Lewolf
保留版权
签名本验证

第二章 面向对象的程序设计

上一章我们简单介绍了计算机程序设计的发展过程，让读者对程序设计有一个整体上的了解。结构化程序设计相对简单、易懂，一般的读者都熟练掌握。面向对象程序设计包含了比较深的思想过程，又是面向状态程序设计的基础，本书的读者应当对其有比较深入的了解。本章将系统的介绍一下面向对象的程序设计思想和 C++语言的一些特点以及常用的程序设计思想，但限于篇幅有限，而且本书的宗旨不是介绍面向对象，因此这里面向对象不进行特别细致的描述，需要的读者可根据自身情况，参阅相关书籍和资料。

对于 C++的初学者，建议认真阅读本章，而对于已经熟练掌握 C++语言的读者可以跳过本章，从第三章开始阅读。

1 对象的基本概念

面向对象的设计思想是以对象为核心的，因此要更好的理解面向对象，就必须从对象的基本概念和基本特征了解起。

1.1 对象的现实含义

在面向对象的程序设计中，必须首先弄清楚什么是对象，在英文里面对象是 Object，可以翻译为目标、物体，因此对象可以理解为一个物体、一件事情、甚至是一些非常抽象的事件。如：一张桌子、一辆汽车、一个车队等等，都可以理解为对象，结合到程序中则可以是一个按钮、一个菜单、一个图形等等。

对象可以是简单的，也可以是相对复杂的事物，比如车队是由若干辆汽车组成的，而汽车本身又是一个对象，这两个对象之间有着密切的联系，却又有各自的特征，而且汽车进一步划分可以由车轮、车厢等组成，车轮、车厢又是更小更细的对象。再比如：就汽车这个对象而言，可以象上一章那样，只关心几个主要的参数和一些主要的操作，也可以非常细致描述到一个螺丝钉、车体的材料、车轮的大小、车身的颜色、轮胎的花纹、开车、修车、转弯、刹车等等。这是根据我们实际的需要来定的。

我们可以这样来理解对象：

- 对象是经过抽象生成的，是将一些联系比较密切的事物整合在一起而形成的。
- 和其相关的一些操作也作为对象的一部分，和对象保持着一定的关系。
- 对象是经过抽象生成的，可以完整的、相对独立的描述该事物。
- 根据描述的细致程度不同，对象允许是相互嵌套的。

比如汽车是车，火车也是车，它们都属于“车”这个对象，但各自的内涵取不同，它们的共性就是“车”这个对象的内涵。也就是车包含着“汽车”、“火车”等不同的种类，这些不同的种类都具备“车”的基本特征。

对象也可以认为是由不同的更小的对象联合而成的，由多个联系比较密切的对象联合在一起也可以组成一个新的对象。

1.2 对象的划分和联系

和结构化程序设计相比，面向对象是一种全新的概念，因而不要试图将一个结构化程序一刀未剪或者稍加修改的“升级”为面向对象的程序，这是徒劳也是不可能的。面向对象的程序需要对数据进行重新分析和抽象，这就出现了如何对对象进行划分，以及实现对象与对象之间联系等问题。

从前面的内容我们也可以看出，对象的划分也并不是唯一的，也没有统一的标准，然而对象的划分的好坏却直接影响着一个程序的质量。一般来说，确定的对象应该是可独立的，即：

- 对象可以独立的描述一个事物或一个操作过程；
- 对象应该尽量的符合事物的客观规律，在结构上、联系上相对紧密和固定的一些元素可以作为一个对象；
- 确定对象时应该全面的对问题进行分析，一个对象应该有其明确的目的以及所能表达的问题；
- 对象应该尽量需求系统中事物的共性，将所有的共性确定为一个新的对象更符合思维的抽象过程。

比如：在一个矢量会图的程序中，将画布上的每一个“圆形”、每一根“直线段”都作为一个对象，那么线段这个对象有其“起点”、“终点”、“颜色”、“宽度”等特征，而圆则有“圆心”、“半径”、“颜色”等特征，将整个画布上的“对象”按照各自的特征显示在画布上，就可以构成我们的整个图形。

如果将画布上所有的“线段”作为一个对象，将画布上的所有的“圆形”也作为一个对象，那么划分这个对象将失去所有的意义，这张画布上的“所有线段”和另外一张画布上的“所有线段”可能具有的共同特点只有一个字“乱”。这样的划分方法也许还不如用结构化的程序设计。

对象在程序中是相对独立的，但并不是孤立的，也是不可能孤立的，但作为我们抽象出来的对象，其内部成员必然有和外界极少发生关系的，甚至不会发生关系的；也有一些操作是和外界不发生关系。这些特征我们完全可以通过一种方式将其隐含起来，这就是面向对象中的数据和过程的隐藏。通过这种技术可以使程序之间的关系明朗、简化，进一步使程序的设计和维护都大大简化。

仍然以车为例：

开车时我们并不需要知道变速箱中的齿轮是如何组织在一起的，也不需要亲自将这些齿轮摆放在正确的位置，而只需要将档位放置在适当的位置就可以了，调整齿轮是车子自己的事情，和操作者不会发生直接的关系，这就是被隐藏的一个过程，齿轮也是被隐藏起来的成员。

而相反的，对象中有一些成员或者操作是必须和外界发生关系的，比如车子的档位和调档这个操作。对象也正是通过这些公共的“接口”与外界联系在一起的。

了解对象的划分和联系对于程序设计是至关重要的，也是面向对象中最基本的概念。只有正确的划分了对象并建立对象之间的关系，才可能更好的利用面向对象的技术来建立高质量的程序，提高设计和开发的速度。接下来我们将要介绍如何在 C++语言中实现面向

对象的程序设计技术。

1.3 类和类的实例

面向对象是一种思想，在不同的计算机语言里，对其的描述和实现过程时有所不同的，在 C++语言里面向对象是以“ 类（Class）”为基础来实现的。

类是对数据和操作即为对象的抽象描述，具体的对象便是类的实例，在现实中我们有着抽象的概念“ 车子 ”，那么张三的车子、李四的车子，都是车子的一个实体，是一个具体的对象。在程序里面，这种关系对应的是“ 类 ”和“ 类的实例 ”。

具有“ 起点 ”、“ 终点 ”、“ 颜色 ”、“ 宽度 ”等特征的简单图形我们可以定义为“ 线段类 ”，那么画布中每一条线段都将是“ 线段类 ”的一个实例。

如何理解类和实例呢？就如同 C 语言中 int 是整数类型，float 是浮点类型，而 int i,j float x,y;其中 i 和 j 是整型变量，x,y 是浮点变量，那么 i,j 和 x,y 分别是 int 和 float 类型的实例。因此上讲：类是一种数据结构，是一个抽象的概念，而实例是一个现实数据，是实实在在存在的。这是两个不同的概念，也是初学者经常容易混淆的概念。另一个应该区分的是语言和开发环境，初学者往往也容易将其混在一起，开发环境是一种高度集成的编译器，而语言则是实现一定算法时所采用的描述方法。

在 C++语言里面，将类中描述类特征的数据称作类的“ 成员 ”或者叫“ 数据成员 ”，将类的特定的操作过程称作“ 成员函数 ”。成员描述的是类的构成特征，而成员函数是描述类的行为特征。

2 C++语言的扩充

C++语言是由 Bjarne Stroustrup 开发的，最初被称为“ 带类的 C ”，是在 C 语言的基础上扩充而形成的，后来人们在 C 后面加上 C++语言中的自增运算符++，而将其命名为 C++语言。通常 C++语言设计的程序源文件的扩展名为 .CPP（Plus），以区别 C 语言的源文件，这一点是很重要的，因为大多数 C++编译器在处理 C 源文件和 C++源文件时的方法和过程时有所不同的，这一点在我们后面稍作说明。

