初始化自定义成员。
- 由于在基类A的构造函数执行时，派生类B的构造函数尚未执行，因此，基类A的构造函数中所调用的虚函数f()只能是本类的f或其基类的f()，而不可能是派生类中中的f()。

虚析构函数的必要性

- 通过基类的指针删除派生类对象时，通常情况下只调用基类的析构函数
  - 但是，删除一个派生类的对象时，应该先调用派生类的析构函数，然后调用基类的析构函数。
- 解决办法：把基类的析构函数声明为virtual
  - 派生类的析构函数可以virtual不进行声明
  - 通过基类的指针删除派生类对象时，首先调用派生类的析构函数，然后调用基类的析构函数
- 一般来说，一个类如果定义了虚函数，则应该将析构函数也定义成虚函数。或者，一个类打算作为基类使用，也应该将析构函数定义成虚函数。
- 注意：不允许以虚函数作为构造函数

虚析构函数的必要性

- 例7.18 虚析构函数的重要性。

```
#include<iostream.h>

class A{
public:
  A( )
  { cout<<"call A( )\n"; }
  ~A( )          //L1
  { cout<<"call ~A( )\n"; }
};

class B:public A{
  char* buf;
public:
  B( )
  {  buf=new char[100];
     cout<<"call B( )\n";
  }
  ~B( )
  { delete []buf;
    cout<<"call ~B( )\n";
  }
};
void main( )
{  A *p=new B;  //L2
   delete p;    //L3
}
```

程序运行结果:
call A()
call B()
call ~A()

- 问题：main()函数执行L3行语句时，因p是基类类型指针，故仅调用基类的析构函数~A()，而不调用派生类的析构函数~B()。这样，派生类的动态对象申请分配的100字节动态内存未被释放。

- 如何解决？有以下三种方法：
  1.将L3行的语句改为：delete (B*)p;
  2.或将L2行的语句改为：B *p=new B;
  3.或将L1行的语句改为：virtual ~A()

  第三种方法将基类A的析构函数说明为虚函数，则其派生类B的析构函数自动成为虚函数。这样，当main()函数执行L3行语句时，因p指向一个派生类对象，故将调用派生类的析构函数~B()。
- 从使用的简便性上看，通常把析构函数说明为虚函数。

思考

“多态”的关键在于通过基类指针或引用调用一个虚函数时，编译时不确定到底调用的是基类还是派生类的函数，运行时才确定 ---- 这叫“动态联编”。“动态联编”到底是怎么实现的呢？

提示：请看下面例子程序：

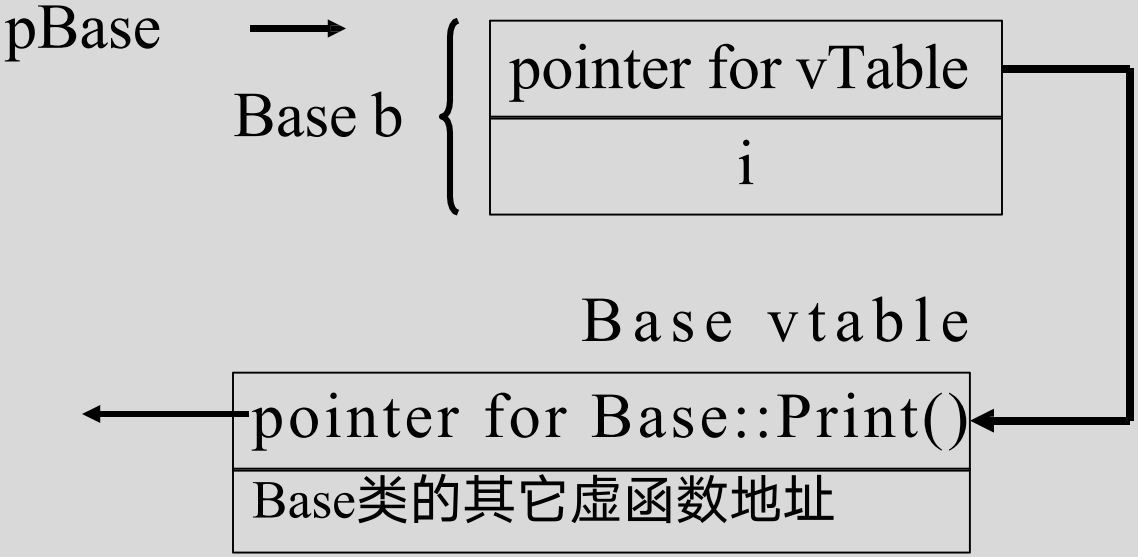
```
class Base {
        public:
        int i;
        virtual void Print() { cout << "Base:Print" ; }
};
class Derived : public Base{
        public:
        int n;
        virtual void Print() { cout <<"Drived:Print" << endl; }
};
int main() {
        Derived d;
        cout << sizeof( Base) << ","<< sizeof( Derived ) ;
        return 0;
}
```

