4.3 派生类的构造函数和析构函数

当创建派生类对象时，由于派生类继承了基类的所有成员，派生类的每一个对象也包含了基类的数据成员的值，就象派生类包含一个基类的对象作为无名成员，因此就必须要提供一种机制，使得在创建派生类的对象时能够自动初始化它所继承的基类数据成员，在C++中这是通过派生类的构造函数来实现的。

派生类构造函数的构成与包含成员对象的容器类(Container Class)构造函数是类同的。将基类看成是派生类的一个无名成员一样，在派生类构造函数定义时在函数头设置基类构造函数的初始化列表。

1.派生类构造函数应提供一种初始化机制，使得派生类对象在创建时，能自动地调用基类的构造函数来初始化基类的数据成员，其格式：

```
派生类名::派生类构造函数名(总参数表)
  : 基类构造函数名(参数表), 成员对象(参数表)
{                <函数体>                }
```

由此可知，基类构造函数是由基类名来标识，成员对象的构造函数是由成员对象名来标识，派生类构造函数的总参数表，应提供足够的参数对基类和成员对象的数据成员进行初始化。


```
#include <iostream>
using namespace std;
class Base {
  float  x, y;
public:
  Base(float x1, float y1);
  ~Base( );
  void SetBase(float x1, float y1)
  { x = x1;  y = y1; }
  float GetX( ){ return x; }
  float GetY( ){ return y; }
  void Print( )
      { cout << " 基类对象或派生类对象继承的x值 = " << x
<< "\t  y值 = " << y << endl; }
};
```



```
class Derived : public Base {
  float z;
  Base a;
public:
  Derived(float x1, float y1,float x2,
          float y2, float z1);
  ~ Derived( );
  void SetDerived(float x1, float y1,float x2,
                  float y2, float z1)
     {    Base::SetBase(x1, y1);
    a.SetBase(x2, y2);      z = z1;  }
  float GetZ( ){ return z; }
  void Print( )
     {  Base::Print( );
        a.Print( );
        cout << " 派生类对象的z值 = "<< z << endl;
     }
};
```

```
Base::Base(float x1, float y1)
{ //内部静态变量i记录基类构造函数被调用次数
   static int i = 0;
   x = x1;  y = y1;
   cout << "第" << ++i
        << "次调用基类的构造函数 !\n";
}


Base::~ Base( )
{     //内部静态变量i记录基类的析构函数被调用次数
  static int i = 0;
  cout << "第" << ++i
       << "次调用基类的析构函数 !\n";
}
```



```
Derived::Derived(float x1,
                 float y1,float x2,
                 float y2, float z1)
     : Base(x1, y1), a(x2, y2)
/* 派生类构造函数的初始化列表，既包括对从基类所继
 承的成员初始化，又包括成员对象a的初始化。*/
{    //内部静态变量j记录派生类构造函数被调用次数。
  static int j = 0;
    z = z1;
    cout << "第" << ++j
         << "次调用派生类的构造函数 !\n";
}
Derived::~Derived( )
{   static int j = 0;
    cout << "第" << ++j
         << "次调用派生类的析构函数 !\n";
}
```



```
void main( )
{ cout << "(1) 对基类Base的测试 : \n";
     Base b(10, 20);
  b.Print( );
     cout << "(2)对派生类Derived的测试:\n";
  Derived d(1, 2, 3, 4, 5);
  d.SetDerived(11, 12, 6, 7, 15);
  d.Print( );
     cout << "(3) 其它测试 : \n";
     float x1 = b.GetX( );
     float y1 = b.GetY( );
  float z1 = d.GetZ( );
  cout << " 读取基类对象b的x值 = "
  << x1 << " , y值 = " << y1 << endl;
  cout << " 读取派生类对象d的z值 = " << z1 << endl;
     cout << "(4) 现在开始撤消所有对象，请注意撤消顺序 :
\n";
}
```


该程序的输出结果为：

```
(1) 对基类Base的测试 :
第1次调用基类的构造函数 !（创建基类对象b时）
基类对象或派生类对象继承的x值 = 10
                          y值 = 20
(2) 对派生类Derived的测试 :
第2次调用基类的构造函数 !
（创建派生类对象d时，先初始化从基类继承的成员）
第3次调用基类的构造函数 !（再初始化对象成员a）
第1次调用派生类的构造函数 !
（最后初始化派生类成员）
基类对象或派生类对象继承的x值 = 11
                          y值 = 12
 基类对象或派生类对象继承的x值 = 6
                          y值 = 7
 派生类对象的z值 = 15
```

(3) 其它测试 :

读取基类对象b的x值 = 10 , y值 = 20

读取派生类对象d的z值 = 15

(4) 现在开始撤消所有对象，请注意撤消顺序 :

第1次调用派生类的析构函数 !