为了更好的理解 C++语言，这里有必要介绍一下 C++对 C 语言的一些重要的扩充，要获得更详细的内容，读者可以参考有关 C++语言的专门书籍。

2.1 类 Class

上面提到了类是 C++中实现面向对象的关键，而类 Class 也正是 C++语言中最重要的扩充。我们看下面一段 C++代码：

```
struct Point
{
    int  X;
    int  Y;
};
```

```
//-----------------------------------------------------
class Line
{
private:                                    //下面的成员为 Private 属性
    int   Color;                            //成员，记录颜色
public:                                     //以下成员为 Public 属性
    Point   StartPoint;                     //成员，起点
    Point   EndPoint;                       //成员，终点
    void  Show( void );                     //成员函数，显示图形
    void  Hide( void );                     //成员函数，隐藏图形
    void  ChangeColor( int NewColor );      //成员函数，改变颜色
};
//-----------------------------------------------------
void Line::Show()
{
  ……          //显示图形；
};
//-----------------------------------------------------
void Line::Hide()
{
  ……           //隐藏图形；
};
//-----------------------------------------------------
void Line::ChangeColor( int NewColor)
{
   Hide();                                  //先隐藏图形；
   Color = NewColor;                        //设置新的颜色值
   ……                                       //改变颜色；
   Show();                                  //设置颜色后重新显示图形
};
```

这正是 1.2 中对于“线段类”描述的代码，在这里面 struct 是 C 语言中本来就有的关键字，“结构”的标识符，而其他的 class、private、public 以及例子中未出现的 friend、protected、virtual、this 等都是 C++扩充的关键字，下面就构成类的及个关键字作以简单介绍：

2.1.1 class 关键字

class 是由 C 语言中 struct 发展而来，但它 struct 比增加了以下几点：

- 增加了对成员引用范围即作用域的限定。
- 增加了成员函数
- 增加了可继承性
- 增加了对自身的引用指针

除此之外，类还具备自动初始化等多种特性，读者可参阅相关书籍。

class 关键字用于定义新类，其格式如下：

```
class  类名称
{
private:
      [私有成员列表]
      [私有成员函数列表]
protected:
      [保护成员列表]
      [保护成员函数列表]
public:
       [公有成员列表]
       [公有成员函数列表]
};
```

[]中的内容表示非必要成员。

在类的声明中 private、protected、public 为成员访问的说明符，跟随在 private 符号后面的成员为私有类型，protected 后面的成员为受保护类型，public 之后的为公有类型。这几个关键字用于类的定义时，后面总要跟一个冒号（：）其范围为自冒号以后至下一个说明符之前，并且可以在类中多处出现。

在类的成员函数中，通常有一个或者多个与类的名称相同的函数，称之为“构造函数”（constructor），构造函数总是没有返回值的，即为 void 类型而且只用于对象的初始化过程，通常不允许用户直接调用，而是在建立一个类的新的实例时被系统自动的调用。如果没有显式的指明构造函数，系统会自己生成一个默认的构造函数，并且系统在生成一个新的对象时，会自动的将所有的成员初始化为“0”，因此，很有必要对一些成员在对象生成时进行一些特殊的初始化而制定构造函数。

除此之外，类中还包含另外一个特殊的函数——“析构函数”（destructor），同构造函数相同，析构函数和类名也相同，但函数名前面多了一个“~”符号；析构函数也没有返回值，但每个类中只能包含一个析构函数。析构函数用于对象被销毁时，来释放对象中的某些数据，并不是所有的类都必须包含显式的析构函数，如果类中没有用户自行分配的内存空间，则可以使用系统默认的“析构函数”来释放对象占用的资源。

利用 class 关键字，我们可以构建自己的对象，读者可以在本书的光盘中找到 2.1 中 Line 类的完整例子，这里不再多讲。

在这个例子中，类 Line 使用了 4 个不同的构造函数，分别有着不同的作用，在 C++

中允许多个函数具有相同名称，这在后面函数重载的内容中详细讲解。另外这个例子中使用了 new 和 delete 关键字，这也是 C++语言中增加的关键字，用于创建对象和销毁对象。

2.1.2 private、protected、public 关键字

private、protected 和 public 这三个关键字在类的声明中为成员访问控制说明符，在类声明中，这几个关键字后面都要跟一个“ :”表示作用域的开始，直到下一个访问说明符出现，表示该说明符的作用域结束。

在类的声明中，被声明在 private 域中的成员及成员函数由具有私有性质，不能被类以外的对象或函数访问，只能对该类的成员所访问。这样一来，可以使一些不常与外界发生关系的、或者比较重要的数据，被有效地保护起来，不会因为不当的操作而被修改。

定义在 protected 域中的成员具有被保护的属性，这类成员可以被派生类访问，也可以被友员访问，但并不是所有的对象或函数可以访问该类的成员或成员函数，protected 类型为成员提供了一种相对较弱的限制，但并没有完全限制，这对派生类需要访问父类的成员时非常有用的。

public 向外界提供完全开放的公共接口，任何对象或函数都可以访问一个类中被定义为 public 类型成员或成员函数。通常会把一些和外界有联系的或联系比较密切成员或成员函数声明为 public 类型,类的私有成员就是通过这些接口间接的和外界进行通讯的。

private、protected 和 public 除了在类的声明中说明成员的访问类型之外，还是类的继承类型的说明符。

继承性和类的派生是 C++的类中增加特性，派生指的是在已有类的基础上，对该进行更深层次抽象和描述，从而形成一个新的，比原来的类具有更广泛内涵的类，这个类就叫做派生类，原来的类叫做父类或者基类。

类的继承和派生可以大大地提高程序代码的可复用性，加快程序开发过程，也是符合人类的认识和抽象过程的。

比如对“ 汽车 ”这个类加上“ 用途 ”就可以生成一个新的类，这个类是通过“ 汽车 ”而派生的，将具有其“ 汽车 ”的所有共性，同时具有自己的个性应，父类的共性便是继承。如：

“ 小轿车 ”和“ 载重汽车 ”都是“ 汽车 ”的派生类，它们继承了“ 汽车 ”的所有特征，却也有各自不同的特征。

C++语言中的继承性正是这种抽象关系在计算机语言中的反映，我们将在后面对继承性作更多的讲述。这里先举一个派生类的例子：

在我们前面定义的类 Line 中，包含了起点、终点、颜色等信息，但并没有包含线段的线型和宽度，我们设想从 Line 派生一个包含线型和宽度的新类 NewLine，在 C++中可以这样实现：

```
class NewLine : public Line          //声明NewLine类，从Line类共有继承
{
private:
```

```
public:
    int  LineStyle;                    //增加 LineStyle 成员
    int  Linewidth;                    //增加 Linewidth 成员

    void  Show();                      //定义新的 Show()成员函数
    void  Hide();                      //定义新的 Hide()成员函数

    void  NewLine( Point StartP, Point EndP, int Style, int Width );
                                       //NewLine 的构造函数
    void  ~NewLine( void );            //NewLine 的析构函数
};
//-----------------------------------------------------
void NewLine::NewLine(Point StartP, Point EndP, int Style, int Width)
   : Line( StartP, EndP )              //增加了父类构造函数的调用
{
    LineStyle = Style;
    Linewidth = Width;
    Show();
}
//-----------------------------------------------------
void NewLine::Show()
……
void NewLine::Hide()
……
void NewLine::~NewLine()
……
```

在这个新类中我们增加了两个共有成员 LineStyle 和 Linewidth，并且有了新的构造函数和析构函数，同时，由于新的类中 Show()和 Hide()的处理方法和 Line 中的有所不同，因此在 NewLine 中也重新定义了 Show()和 Hide()，但是问题也随之而来，类 Line 中的 Show()和 Hide()同样被 NewLine 继承，这使得在 NewLine 中出现了两个版本的 Show()和 Hide()，一个是自己的，一个是从父类继承而来的，函数名称相同在 C++语言中是允许的，但在调用这两个函数时到底应该是那一个版本被调用却成了一个问题，这涉及到了多态性及虚拟函数等概念，我们在后面作以介绍。