程序运行输出结果： 8, 12

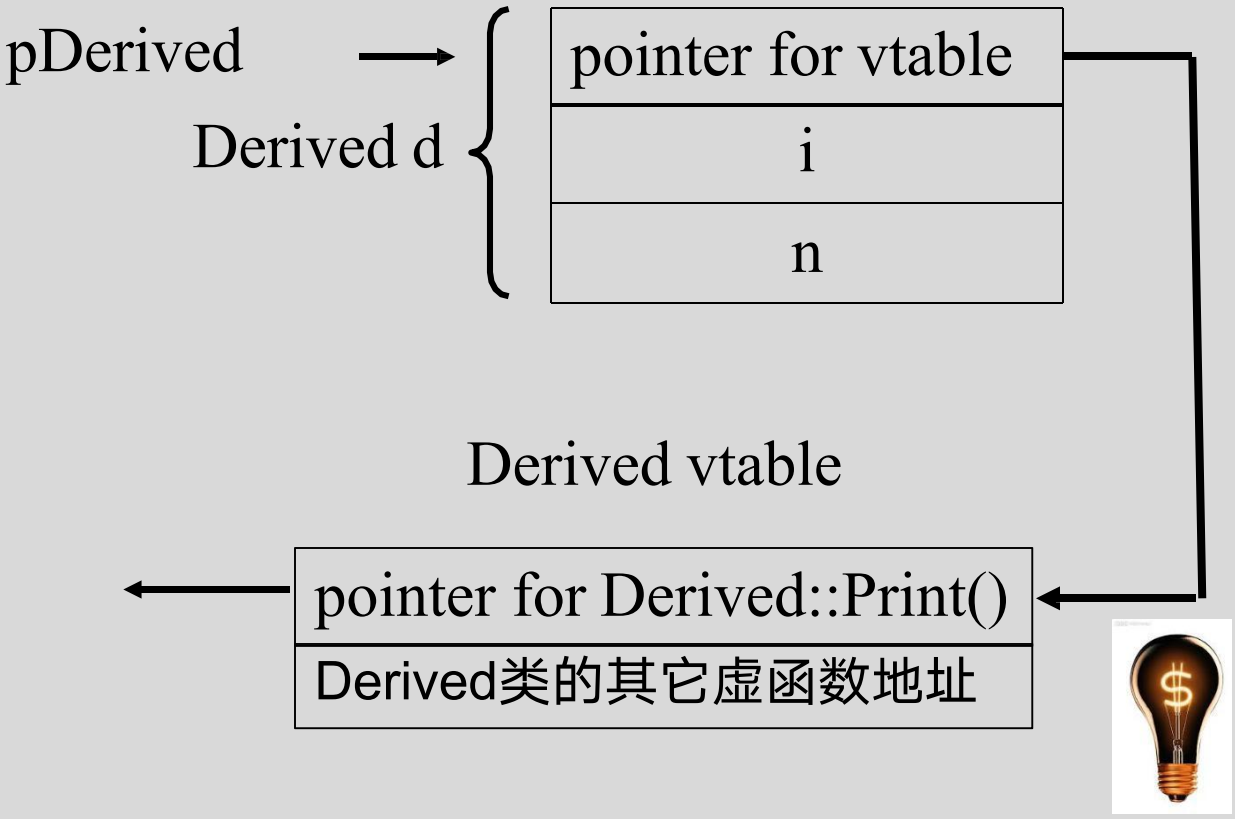
为什么都多了4个字节？

多态实现的关键--- 虚函数表

每一个有虚函数的类（或有虚函数的类的派生类）都有一个虚函数表，该类的任何对象中都放着虚函数表的指针。虚函数表中列出了该类的虚函数地址。多出来的4个字节就是用来放虚函数表的地址的。