（撤消派生类对象d时，先撤消派生类新增成员）

第1次调用基类的析构函数 !（再撤消对象成员a）

第2次调用基类的析构函数 !

（最后撤消从基类继承的成员）

第3次调用基类的析构函数 !（撤消基类对象b）

说明：

1.在基类Base中定义了一个带有两个参数的一般构造函数，派生类Derived中还包含了一个基类的对象b作为私有数据成员，则派生类的构造函数共带有5个参数，前两个参数x1，y1用于基类数据成员的初始化，x2，y2用于成员对象b的初始化，z1用于初始化派生类的私有数据成员z。

2.当创建派生类的对象d时，首先调用基类的构造函数Base，初始化派生类Base继承基类的数据成员x和 y，然后对成员对象b进行初始化，最后才执行派生类构造函数，一言以蔽之一个派生类构造函数的执行次序是，基类第一，内部的成员对象第二，派生类最后。


```
void main( )
{
   Base b(10,20)
   b.Print();
   Derived d(1, 2, 3, 4, 5);
        …
   d.Print( );
        …
}
```

main()函数中创建派生类对象d时，首先调用基类的构造函数，初始化DerivedClass继承基类的数据成员x和y，然后调用对象b的构造函数，初始化对象成员b的数据成员，最后再初始化新添加的数据成员z。

3.析构函数的操作顺序与构造函数相反，当派生类的一个对象离开作用域，自动地调用析构函数，其执行顺序是派生类第一，内部成员对象第二，基类最后，源程序中不允许显式地调用析构函数。

4.4基类和派生类的赋值规则

1.在公有继承方式下，可以把一个派生类对象赋值给一个基类对象。在此赋值中，只有基类中的成员被复制。这是反映客观实际的。例如，

```
Point p;        Circle c;
```

由于基类 ⊂ 派生类，可以写： `p = c;`

一般格式为： 基类对象 = 派生类对象;

c到p的赋值，实际上是进行如下操作：

```
p.x = c.x;   p.y = c.y;
```

其中，只有x和y被复制，即复制了派生类对象c的圆心坐标值x和y，而radius未被复制，因为它不在Point基类中。但是反过来不能把基类Point的对象p赋值给派生类对象c。

```
c = p;      //error
```

因此基类和派生类的赋值是单方向的，即

派生类对象 → 基类对象。

C++ 程序设计

2.赋值规则扩展到对象指针和对象引用。在公有继承方式下，则指向派生类对象的指针可以自动地转换为指向基类对象的指针。例如可以将Circle类的对象指针赋值给Point类的对象指针：

```
Point p, * bp = &p;
Circle c, * dp = &c;
```

则可以写： `bp = dp;`

//隐式转换后，基类的对象指针bp就指向了派生类对象c。

一般格式为：

基类的对象指针 = 派生类对象指针;

也可把Circle类的对象引用赋值给Point类的对象引用，如：

```
Point p;
point &p_ref = p;
Circle c;
Circle &c_ref = c;
```

则可以写： `p_ref = c_ref;`

C++ 程序设计

一般格式为：基类的对象引用 = 派生类对象引用;

但反之不能进行，即不能把基类Point的对象、对象指针和对象引用赋值给派生类Circle的对象、对象指针和对象引用，例如：

```
c = p;                  //error
dp = bp;                //error
c_ref = p_ref;          //error
```

但基类的对象指针转换成派生类的对象指针必须显式进行，例如：

```
Point p, * bp = &p;
Circle c, * dp = &c;
dp = (Circle *)bp;// 显式转换
```

在私有继承方式下，由于基类公有成员在派生类中变成私有的，所以基类的公有成员只能通过基类的对象指针访问，而不能通过派生类的对象指针访问，因此派生类的对象指针和基类的对象指针之间的所有转换都不能进行。