在 NewLine 的声明中，与 Line 的声明不同，class NewLine 后面多了一些内容“：public Line”，这表示 NewLine 类是由 Line 共有派生而来。同样 private、protected 也都可以作为继承类型的说明符，但它们继承来的成员却会产生不同的影响。

被标记为:public 型式继承的父类，其成员可以访问类型会被原封不动地继承到派生类中 标记为:protected 型式继承的父类,其 public 域的成员被继承为 protected

类型的成员，protected 和 private 域的成员仍然保持原来的性质不变；而标记为:private 型式继承的父类，其所有的成员均被按照 private 类型继承到派生类中。

2.1.3 friend 关键字及友员

friend 关键字用于声明友员函数和友员类，友员函数和友员类都是宿主类的作用于以外定义的，但可以访问该类的私有成员和受保护的成员。友员关系是单向的、非对称的、而且不可传递，但友员关系是可以继承的。

比如有两个类 A、B，若声明 A 为 B 的友员类，则 A 类中的成员函数可以访问 B 类中的私有数据和受保护数据，但反过来 B 类不能访问 A 类中的私有数据和受保护数据。若类 B 又是另外一个类 C 的友员类，则 A 和 C 之间不存在友员关系。友员的关系是在宿主类中定义的，是“给予”的而不是“索取”的。

友员关系的声明形式如下：

```
class A                              //定义类A
{
   ……
}
//-----------------------------------------------------
class B
{
    ……
    friend  class A;                 //说明类A为B的友员类
    friend  void   FuncA();          //说明FuncA为B的友员函数
}
```

友员的声明不会受到 private、protected 和 public 这三个关键字的限制，可以出现在类定义的任何位置。

友员为 C++的私有数据提供了访问的接口，但同时也破坏了信息隐藏的特性，破坏了 C++的封装性。因此，并不主张频繁的使用友员关系，在 C++ Builder 里面，尽管仍旧支持 friend 关键字，但几乎不再使用友员关系。在对类的定义合理的情况下，也完全可以避免频繁的使用友员关系。

2.1.4 virtual 关键字

C++语言里面允许在同一个程序或者原文件中有相同的函数名称，在一个类里面也可以有不同参数列表但函数名称相同的成员函数，这就是所谓的多态性，但这也带来了一个新的问题，就是如何判断调用适当版本的函数。如我们上面例子中的 Line 和 NewLine 中都有 Show()和 Hide()，如果用 new 关键字生成一个 NewLine 的实例，

```
Line1 = new NewLine( StartP, EndP, Style, Width );
```

那么下面这条语句的调用是会出现问题的，

```
Line1->Show();
```

为了解决这一矛盾，在 C++里我们引入了虚函数的概念，虚函数的定义只需要在函数的声明前面加上一个 virtual 关键字，虚函数是一个全新的概念，关于它的其它特点，我们在多态性中详细讲述。

2.1.5 this 关键字

在 C++语言里，一个类可以由多个不同名称的实例，在程序运行时，这些实例的数据是独立存放的，但和其相关的成员函数在内存中只有一份拷贝，所有相同类的实例都是共享其成员函数的代码。

this 指针是类中一个特殊的指针，它始终指向对象自身的，每一个成员函数中，虽然没有明确指定引用那一个实例的成员，但程序也会引用正确的对象的成员，这是因为，在类中 this 始终作为一个隐含的参数出现的，当然程序员也可以显式的使用 this 指针。

比如：

```
class A
{
   int X;
   int Y;
   int SumXY(void )
        {
           return X + Y;
        }
};
//-----------------------------------------------------
A* A1;
A* A2;
A1 = new A();
A2 = new A();
int SumA;
SumA = A1->SumXY();                    //求对象 A1 中 X、Y 的和
SumA = A2->SumXY();                    //求对象 A2 中 X、Y 的和
```

在成员函数 SumXY()中有对成员的引用(X+Y)，它和(this->X + this->Y)是等价的，this 指针被隐含的包括在对成员的引用里面，在 A1->SumXY()调用时，this 将指向 A1，而在 A2->SumXY()调用时，this 指向对象 A2。因此，A1->SumXY()得到的是 A1->X 与 A1->Y 的和，而 A2->SumXY()得到的是 A2->X 与 A2->Y 的和。

这只是 this 指针的一个用途，也是非常重要的一个用途，一般不需要显式的使用

this 指针，在某些情况下，程序员必须明确的使用 this 指针来完成一些操作，比如在 C++ Builder 的 VCL 组件里面，要求有一个父组件(Parent)，如果程序运行时动态生成一个运行时可视的组件，像 TButton、TImage 等，就必须指明父组件(Parent)，否则该组件不会显示在屏幕上。通常我们用以下代码解决：

```
Form1::CreatButton()
{
    Tbutton * Button = new Tbutton( NULL );    //创建一个新的 Button
    Button->Parent = this;                     //指明 Button 的 Parent 为 Form1
};                                             //新建的 Button 将显示在 Form1 上
```

2.2 ::作用域运算符

在 C++语言中，作用域是一个很重要的概念，因为在一个程序中允许出现相同的函数名称，作用域对变量和函数都是有效的。C++中作用域运算符为两个紧连着的冒号(::)，运算符前面为作用域的范围或作用于名称，后面为变量、成员、成员函数等。

如我们在 Line 类中定义了成员函数 Show（）的原型，在后面的实现代码中，必须注明 Line::Show（），如果省略了::和前面的 Line，编译器将认为该函数是公共域中的 Show（）函数，因而编译时会报错。

在成员函数中也可以通过::运算符对公共域中的变量进行引用，只要将::运算符前面的参数为空就行。

2.3 new、delete

new 和 delete 为 C++语言中另外两个重要的关键字。面向对象编程时，很多情况对象是运行时创建的，因此系统必须为其分配适当的内存空间，在 C 语言里面，动态分配内存用 malloc 和 free 函数，但将其用在 C++里面便有一些笨拙，不够灵活。

首先是 malloc 分配内存时必须指定所需内存的空间大小；而且在早些版本的 C 语言中分配的内存返回为 void 类型的指针，创建类时还要进行类型转换；最令人头疼的是，在 C++中一个类的生成，往往需要进行一些初始化工作，用 malloc 只能为其分配内存，而无法自动的初始化对象。因此在 C++中引入了 new 和 delete 关键字。

new 关键字在使用时也非常方便，我们前面用 new 创建 Line 的实例：

```
Line1 = new Line( …… );
```

使用起来非常直观，也非常方便，更重要的是用 new 关键字创建 Line1 时，系统会自动调用 Line 的构造函数，并根据所带的参数选择正确的版本，完成对对象的初始化。

除了对类可以初始化之外，使用 new 关键字还可以对其它类型的变量进行初始化，比如一个整型变量：

```
int* IP = new int(0);
```

系统会创建一个指向整型变量的指针，并将其内容初始化为“0”。

将 new 关键字用于数组的创建也非常方便，比如需要创建一个 10 乘 10 的浮点类型

数组，使用以下代码就可以：

```
float* fPrt = new float[10][10];
```

使用 new 关键字创建的内存空间，必须使用 delete 关键字予以释放，相对应的，使用 delete 关键字释放内存时，也不需要指明内存空间的大小，即使数组也是一样。而且对于销毁一个对象，delete 关键字的使用同样会让系统自动的调用该类的析构函数，以释放对象的其他资源。

上面的几个用 new 关键字分配的内存可以使用以下语句释放：

```
delete Line1;                    //自动调用析构函数
delete IP;                       //销毁该数据
delete [] fPrt;                  //释放该数组所有元素占用的内存
```

new 关键字分配的内存必须用 delete 关键字释放，一个好的程序员应该经常检查是否有闲置的内存开销，并及时释放，以获得更大的系统资源。但并不是只有 new 分配的内存才可以使用 delete 释放，在经典的 C++里面是这样的，但在 C++ Builder 中并不是这样，比如：Form 往往是由系统自动创建的，或者使用 CreatForm 也可以创建，new 当然也可以创建，但都可以使用 delete 销毁，一些设计时创建的可视化组件也是可以用 delete 销毁的。