多态实现的关键--- 虚函数表




```
pBase = pDerived;
pBase->Print();
```

➢多态的函数调用语句被编译成一系列根据基类指针所指向的（或基类引用所引用的）对象中存放的虚函数表的地址，在虚函数表中查找虚函数地址，并调用虚函数的指令。

纯虚函数

- 若类的某些虚函数只能抽象出原型，无法定义其实现，则可定义为纯虚函数，其实现由它的派生类定义。
- 纯虚函数的定义格式：

  virtual <type> FuncName(<ArgList>)=0;

  - 定义纯虚函数时，其实现不能在类内同时定义，但可在类外或其派生类中定义。如例7.19。
  - 虚函数名赋值为0，与函数体为空不同。
  - 含有纯虚函数的类肯定是抽象类，因虚函数没有实现部分，不能创建对象。但可定义抽象类类型的指针(或引用),以便用这种基类类型的指针变量指向其派生类的对象(或用这种基类类型的引用变量关联其派生类的对象)时，调用其派生类重定义的纯虚函数，引发运行时多态性。

例7.19 纯虚函数

```
#include<iostream.h>
class A{
protected:
   int x;
public:
   A(){ x=1000; }
   virtual void print()=0;
};
void A::print(){  cout<<"x="<<x<<'\n'; }

class B:public A{
   int y;
public:
   B(){ y=2000; }
   void print(){  cout<<"y="<<y<<'\n'; }
};

void main(void)
{  B b;
   A *pa=&b;pa->print();pa->A::print();
}
```

定义纯虚函数：
①接口统一②接口与其实现分离。

③纯虚函数的实现只能在类外或其派生类中定义，通常为后者。

④定义抽象类类型的指针指向其派生类的对象，调用其派生类重定义的纯虚函数，引发运行时多态性。

程序运行结果：

```
y=2000
x=1000
```

综合应用举例

- **纯虚函数的好处：**接口与其实现分离，提高了类的抽象层次，便于建立接口统一、功能与时俱进的类体系，如微软的MFC类体系和第14章介绍的I/O流类体系等。

- 例7.20 T是计算 $\int_a^b f(x)dx$ 的抽象类；Tf1是T的派生类，计算 $\int_a^b (x-\sin x)dx$ 。数值积分使用矩形法。

```
#include<iostream.h>
#include<math.h>

class T{      //定义一个使用矩形法求定积分的抽象类
protected:
   float a,b; //分别为积分下限和上限
   int n;     //矩形积分的等分数
public:
   T(float a,float b,int n)
   { this->a=a; this->b=b; this->n=n; }
   virtual float f(float x)=0;//纯虚函数：  求f(x)的值
   float Area(void);          //求f(x)在[a,b]上的积分
   void Show(char *str)
   { cout<<"函数f(x)="<<str<<"在["<<a<<','<<b
         <<"]上的积分值="<<Area()<<endl;
   }
};
```

```
float T::Area(void)
{  float s=0.0f,x=a,h=(b-a)/n;
   for(int i=0;i<n;i++)
   { s+=f(x); x+=h; }
   return s*h;
}
class Tf1:public T{
   float f(float x)//重定义纯虚函数
   { return x-sin(x); }
public:
   Tf1(float a=0,float b=2,int n=100):T(a,b,n){ }
};
void main(void)
{  T* p=new Tf1(0,3);
   p->Show("x-sinx");
   delete p;
}
```