C++ 程序设计

3.基类和派生类的(对象)指针

指向基类的指针和指向派生类的指针是密切相关的，其表现为：

(1)一个基类类型的指针可用来指向基类公有派生类的任何对象，这是C++实现程序运行时多态性的关键途径，

```
#include <iostream>
using namespace std;
class Point {
public :
  int x ,y;
  Point(int xi, int  yi)
  { x = xi;  y = yi; }
};
class Circle : public Point {
public :
  int  radius;
  Circle(int x1, int y1, int r): Point(x1, y1)
  {
      radius = r;
  }
};
```


C++ 程序设计

```
void main( )
{ Point pnt(160, 180), * bp;
  Circle spotlight(320, 240, 40), * dp;
  bp = & pnt;
  cout << "(1) pnt's X = " << bp -> x
       << "\t Y = " << bp -> y << endl;
  bp = & spotlight;
  cout << "(2) spotlight's X = "
    << bp -> x << "\t Y = " <<  bp -> y ;
  dp = & spotlight;
  cout << "\t radius = "
       << dp -> radius << endl;
  cout << "(3) spotlight's X = "
    << bp -> x << "\t Y = " <<  bp -> y ;
  cout << "\t radius = "
    << ((Circle *)bp) -> radius << endl;
}
```

该程序的输出结果：

```
(1) pnt's X = 160        Y = 180
(2) spotlight's X = 320  Y = 240
                              radius = 40
(3) spotlight's X = 320  Y = 240
                              radius = 40
```

在例程中，

公有继承
Point → Circle

在main()函数内定义的基类指针bp，不仅可以指向基类point的对象pnt，还可用来指向公有派生类Circle的对象spotlight，因此，用它同样可以代替派生类对象名spotlight去访问公有派生类中从基类Point继承的公有成员（数据成员和成员函数）。但是它不能访问派生类中的新增成员，如例中的新增数据成员radius。

(2)如果希望用基类指针访问其公有派生类的新增成员，必须将基类指针用强制类型转换成派生类指针。格式为：

((派生类名*)基类指针名) -> 新增成员

例程中：

```
cout << "\t radius = "
     << ((Circle *)bp) -> radius << endl;
```

(3)在公有继承方式下，基类指针是联系基类和派生类对象间消息传递的桥梁。例程中是采用地址赋值语句使基类指针bp指向了公有派生类对象spotlight。但更多的情况是在调用函数时，用实参传递给形参来实现这种地址赋值。因此任何使用基类指针作为形参的函数，都同样可以用来处理派生类的对象，只需在调用该函数的语句中，以“&公有派生类对象名”作为实参即可，格式为：

函数原型：<返回>函数名(基类名*指针名，…)；
调用语句：函数名(&公有派生类对象名， …)；

在函数体内凡是对基类对象的数据成员进行操作，即是用来对公有派生类中从基类继承的数据成员进行操作。

(4)由于引用也是通过地址传递，且采用引用传递函数的参数还可以避免取地址运算和取内容运算的麻烦。因此，凡是使用对基类对象的引用作为形参的函数，同样可以用来处理公有派生类的对象，其格式：

函数原型：<返回>函数名(基类名&引用名，…)；
调用语句：函数名(公有派生类对象名， ...)；

4.子类型和类型适应：若类B是类A的公有派生类，则称类B是类A的一个子类型，当然类A还可以有其他的子类型。根据上述的赋值规则，凡是基类A的对象可使用的任何地方，都可以用公有派生类B的对象代替之，这正是有些文献上所称的B类型适应A类型，所谓“类型适应”是指两种类型之间的关系，即类A中的操作可以被用于操作类B的对象。例如：考察确定两点之间距离的函数Distance()，

```
double Distance(Point &a, Point &b)
{  double dx = a.x - b.x;
   double dy = a.y - b.y;
   return sqrt(dx * dx + dy * dy);
}                Point &a = k;  Point &b = r;
void main()
{
   Circle k , r;     //公有派生类的对象
   Double dc = Distance(k , r);
         …
}
```

Distance()函数的两个形参是基类Point的对象引用，当公有派生类的对象k和r作为实参赋给形参a和b后，其函数体内凡是对基类Point的对象引用a 和b所进行的各种操作，实际上都是用来对公有派生类的对象k和r进行操作，由于“类型适应”而不会发生任何问题。

应该注意的是当Circle 类的k和r对象作为实参赋值给形参a和b时，如前所述只有基类中的数据成员被复制，因此实际上是把每个对象的圆心点坐标值传递给Distance()函数，然后由该函数计算出两圆心的距离。