2.4 &引用

在 C 语言中函数调用的参数都属于按值传递，也叫做形式参数，即函数调用时，会将所有参数拷贝一份并交给被调用的函数使用，这样被调用的函数可以随意的改变参数的值而不担心会无意中修改了原来参数的值。但是，任何函数的返回值只有一个，因此有些时候希望多个被传递的参数在经过处理后，能将处理后的值返回给调用者，就只能用指针来完成，而且 C 语言的这种调用方式在将一个类或者结构等体积较大的变量作为参数时，系统将不得不花费额外的时间来复制参数。在 C++语言里，引入了其它一些高级语言中的引用调用的概念，实现了类似实参数调用方法，从而大大地提高了程序的灵活性和实用性。

引用的说明符为&，比如定义一个引用的变量如下：

```
int i;                           //定义一个整型变量 i
int &y = i;                      //定义一个整型引用 y（i 的引用）
```

定义一个引用参数的函数如下：

```
bool Exchange(int &X, int &Y)    //返回值为整型，参数为整型引用 X,Y
{                                //交换 X,Y 值
    int Temp;
    Temp = X;
    X = Y;
    Y = Temp;
```

```
    return true;
};
```

需要调用该函数可以是如下的代码：

```
int i, j;
bool Ret;
Ret = Exchage(i, j);
```

同样的功能如果采用 C 语言可能需要如下的代码：

```
bool Exchange(int *X, int *Y)            //返回值为整型，参数为整型指针X,Y
{                                        //交换X,Y值
    int Temp;
    Temp = *X;
    *X = *Y;
    *Y = Temp;
    return true;
};
//-----------------------------------------------------
int i, j;
bool Ret;
Ret = Exchage( &i, &j );                 //这里的&是取地址运算符
```

实际上这两段的代码在编译后生成的可执行代码是完全相同的，但在书写和表达上，显然采用引用&要方便的多。

需要注意的是不要将 C 语言中取址运算符“&”和 C++中的引用“&”相混淆，尽管这两者有许多相似之处。

取址运算符只会出现在“=”的右边或者是在函数调用时传递参数，如：

```
int *p = &i;
```

的含义是将变量 i 的地址传给整型指针变量“p”。

```
Ret = Exchage( &i, &j );
```

的含义是将 i 和 j 的地址作为参数传递给函数 Exchage。

而引用说明符“&”只能出现在函数声明时或者变量声明时，如：

```
int i;
int &y = i;
```

的含义是定义一个整型引用 y，并指定 y 为 i 的引用，绝对不可能是将 i 的值传递给 y 的地址，单纯的 int &y 是不合法的。这里 y 只是 i 的一个别名，在内存中也并不会为 y 另外的开辟空间。引用更多的是用在函数的声明中：

```
bool Exchange(int &X, int &Y);
```

表明 Exchage 的两个参数均为整型引用，在这里似乎按照取址运算符理解也是正确的：&X 和&Y 作为 X 和 Y 的地址被当做形式参数传递给 Exchange，函数体内直接操作 X、Y，&为取址运算符是*指针运算符的逆运算，但实际并不是这样，如果按照这样的理解，那么函数调用时应该是：

```
int i, j;
bool Ret;
Ret = Exchage( &i, &j );
```

而不应该是：

```
Ret = Exchage( i, j );
```

在面向对象的程序设计中，很多时候都需要使用引用作为函数的参数。同指针一样，引用也可以作为函数的返回值传递给调用者，但必须注意的是，如果返回的是该函数局部变量的引用，则该变量必须在函数体内用 static 声明为静态变量，否则在函数返回时，该变量实际上已经被销毁，而该变量的引用也将不可预知，处理不当将会给程序带来灾难性后果。需要作更多的了解，请读者翻阅相关资料。

2.5 inline 关键字

inline 也是 C++扩充的一个关键字，它主要用来声明所谓的“内联函数”。在 C 语言中，如果有一个体积非常小用的函数又在多处使用，为了减小调用的时间开销，通常使用宏定义函数，这样编译器会按照宏定义进行文本替换而将这一段代码插入到相应的位置。但是采用宏定义函数仅仅是进行简单的文本替换，如果有这样的代码：

```
#define SQUARE( X )   X * X
SQUARE (3 + 2);
```

在编译时会被展开为：

```
3 + 2 * 3 + 2
```

这显然与最初的目的是不相同的，而在 C++中可以使用内联函数：

```
inline double Square( double X );
```

采用内联函数，也可以象宏那样将短小的函数代码直接联编嵌入在调用它的位置，但内联函数本质上还是一种合法的 C++函数，因此编译器会对内联函数进行正常的语法检查和类型检查，这是宏所不具备的。

2.6 const

在 C++中另外一个非常有用的关键字是 const。用这个关键字可以声明“常量变量”，这是一个矛盾的叫法，实际上更确切的叫法应该是“只读变量”，意思说被 const 修饰的

变量除了不能被改变之外，和其他变量是一样的。const 关键字也可以在函数声明时，指定常量参数。

使用 const 关键字可以定义常量变量来替代在 C 语言中常用的宏定义常量，常量变量可以和其他变量一样的被程序所识别和应用，也可以进行取地址等操作。此外使用 const 关键字还可以定义常量指针，如：

```
int i;
int * const ip = &i;
```

或者：

```
const int *ip = &i;
```

注意：这两者定义方法的含义是不同的，int * const 的含义是一个整型的常量指针，即 ip 是不可以改变的，但它所指向的变量是可以被改变的。

```
*ip = 8;
```

是合法的，但是 ip 的值不能够被改变，而 const int *的含义是定义一个指向整型常量变量的指针，尽管变量 i 在定义时并没有声明为常量变量，但是被 ip 引用后，将不能通过*ip 来改变它的取值，但是 ip 的取值是可以被改变的：

```
*ip = 8;
```

是不合法的，而

```
ip = &j;
```

是合法的。

从这也可以看出，const 关键字修饰紧跟其后内容：在 int * const ip 中 const 修饰 ip,因此 ip 是一个不能改变的变量；在 const int *ip 中 const 修饰 int，因此 ip 可以被改变，但是它是一个指向“const int”类型的指针，所以*ip 的内容不能被改变！

编译器可以识别不同类型的指针和变量，从而使用 const 可以有效地保护某些数据在传递给其他函数时不会被非法更改。

C++语言除了这些以外还增加了许多有用的扩充，包括语法的、书写格式上的等等，而且不同的 C++编译器也可能会有一些生产厂商对其的扩充。这里仅仅就一些 Ansi C++主要的扩充内容作以简单介绍，读者可参阅相关书籍做更深一步了解。

3 C++中对象的封装性

封装实际上就是信息隐藏，在面向对象的系统中封装是构成对象的处理过程，一个封装的对象也常被称之为一个抽象的数据类型，对象没有封装性，实际上就不能构成真正的面向对象的程序设计。