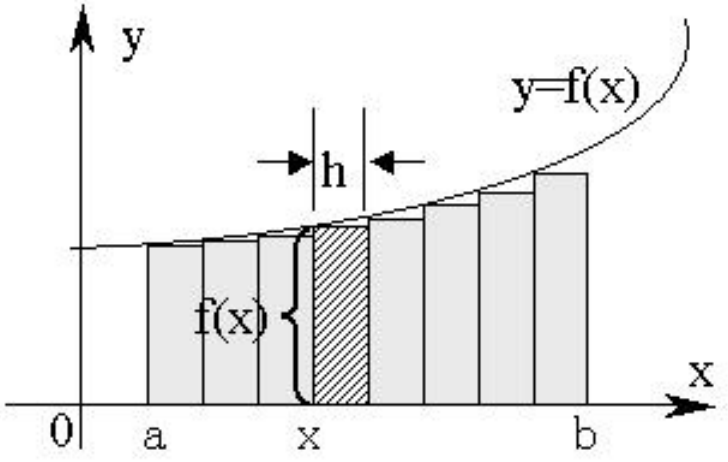


图 13-5　矩形法求定积分

程序运行结果:

函数f(x)=x-sinx在[0,3]上的积分值=2.46727

- 例7.21* 建立一个通用单向链表，完成插入一个结点、删除一个结点、查找一个结点操作，并输出链表上的各个结点值。并使结点只含一个整数，测试所建立的通用单向链表。
- 分析：由于单向链表的插入、删除、查找等操作都是相同的，只是结点上包含的数据随不同的应用有所不同，因此，抽象表达结点数据是编写通用单向链表操作程序的关键。