5.继承传递过程中，如下成份将不能被继承：

(1)构造函数和析构函数：在创建派生类对象时，自动调用基类的构造函数，但它们只能在建立派生类对象时由编译系统自动调用，基类的构造函数不能象其他被继承的成员那样由派生类显式调用，写在程序上的只能是初始化列表。当对象离开作用域时由编译器自动调用析构函数，派生类也不允许显式地调用基类的析构函数。

例如：
```
B(int v, int t) : A(v)
{  A::A(v);/*error基类构造函数不能象其他被继承的成员那样由派生类显式调用。*/
   Total = t;
}
```

(2)友元关系也是不可继承的吗？(是可以部分继承的)。这与现实生活也是类似的，父母的朋友不一定自然是子女的朋友，继承树中类的友元联系并不向上或向下传递。但是父母的朋友可以访问子女继承的父母的部分。（测试示例）

4.5多继承

1．多继承派生类：可同时具有多个基类的派生类称为“多继承派生类”。它同时继承了这些基类的所有成员。因此每个派生类的对象都包含了所有基类的成员，如前所述，多继承派生类可以组成有向图层次结构。

定义一个多继承派生类与单继承派生类类似，不仅要指明该多继承派生类从哪几个基类继承而来，而且还要指明每个基类的访问限制符，其格式如下：

```
class 派生类名 : <继承方式1> 基类1,
                 <继承方式2> 基类2, …
{                ...                }
```



```
#include <iostream>
using namespace std;
#define  WHITE  15
const double PI = 3.14159;
class Circle {
   double radius;
public :
   Circle(double r) { radius = r; }
   double Area(void)
   { return PI * radius * radius; }
};
class Table {
   double height;
public :
   Table(double h) { height = h; }
   double Height( ) { return height; }
};
```

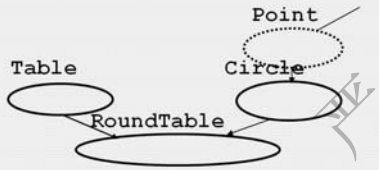
```
class RoundTable :
      public Table , public Circle {
   int color;
public :
   RoundTable(double h, double r,
                int c);
   int Color( ) { return color; }
};


RoundTable::RoundTable(double h,
 double r, int c) : Circle(r), Table(h)
{ color = c; }
```



```
void main( )
{
   RoundTable table(160.0, 180.0,
                        WHITE);
   cout << "The table
           properties are:\n";
   cout << "\t Height = "
        << table.Height( ) << endl;
   cout << "\t Area = " << table.Area( )
        << endl;
   cout << "\t Color = "
         << table.Color( ) << endl;
}
```


如上例中，对一组指明的基类，每个基类都有各自的访问限制符，这些基类就是直接基类。例如要定义多继承派生类RoundTable(圆桌)，应具有桌子的特征和圆的几何特征，可用下图的层次结构图表示。

RoundTable类的间接基类



多继承派生类RoundTable的类层次结构

Table类和Circle类是RoundTable类的基类，但Table类和Circle类的任一基类都将是RoundTable类的间接基类。如前所述，Point类(为了简单在例程table.cpp中未写出)是Circle类的直接基类，则又是RoundTable类的间接基类

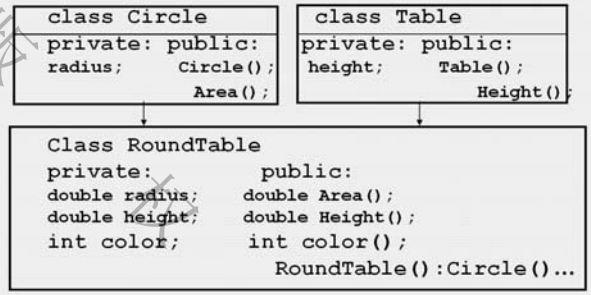


图4.12RoundTable类继承了上层基类的成员

2.多继承派生类的构造函数：其格式为：

多继承派生类名(总参数表) ：
  基类名1(参数表1)，基类名2(参数表2)，…
  成员对象名(参数表n + 1)，…
{　　多继承派生类构造函数体　　}

(1) 其中总参数表所提供的参数应包含其后的各个分参数表。

(2) 多继承派生类的构造函数必须承担调用该派生类所有直接基类构造函数，以完成这些基类的初始化任务，同时派生类构造函数的参数个数必须包含完成所有基类初始化任务所需的参数个数(基类有无参构函的除外)。