前面我们已经提到过，在 C++语言中封装是通过类 class 来实现的，并且举了一个简单的例子来说明 class 关键字的一些基本用法。

3.1 对象的封装

面向对象系统的封装性主要是通过对象的封装性来体现的。对象的封装性是指将对象的属性(数据内容)和作用于这些属性上的操作封装在一起，对象的使用者只能通过提供它的接口(类声明)使用这个对象。对象的内部是被保护起来的，它的使用是通过调用公有成员函数完成的。下面我们看一个没有封装概念的例子：

```
#include <graphics.h>
#include <string.h>
#include <conio.h>
#include <stdlib.h>
//-------------------------------------------------------
struct square
{                          //定义一个正方形结构
   int  x, y;              //定义中心坐标
   int  length;            //定义正方形边长
};
//-------------------------------------------------------
struct circles             //定义一个圆形结构
{
   int  x,y;               //定义圆心坐标
   int  length;            //定义圆直径
};
//-------------------------------------------------------
int sarea(int length)    //计算正方形面积
{
   return length*length;
}
//-------------------------------------------------------
float carea(int length) //计算圆面积
{
   return 3.1416*length*length/4;
}
//-------------------------------------------------------
void sshow(int x,int y,int length)
{    //以 z,y 为中心,以 length 为边长在屏幕上画一个正方形
   rectangle(x-length/2,y-length/2,x+length/2,y+length/2);
}
//-------------------------------------------------------
```

```
void cshow(int x,int y,int length)
{    //以x,y为圆心坐标,以length为直径在屏幕上画一个圆
   circle(x,y,length/2);
}
//-----------------------------------------------------
main()
{
   char *str, *s1;
   square s;            //定义一个正方形结构变量s
   circles c;           //定义一个圆形结构变量c
   s.x = 200;           //为正方形的各分量赋值
   s.y = 200;
   s.length = 100;
   c.x = 200;           //为圆形的各分量赋值
   c.y =  200;
   c.length = 70;
   int gdriver = DETECT,gmode;
   initgraph(&gdriver,&gmode,"..\\bgi");    //置图形工作模式
   setcolor( 4 );
   sshow(s.x,s.y,s.length);       //在屏幕上画正方形
   setcolor(14);
   cshow(c.x,c.y,c.length);       //在屏幕上画图
   itoa(sarea(s.length),str,10);  //求正方形面积,并将其转换成字符串str
   s1 = "square area:";
   strcat(s1,str);                //将两个字符串合并
   outtextxy(20,400,s1);          //显示字符,即正方形的面积
   gcvt(carea(c.length),10,str);  //求圆面积,并将浮点型结果转换成字符串str
   s1 = "circle area：";
   strcat(s1,str);
   outtextxy(20,440,s1);          //显示圆面积
   getch();
   closegraph();                  //关闭图形模式
   return 1;
}
```

在上面的例子中有两个结构——正方形结构和圆形结构，对两个结构分别进行的操作——显示操作和求面积操作的定义都是全局的，任何函数过程都可以对它们进行访问。它们没有被封装隐藏起来。这样就容易造成错误，例如在主函数中我们定义了正方形 s 和圆形 c，由于它们的分量都是 x，y，1ength，就不免要出现错误。再如在调用正方形显示

函数时，写成如下形式：

```
sshow(s.x, s.y, c.1ength);
```

在这里 s.1ength 用成了 c.1ength,表示用圆形变量的直径作为正方形边长画一个正方形，这是一个严重的错误，并且是没有任何异议的做法。但是程序在运行过程中非但不拒绝，其结果看上去也没有什么不正常。

上述这些都说明了在程序中对数据和操作设置封装保护机制的必要性。

在有了封装机制以后，我们把数据和对数据的操作封装在一个类中，除去一些与外界的接口外，类中的内容是被隐藏保护起来的，外界要使用类是通过接口来完成的。

上面的例子我们将它改为类来做：

```
class square
{   //定义一个正方形类
private:
    int x, y;
    int length;
public:
    square(int vx,int vy,int vlength);
    int area();
    void show();
};
//-----------------------------------------------------
class circles    //定义一个圆类
{
private:
    int x, y;
    int length;
public:
    circles(int vx,int vy,int vlength);
    float area();
    void show();
};
//-----------------------------------------------------
square::square(int vx,int vy,int vlength)
{   //正方形类的构造函数,给类中数据成员初始化
    x = vx;
    y = vy;
    length = vlength;
```

```
}
//-----------------------------------------------------
circles::circles(int vx,int vy,int vlength)
{   //圆类的构造函数,给数据成员赋初值
    x = vx;
    y = vy;
    length = vlength;
}
//-----------------------------------------------------
int square::area()
{   //求正方形面积,不需传递参数
    return length*length;
}
//-----------------------------------------------------
float circles::area()
{   //求圆面积
    return 3.1416*length*length/4;
}
void circles::show()    //在屏幕上画圆
{
    circle(x,y,length/2);
}
void  square::show()
{
    rectangle( x - length/2,y - length/2,x + length/2,y + length/2 );
}     //在屏幕上画一个正方形

main()
{
    char *str1,*s1,*str2,*s2;
    square s(200,200,100);    //创建一个正方形对象,并初始化
    circles c(200,200,70);    //创建一个圆形对象,并初始化
    int gdriver = DETECT,gmode;
    initgraph(&gdriver,&gmode,"..\\bgi");
    setcolor(4);
    s.show();                 //画正方形
    setcolor(14);
    c.show();                 //画圆
    setcolor(5);
```

```
    gcvt(s.area(),10,str1);  //求正方形面积,并将其转换成字符串 strl
    s1 = "square area:";
    gcvt(c.area(),10,str2);  //求圆面积,并将其转换成字符串 str
    s2 = "circle area:";
    strcat(s1,str1);         //将两个字符串合并
    strcat(s2,str2);
    outtextxy(20,400,s1);    //显示字符,即正方形的面积
    outtextxy(20,440,s2);    //显示圆面积
    getch();
    closegraph();            //关闭图形模式
    return 1;
}
```

在以上程序中，将正方形的中心坐标、边长以及求面积和画正方形操作封装在一个正方形对象中，将圆的圆心坐标、圆的直径以及求圆面积和画圆操作封装在一个圆对象中。关于对象的数据内容，我们创建对象时传递给构造函数参数使其对对象进行初始化，而在后面的求面积及显示图形操作中均不需传递参数，函数从对象的数据成员中取值，外界不能直接访问对象的私有内容，而只有通过调用公有成员函数来间接访问，因此对象是封装是比较完善的，这样做可以避免了很多犯错误的机会。

3.2 友员的使用

前面已经提到过 friend 关键字以及友员的概念。之所以引入友员是由于：一个对象的私有数据和方法只能在类定义的范围内使用，也就是说只能通过它的成员函数来访问。对象的这种数据封装和数据隐藏使对象和外界以一堵不透明的墙隔开。因此要求设计者确保每个类提供足够的方法对所有遇到的情况进行处理，这会给软件设计者增加了负担。

而数据隐藏给两个类共享同一函数或数据带来了额外的开销，这是因为每次访问这些共享内容都需要通过函数调用来完成。如果这种访问不经常发生，这类开销还能接受；如果经常发生，就会变得难以容忍。

因此，就须寻求一种途径使得类以外的对象或函数能够访问类中的私有成员，在 C++ 中就用友元作为能够实现这个要求的辅助手段。C++中的友元为封装隐藏这堵不透明的墙开了一个小孔，外界可以通过这个小孔窥视内部的秘密。

然而使用友元会使数据封装性受到削弱，导致程序可维护性变差。实际上友员已经违背了最初提出面向对象目的，但是在最初的 C++语言中，友员却为程序设计工作者带来极大的方便，随着技术的不断发展，这种特殊的关系在实际中用的机会日益见少。

3.2.1 友元的说明和定义

只要将外界的某个对象说明为某一个类的友元，那么这个外界对象就可以访问这个类对象中的私有成员。

声明为友元的外界对象既可以是另一个类的成员函数，也可以是不属于任何类的一般的函数，还可以是整个的一个类（这样，此类中的所有成员函数都成为友元函数）。

友元声明包含在其私有成员可被作为友元的外界对象访问的类定义中，也就是将 friend 关键字放在函数名或类名的前面。此声明可以放在公有部分，也可以放在私有部分，其结果是相同的。例如：

```
class point
{
private:
    int x,y;
public：
    point(int, int);
    friend void print();
    friend OtherClass::OtherFun();
}
```

在这个例子中，void print()并不是 point 类的成员函数，而是它的友元，它可以访问point类对象的私有成员x,y。另一个友员是类OtherClass的成员函数OtherFun，与 print 类似，只是在函数前面加上作用域的范围。