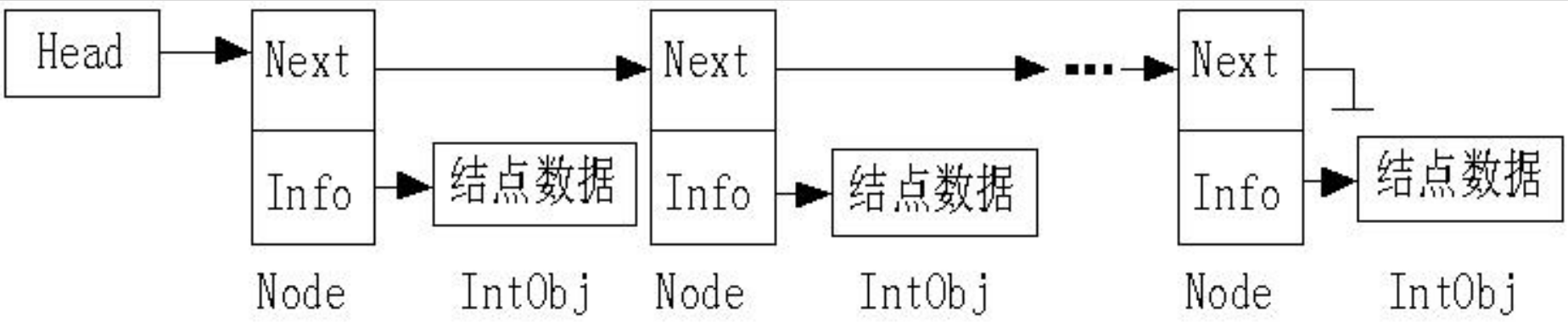


图 13-6 单向链表结构

- 下面结合图13-6阐述编程思路：

1. 单向链表结点用结点类Node描述。类Node的成员数据中，包括一个指向结点数据的指针Info和一个指向下一个结点的指针Next。其中，Info是抽象结点数据类Object的指针。
2. 实际结点数据由结点数据类IntObj描述。IntObj是抽象结点数据类Object的派生类，成员数据是一个整数，成员函数完成两个结点的比较，输出结点上的数据等。
3. 单向链表由类List描述。List的成员数据为指向链表的首指针Head，成员函数实现链表的各种操作，如插入、删除一个结点等。由于类List需要频繁访问类Node，为了提高类List访问类Node的效率，将类List说明为类Node的友元。

```
//ex13_21.h
#include<iostream.h>

class Object{ //抽象结点数据类：用于派生实际结点数据类
public:
  Object( ){ }                    //可省略
  virtual int IsEqual(Object &)=0;//结点数据的相等比较
  virtual void Show( )=0;         //输出结点数据
  virtual ~Object( ){ };          //虚析构函数
};

class Node{           //结点类
  Node *Next;         //指向下一个结点
  Object *Info;       //指向结点的数据域
public:
  Node( ){  Info=NULL; Next=NULL;  }
  Node(Node &node)   //拷贝构造函数
  {  Info=node.Info; Next=node.Next;  }
```

```
  void FillInfo(Object*obj){Info=obj;}//Info指向数据域
  friend class List;                  //声明友元类
};

class List{                  //定义单向链表类
  Node *Head;                //链表首指针
public:
  List( ){  Head=NULL;  }
  ~List( ){  DeleteList( ); }
  void AddNode(Node *);    //在链表首加一个结点
  Node* DeleteNode(Node*); //删除链表中的一个指定结点
  Node* LookUp(Object&);   //在链表中查找一个指定结点
  void ShowList( );        //输出整条链表上的数据
  void DeleteList( );      //删除整条链表
};
```

```
void List::AddNode(Node *node)    //在链表首加一个结点
{  if(Head==NULL)                 //若为空链表时
   {  Head=node; node->Next=NULL;  }
   else
   {  node->Next=Head; Head=node;  }
}

Node* List::DeleteNode(Node *node)//从链表上删除指定结点
{  if(node==Head) Head=Head->Next;//删除首结点
   else                           //删除中间结点或尾结点
   {  Node *p1,*p2;
      p1=p2=Head;
      while(p2!=node&&p2->Next)
      {  p1=p2; p2=p2->Next;  }
      if(p2==node) p1->Next=p2->Next;
   }
   return  node;
}
```

```
Node* List::LookUp(Object &obj)//从链表上查找一个结点
{  Node *p=Head;
   while(p)
   {  if(p->Info->IsEqual(obj)) return  p;
      p=p->Next;
   }
   return NULL;
}

void List::ShowList( )        //输出链表上各结点的数据
{  Node *p=Head;
   while(p)
   { p->Info->Show( );
     p=p->Next;
   }
}
```

```
void List::DeleteList( )       //释放链表上各结点
{  Node *p;
   while(Head)
   {  p=Head;
      delete p->Info;
      Head=p->Next;
      delete p;
   }
}
```

```
// ex13_21.cpp
#include "ex13_21.h"

class IntObj : public Object{//由Object派生的实际结点数据类
  int data;              //实际结点数据
public:
  IntObj(int x=0){ data=x;  }
  void SetData(int x){  data=x;  }
  int IsEqual(Object& obj)//重定义虚函数
  {  IntObj& t=(IntObj&)obj;
     return data==t.data;
  }
  void Show( )           //重定义虚函数
  { cout<<data<<'\t'; }
};
```

```
void main(void)
{  IntObj *p;
   Node *pn,*pt,node;
   List list;

   for(int i=0;i<5;i++)//建立一个包含5个结点的单向链表
   {  p=new IntObj(i+100);//动态建立一个IntObj类的对象
      pn=new Node;        //建立一个新结点
      pn->FillInfo(p);    //填写新结点的数据域
      list.AddNode(pn);   //将新结点加入链表尾
   }
   list.ShowList();       //输出链表上各结点的数据值
   cout<<endl;

   IntObj da;
   da.SetData(102);       //初始化da对象
   pn=list.LookUp(da);    //从链表上查找含da对象的结点
```

```
    //若找到，则从链表上摘下该结点
    if(pn) pt=list.DeleteNode(pn);
    list.ShowList();        //输出链表上各结点的数据值
    cout<<endl;

    if(pn) list.AddNode(pt);//将摘下的结点加入链表首
    list.ShowList();        //输出链表上各结点的数据值
    cout<<endl;
}
```

例7.22*几何形体处理程序

几何形体处理程序: 输入若干个几何形体的参数， 要求按面积排序输出。输出时要指明形状。

Input:

第一行是几何形体数目n（不超过100).下面有n行，每行以一个字母c开头.