(3) 多继承派生类构造函数的执行顺序是先调用所有基类的构造函数，再执行派生类本身的构造函数。处于相同层次的各基类构造函数的调用顺序取决于多继承派生类定义时，在类头部分指定各基类的先后顺序，而与多继承派生类构造函数定义时，函数头所列的成员初始化表的排列次序无关。

3.虚基类：

虚基类仅用于多继承，因为多继承建立的类层次结构中很可能有复杂的关系，有时编程者必须对基类继承的某个方向要进行一定程度的控制。因为一个类不能多次作为某个派生类的直接基类，但可以多次作为一个派生类的间接基类。例如：

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

 //error，类B不能两次作为类D的直接基类
```



```
class A {
public :
  int value;
        …
};
class B : public A { … };
class C : public A { …};
class D : public B, public C {
public :
  int Value() { return value; }
        …
};
```

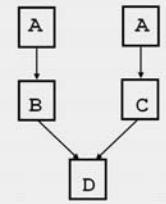



图4.13 两条继承路径的类层次结构

这相当于实际生活中有两个祖父，他们不仅是孪生兄弟，而且名字完全相同，他们的孙子辈无法区分。如何解决这种二义性问题呢？

```
#include <iostream>
using namespace std;
class A {
public :
  int value;
  A(int v) { value = v; }
};
class B : virtual public A {
public :
      int Bvalue;
      B(int v, int b) : A(v) { Bvalue = b; }
      int BValue() { return Bvalue; }
};
class C : virtual public  A {
public :
      int Cvalue;
      C(int v, int c) : A(v) { Cvalue = c; }
      int CValue() { return Cvalue; }
};
```



```
class D : public B, public C {
public :
  int Dvalue;
  int Value() { return value; }
  D(int v1,int v2, int b, int c, int d)
        : B(v1, b), C(v2, c), A(v1)
  { Dvalue = d; }
  int DValue() { return Dvalue; }
};

void main( )
{ D obj(1, 2, 3, 4, 5);

  cout << " value of obj = "
       << obj.Value() << endl;
}
```


该程序的输出结果：

```
value of obj = 1
```

(1) 多继承中的二义性问题：从例程4-1-11.cpp可知，类A是类D的间接基类，但是类A通过两个继承路径到底层的派生类D，因此派生类D的每个对象同时保存了两份类A数据成员value的值，从而产生了二义性

(2) 虚基类：将基类定义为虚基类，就可以确保该基类通过多条路径为派生类继承时，派生类仅仅只继承该基类一次，即本例中派生类D的对象obj只保存一份该基类A的数据成员value。

```
class 派生类名 : virtual <继承方式> 基类
{                    …                }
```

其中virtual是指明虚基类的关键字，例4.11中采用虚基类机制后，可简写为：

```
class A {
public :
  int value;
          …
};
class B : virtual public A { … };
class C : virtual public A { …};
class D : public B, public C {
public :
  int Value() { return value; }
          …    };
```


其层次结构图变为如图4.14所示，由此可知

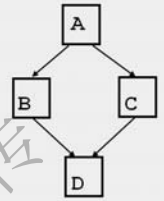


图4.14 使用虚基类A的类层次结构

将基类A定义成虚基类后，派生类D的对象obj只保存一份虚基类A的数据成员value值，从而解决了多继承中的二义性问题。

顺便指出，在定义虚基类时，关键字virtual和 public顺序可以互换而不产生编译错误。同时，为了确保虚基类在派生类中只继承一次，就必须将它在所有直接派生类中指定为虚基类，否则仍然是多次继承而导致二义性。

(3) 虚基类的构造函数：虚基类仅用于多继承，且它的构造函数必须只被调用一次。

★若一个派生类有一个直接或间接的虚基类，那么由该虚基类直接或间接继承的派生类，在这些派生类构造函数的初始化列表中都应列出对该虚基类构造函数的调用，如例程4-1-11.cpp中：

```
class B : virtual public A {
public :
      B(int v, int b) : A(v) { Bvalue = b; }
            …
};
class C : virtual public  A {
public :
      C(int v, int c) : A(v) { Cvalue = c; }
            …
};
```


此外，该派生类构造函数的初始化列表中还必须列出对虚基类构造函数的调用，如果没有列出则编译系统将调用该虚基类的缺省构造函数。

如例程vtbase.cpp中：

```
class D : public B, public C {
public :
  D(int v1,int v2, int b, int c, int d)
        : B(v1, b), C(v2, c), A(v1)
  { Dvalue = d; }
            …
};
```