3.2.2 友员的使用示例

下面的例子使用了一般函数 AddXY 和类 TFriend 作为类 TFirstClass 的友员，由此可以看出，使用友员可以访问一个对象的私有成员。代码如下：

```
#include "conio.h"
#include "iostream.h"
class TFirstClass;

int __fastcall AddXY(TFirstClass &Cls);
class TFriend
{
public:
    int __fastcall Multip(TFirstClass &Cls);
};

class TFirstClass
{
private:
    int x;
```

```
    int y;
public:
    __fastcall TFirstClass(int Ox, int Oy);
    friend int __fastcall AddXY(TFirstClass &Cls);
    friend class TFriend;
};
int main(int argc, char* argv[])
{
    TFirstClass  Demo(5,6);
    TFriend     Frnd;
     cout<<"TFirstClass Object Demo, X = 5, Y = 6\n";
     cout<<"This is Use Friend Function AddXY \n";
    cout<<"The Sum of Demo.x &Demo.y Is: "<<AddXY(Demo);
    cout<<"\n\n";
    getch();
     cout<<"This Is Use TFirend Class's Member Fuction Multip\n";
     cout<<"The Meyilp of Demo.x & Demo.y Is: "<<Frnd.Multip(Demo);
    getch();
    return 0;
}
__fastcall TFirstClass::TFirstClass(int Ox, int Oy)
{
    x = Ox;
    y = Oy;
}
int __fastcall AddXY(TFirstClass &Cls)
{
    return Cls.x + Cls.y;
}
int __fastcall TFriend::Multip(TFirstClass &Cls)
{
    return Cls.x * Cls.y;
}
```

程序的运行结果为：

```
TFirstClass Object Demo, X = 5, Y = 6
This is Use Friend Function AddXY
The Sum of Demo.x &Demo.y Is: 11
```

按任意键后显示：

```
This Is Use TFirend Class's Member Fuction Multip
The Meyilp of Demo.x & Demo.y Is: 30
```

在按任意键程序结束，注意在这个例子中，只能通过友员来访问 TFirstClass 的私有成员 x、y 因此在 TFriend 的成员函数中只能以 TFirstClass 作为参数，而不能直接将 TFirstClass::x 和 TFirstClass::y 作为参数。

4 C++的继承性与类的派生

类确实提供了良好的模块分解技术，通过信息隐藏，将它们的界面与实现清楚地分开。但是，仅仅这一些是不够的，人们更希望能在类的基础上取得可重用性和可扩充性的目标。

要设计可重用性模块，任何方法都必须面对重复和差别，为了避免一再地重写同样的代码，引入不一致的错误，就必须进行抽象，抽象出一般特性后，还需在此基础上扩充其特殊的功能，使之能表达具体的事物。这就是面向对象中的派生与继承，继承和派生是一对相辅相成的关系，

继承是 C++面向对象程序设计的重要特性之一，它是指建立一个新的派生类，从一个或多个先前定义的基类中继承函数和数据，而且可以重新定义或加进新的数据和行为，从而建立了类的层次或等级。

4.1 类的派生

在了解派生类的概念之前，我们先看一下人们对现实中的事物抽象的过程。现实世界中许多实体之间并不是孤立的，它们具有共同特征，也有细微的差别，人们可以使用层次结构来描述它们之间的相似点和不同点。例如鱼的分类，如图 2.1 所示。

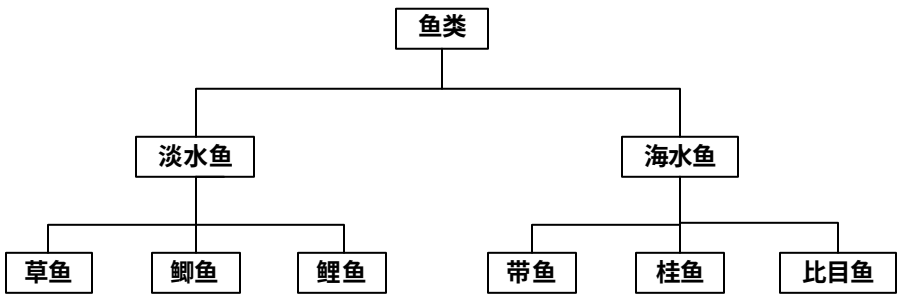


图 2.1 鱼分类图

这是树状的分类结构，这棵树反映了鱼类的派生关系。最高层是最普遍的最一般的，越往下反映的事物越具体，内涵越丰富，并且下层都包含了上层的特征。这中关系也是符合人类认识事物的过程，层次越高的分类，内涵越少，包括的事物越多，层次越低的分类越具体，内涵越多，个性更明显。这就是一种基类和派生类的关系。

派生和继承总是联系在一起的，例如确定某一条鱼是带鱼以后，没有必要指出它是生成在海水中的，因为带鱼本身就是从海水鱼类中派生出来的，它继承了这一特性；同样也

不必指出它会游，因为凡是鱼，都会游。带鱼是从海水鱼类中派生而来，而海水鱼又是从鱼类派生而来，因此带鱼也可以继承鱼类的一般特性。

从上面的例子可以看出派生和继承是对象类间的一种相关关系，具有以下性质：

- 表达了类间的共同特性；
- 体现不同类间的区别；
- 存在这类间的层次结构。

4.1.1 为什么使用继承

引入继承为代码重用提供有效手段。继承的优点主要有以下两个方面：

一方面使用继承可以重用先前项目的代码，如果原来的代码不能完全满足要求，还可以做少量的修改，满足不断变化的具体应用要求，从而提高程序设计的灵活性，避免不必要的重复设计。一般情况在选择了一种编译器之后，编译器会将许多常用的、提供特定功能的类提供用户，这就是通常称作 FrameWorks 框架。

另一方面若某个项目使用了几个非常相似或稍有不同的类，就可以通过派生类的继承性达到函数和数据继承的目的。

下面举例说明使用继承优点，有一个字符串类，可以如下定义：

```
class String
{
    int Length;
    char *Data;
public：
    String();
    ~String();
    int GetLength();
    char *GetContents();
    int SetContents(int DataLength，char *TheData);
    int SetContents(char *TheData);  //假如现在要定义的是一个可编辑的字
                                     //符串类，在一般情况下，我们将这样定义：
class EditString
{
    int Length;
    int Position；
    char *Data;
public：
    EditString();
    ~EditString();
    int GetLength();
    int GetPosition();
```

```
    void SetPosition(int ThePosition);
    char *GetContents();
    int SetContents(int DataLength, char *TheData);
    int SetContents(char *TheData);
    int Insert(char *TheText,int Pos);
    int Replace(char *Sour,char *Dest);
    int Delete(int Pos,int Count);
};
```

从上面两个类的定义中看出，这两个类中的数据成员和函数成员有许多相同的地方，有一些 EditString 类中的函数实际上只需将 String 类中的稍加修改即可；还有一些成员函数在两个类中完全相同。这是一个简单的例子，实际中的情况比这个例子要复杂的多，因此重复的代码也要严重的多，在引入了继承以后，代码的利用将会大大提高。后面的内容将讲解如何实现类的继承。

4.1.2 派生类的定义

前面已经提到继承和派生关系是相互依存的，只有派生而来的类才能够继承被派生类的特性。C++提供了从现有类派生出一个新类的能力，所派生的新类具有下列特性：

- 新的类可在基类所提供的基础上包含新的成员；
- 在新的类中可隐藏基类的任何函数成员；
- 可为新的类重新定义成员函数。

类的派生格式如下：

```
class x
{
    //...
};
class y : x
{
    //...
};
```

其中类 y 是从类 x 派生而来的，我们将 x 称为 y 的基类，也可以称作超类或父类；y 是 x 的派生类，也称子类。上面 String 和 EditString 类如果使用派生，可定义如下：

```
class String
{
   //String 类不变
};
```

```
class EditString : String        //EditString 从 String 派生而来
{
   int Position;
public：
   EditString();
   ~EditString();
   int GetPosition();
   void SetPosition(int ThePosition);
   int Insert(char *TheText,int Pos);
   int Replace(char *Sour,char *Dest);
   int Delete(int Pos,int Count);
};
```