若 c 是 ‘R’，则代表一个矩形，本行后面跟着两个整数，分别是矩形的宽和高

； 若 c 是 ‘C’，则代表一个圆，本行后面跟着一个整数代表其半径

若 c 是 ‘T’，则代表一个三角形，本行后面跟着三个整数，代表三条边的长度

几何形体处理程序

Output:

按面积从小到大依次输出每个几何形体的种类及面积。每行一个几何形体，输出格式为：

形体名称：面积

几何形体处理程序

Sample Input:

3
R 3 5
C 9
T 3 4 5

Sample Output

Triangle:6
Rectangle:15
Circle:254.34

```
#include <iostream>
#include <stdlib.h>
#include <math.h>
using namespace std;
class CShape
{
```

- public:
- virtual double Area() = 0; //纯虚函数
- virtual void PrintInfo() = 0;
- };
- class CRectangle:public CShape
- {
- public:
- int w,h;
- virtual double Area();
- virtual void PrintInfo();
- };

- class CCircle:public CShape { public:
- int r;
- virtual double Area(); virtual void PrintInfo();
- };
- class CTriangle:public CShape { public:
- int a,b,c;
- virtual double Area(); virtual void PrintInfo();
- };

```
double CRectangle::Area()
{
return w * h;
}
void CRectangle::PrintInfo() {
cout << "Rectangle:" << Area() << endl;
}
double CCircle::Area() {  return 3.14 * r *
r ;
}
void CCircle::PrintInfo() {
cout << "Circle:" << Area() << endl;
}
```

```
double CTriangle::Area() {  double p = (
    a + b + c) / 2.0;
return sqrt(p * ( p - a)*(p- b)*(p - c));
}
void CTriangle::PrintInfo() {
cout << "Triangle:" << Area() << endl;
}
```

```
CShape * pShapes[100];
int MyCompare(const void * s1, const void * s2);

int main()
{
int i; int n;
CRectangle * pr; CCircle * pc; CTriangle * pt;
cin >> n;
for( i = 0;i < n;i ++ ) {
char c;
cin >> c;  switch(c) {
case 'R':
pr = new CRectangle();  cin >> pr->w >> pr->h;
pShapes[i] = pr;
break;
```

```
case 'C':
pc = new CCircle();  cin >> pc->r;
pShapes[i] = pc;  break;
case 'T':
pt = new CTriangle();
cin >> pt->a >> pt->b >> pt->c;
pShapes[i] = pt;  break;
}
}
qsort(pShapes,n,sizeof(
CShape*),MyCompare);  for( i = 0;i <n;i
++)
pShapes[i]->PrintInfo();
return 0;
}
```

```
int MyCompare(const void * s1, const void * s2)
{
double a1,a2;
CShape * * p1 ; // s1,s2 是void * ，不可写 “* s1”来取得s1指向的内容
CShape * * p2;
p1 = ( CShape * * ) s1; //s1,s2指向pShapes数组中的元素，数组元素的类型是CShape *
p2 = ( CShape * * ) s2; // 故p1,p2都是指向指针的指针，类型为 CShape **
a1 = (*p1)->Area();              // * p1 的类型是 Cshape * ,是基类指针，故此句为多态
a2 = (*p2)->Area();  if( a1 < a2 )
return -1;
else if ( a2 < a1 )
return 1;  else
return 0;
}
```

```
            case 'C':
            pc = new
            CCircle();  cin >>
            pc->r;  pShapes[i]
            = pc;  break;
        case 'T':
        pt = new CTriangle();
        cin >> pt->a >> pt->b >> pt->c;
        pShapes[i] = pt;  break;

    }
  }
  qsort(pShapes,n,sizeof( CShape*),MyCompare);
  for( i = 0;i <n;i ++)
  pShapes[i]->PrintInfo();
  return 0;
}
```

如果添加新的几何形体，比如五边形，则只需要从CShape派生出 CPentagon, 以及在main中的switch语句中增加一个case，其余部分不变有 木有!

猫科动物叫声派生类程序

猫科动物叫声派生类程序: 定义猫科动物Felid类，派生出猫科（Cat）和豹类（Leopard），二者都包含虚函数sound()，要求根据派生类对象的不同调用各自重载后的成员函数。

分析:
虚函数sound重载后，派生类对象调用的是自己的sound成员函数，而不会是基类或者其他子类的sound成员函数。
但是，如果我们希望用统一的接口（基类指针），当指向不同子类对象的时候，调用sound函数是子类自己的成员函数，而不是基类的成员函数，sound就必须是虚函数。基类的sound可以是纯虚函数么？

猫科动物叫声派生类程序

Sample Output

猫科动物都会叫
喵。。。
嗷。。。