从这里可以看出,EditString 的声明比原来简单了许多,实现的代码相应也会简化，但是具备着和原来相同的功能。再如有一个汽车类，被定义如下：

```
class Vehicle
{
   int Wheels;              //表示车轮数
   float Weight;            //表示汽车重量
public:
   void Init(int TheWheels,float  TheWeight);   //给数据成员初始化
   int GetWheels();         //取车轮数
   float GetWeight();       //取车重
};
```

现要定义一个小车类 Car，其中很多成员可以是继承 Vehicle 类的，此外还具有它自身所特有的性质，故将 Car 类定义成 Vehicle 类的派生类，如下：

```
c1ass Car : Vehicle
{
   int Person;      //允许载客数
public：
   void Init(int TheWheels，float TheWeight，int ThePerson = 4)；
                            //初始化函数
   int GetPerson();         //返回载客数
};
```

如果再有一个卡车类 Truck，同样可以定义成 Vehicle 的派生类，因为卡车和小汽

车都具有车的共同属性，但是卡车和小汽车各自又都有其特点。Truck 类要从 Vehicle 类派生，只需要根据卡车的特点增加新的成员和成员函数等，比如加一个载重量，因为一般卡车是用来装运货物的。如下：

```
class Truck : Vehicle
{
   float CarryWeight;
public：
   void Init(int TheWheels,float TheWeight,float TheCarry);
   float GetCarryWeight();
};
```

4.1.3 派生类对基类成员的访问权

前面讲过，类中的成员由两部分组成，即私有成员和公有成员。私有成员是收到类保护的成员，不能被外界访问，但派生类和基类又有着密切的关系，这就出现了派生类对基类成员访问权限的问题。

派生类并不是对基类中的所有成员都可以无条件地进行访问的，这个问题上在 C++语言中将类的派生类按照“外界”来处理，也就是说派生类不能够访问基类的私有成员。但是派生类可以访问基类的受保护成员，可以完全访问基类的公有成员。从这里也可以看出 protected 似乎是为了继承和派生而引入的，处于 protected 定义域的成员可以被其派生类访问，但不能被外界所访问，是界于 private 和 public 之间的一种访问限制。

对类的派生同样可以分作几种不同的派生类型，C++中提供了将 public、private 和 protected 关键字用于派生关系用法，因此派生关系也有三种：

- 私有派生
- 保护派生
- 公有派生

需要指定派生类型时,只需要在基类的前面使用 public、private 或者 protected 关键字就可以，如：

```
class Base
{
};
class Driv : public Base
{
};
```

就表示 Driv 从 Base 公有排生而来。这几种不同的派生关系，对基类的成员访问权限也有所不同。

1．私有派生

前面所定义的派生类都是私有派生。由私有派生得到的派生类，对它的基类的公有成

员只能是私有继承。也就是说基类的所有公有成员都只能成为私有派生类的私有成员，这些私有成员只能被派生类的成员函数访问，而派生类的使用者无权访问。

2．保护派生

和私有类似，基类的私有成员被继承到派生类中只能成为保护成员，基类的私有成员派生类不能访问，基类的保护成员被继承到派生类中访问属性不变，仍然为保护成员，由这个派生类派生的类可以访问。

3．公有派生

在公有派生中，基类的所有成员的访问属性被不变的继承下来，私有成员仍然为私有成员，且在派生类中不能访问，保护成员同样被继承为保护成员，派生类可以访问，派生类的派生类也可以访问，但是外界不能访问，公有成员仍为共有成员，可以被派生类和外界同时访问。

实际上这几种派生关系始终遵循这一个规则，就是访问控制权限就低不就高，下图给出了几个不同派生类关系的类，我们来分析一下它们的访问权限。

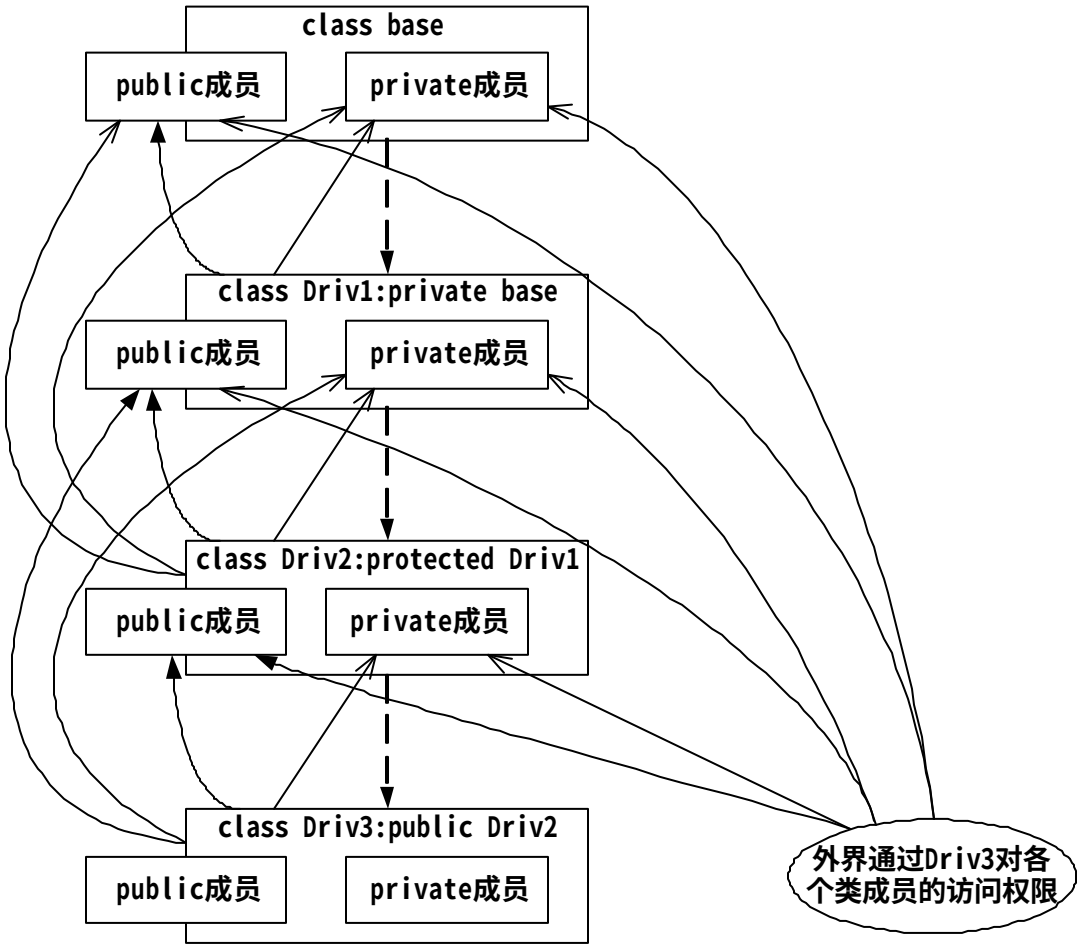


图 2.2 几种不同派生关系对基类的访问权限

图 2.2 中总共有四个类：Base、Driv1、Driv2、Driv3。虚线表示它们的派生关

系，曲线表示访问方向，实心箭头表示可以访问，空心箭头表示不能访问。我们来逐个分析：

Driv1 从 Base 私有派生而来，因此 Driv1 可以访问 Base 的公有成员，但是不能访问 Base 的私有成员，Base 的公有成员被继承到 Driv1 中将成为私有成员，不能被外界和 Driv1 的派生类访问。

Driv2 从 Driv1 保护派生而来，Driv2 可以访问 Driv1 的公有成员，不能访问 Driv1 的私有成员，同时也不可以访问 Driv1 继承 Base 的公有成员。Driv1 的公有成员被继承到 Driv2 中成为保护成员，不能被外界访问。但是可以被 Driv2 的成员函数或者派生类所访问。

Driv3 从 Driv2 公有派生而来，因此 Driv3 可以访问 Driv2 的公有成员和保护继承 Driv1 的公有成员，不能访问 Driv2 的私有成员和继承 Driv1 的私有成员即 Base 的所有成员。

当外界通过 Driv3 访问这几个类时，只能访问到 Driv3 的公有成员和 Driv3 公有继承 Driv2 的公有成员，Driv1 的公有成员被继承到 Driv2 中成为保护成员，因此也不能百外界访问。Base 的所有成员都不可能通过 Driv3 让外界访问到，显然外界对 Base 的成员在其派生类中比在自身中的访问权限更低，这就是因为访问权限就低不就高造成的。

4.1.4 派生类的构造函数和析构函数

面向对象的继承机制可以从一个类中派生出另一个类，而派生出来的类又可以拿去派生新的类，这就构成了类层次，建立了类等级。人们可以通过创建某个派生类的对象来使用这个类等级，包括隐含地使用基类的数据和函数。基类往往有构造函数和析构函数，当创建一个派生类对象时，怎样调用基类的构造函数对基类数据初始化，在结束派生类对象时，又如何调用基类的析构函数来对基类对余的数据成员进行善后处理，这涉及到派生类对基类的构造函数和析构函数的使用规则，不同的语言在类的构造和析构规则上不太一样，这里就 C++中的构造函数和析构函数作个简单介绍。

1．派生类构造函数及析构函数的构造规则

构造函数

构造函数是在构建一个对象的时候，为对象初始化成员的，一个类可以包含多个不同版本的构造函数。我们知道派生类继承了基类的成员，也包括了构造函数，但是每一个派生类都必须有自身的构造函数，用于初始化自身的成员，C++允许声明一个类的时候不指定构造函数，编译器会自动生成缺省的构造函数。

既然派生类继承了基类的成员，就必须在构建派生类的时候，同时对基类的成员进行初始化。在下面两种情况下，必须定义派生类的构造函数：

- 派生类本身需要构造函数；
- 在定义派生类对象时，其相应的基类对象需调用带有参数的构造函数。

构造函数的格式：

派生类构造函数(参数列表)：基类构造函数(参数列表)，对象成员 1 的构造函数(参

数列表)，……，对象成员 n 的构造函数(参数列表)

例如：

```
class Base    //定义一个基类
{
   //...
public：
   Base(int First，int Second);//基类构造函数
   //...
}
//-----------------------------------------------------
class Driv : Base
{
   //...
public：
   Driv(int First, int Second,int Other);
   //派生类构造函数，此处不用缀上基类的构造函数
};
//-----------------------------------------------------
Driv::Driv(int First,int Second,int Other) : Base(First,Second)
{   //定义派生类构造函数时缀上基类的构造函数
    //...
}
```

若基类使用缺省构造函数或不带参数的构造函数，则在派生类中定义构造函数时可略去“ ：基类构造函数(参数类列表)”；此时若派生类不需要特殊的初始化，则可不定义构造函数。构造函数在派生类被构建时执行的顺序是先祖先(基类)，再客人(对象成员)，后自己(派生类本身)。这和 Object Pascal 是不一样的，而 C++ Builder 的 VCL 是使用 Object Pascal 编写的，因而 C++ Builder 中 VCL 类的构造顺序和 Ansi C++ 有所不同，这是读者需要注意的。

析构函数

在派生类中是否要定义析构函数与它所属的基类无关。若派生类在被销毁之前有数据需做善后工作(如释放在构造函数中分配的内存等)，就需定义析构函数。基类的析构函数不会因为派生类没有析构函数而得不到执行，它们各自是独立的，也因此析构函数总是不带参数的，而且一个类的析构函数只有一个。

若基类、成员类、派生类均有析构函数，则执行时的顺序刚好与构造函数的相反，即先自己(派生类)、再客人(对象成员)、后祖先(基类)。

2．示例

下面举几个带有构造函数和析构函数的例子，来说明派生类中构造函数与析构函数的使用。

有四个类。Data 为基类，它含有一个需传递一个参数的构造函数，用它来为其私有成员 x 赋值，并显示一句话；类 a 中含有一个 Data 类的成员对象；类 b 为类 a 的派生类，并在其中也含有一个 Data 类的成员对象；类 c 为类 b 的派生类。构造这些类，并在它们的构造函数中均加上输出本类名的语句，这样可验证这些构造函数的执行顺序。程序如下：

```
#include <iostream.h>
class Data                              //定义 data 类
{
   int x ;
public：
   Data(int x) {data::x = x;}           //基类构造函数
   cout<<"class Data \n";
};
//-----------------------------------------------------
class a                                 //定义 a 类
{
   Data d1;                             //dl 为 data 类对象；作为 a 类的成员对象
public:
   a(int x) : d1(x)                     //定义构造函数，缀上对象成员的构造函数
   {
      cout<<"class a \n";
   }
};
//-----------------------------------------------------
class b : public a                      //定义 b 类，为 a 类的派生类
{
   Data d2;                             //d2 为 data 类对象，作为 b 类的成员对象
public:
   b(int x) : a(x), d2(x)               //定义构造函数,缀上基类和成员对象的构造函数
   {
     cout<<"class b\n";
   }
}
//-----------------------------------------------------
class c : public                        //定义 c 类，为 b 类的派生类
{
public:
   c(int x)：b(x)                       //定义构造函数，缀上基类 b 的构造函数
   {
```

```
        cout<<"class c\n";
    }
};
//-------------------------------------------------
main()
{
    c object(5)；    //定义c类的一个对象object，为构造函数传递参数为5
    return 1;
}
```

运行结果为：

```
class Data
c1ass a
class Data
class b
c1ass c
```

从得到的结果看出，构造函数的调用严格地按照先祖先(基类)，再客人(成员对象)，后自己(派生类本身)的顺序。

4.2 多继承

前面介绍的是，一个派生类是由一个基类派生的，但在解决实际问题时，有时还会遇到更复杂的情况，某个类也许具有多个类所具有的特点，这就是下面要讨论的多继承。

4.2.1 多继承的概念

当一个类 B 从类 A 派生出来，类 B 继承了类 A 的部分或全部属性。即使给定的一个类本身是派生类，也还可以从它派生出所需要的任意多的类，这就构成了类层次。例如，有下列几个类定义：

```
class A  {//…};
class B : A  {//…};
class C : A  {//…};
class D : A  {//…};
class E : B  {//…};
class F : B  {//…};
class G : D  {//…};
class H : F  {//…};
```

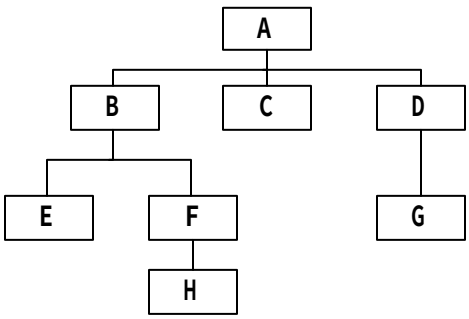


图 2.3 单继承的类层次图

定义的这八个类组成了如图 2.3 的类层次。

图 2.3 中的各个派生类都只有一个基类，因此组成了一个单继承的类层次。

C++中允许派生类具有多个基类，这就是多继承。所谓多继承就是某一个派生类具有多个基类的继承。例如，用户界面所提供的窗口、滚动条、尺寸框以及多种类型的按钮，所有这些假定都是通过类来支持的，若把这些类型中的两个或多个合并产生一个新类，如把窗口和滚动条合并起来产生一个可翻滚的窗口，这个可翻滚的窗口就是由多继承得来的。

读者需要注意的是，在 Object Pascal 中不允许多重继承，因而 C++ Builder 中 VCL 类库也是不可以多继承的，实际上多继承在实际的应用中也不是必须，可以通过适当的变通建立另一种组合关系，比如上面说的“可翻滚的窗口”，如果不使用多继承，就可以用一个容纳了一个滚动条的窗口来描述，这是将两个类组合在了一起，而形成的，属于包含关系，而不是单个一个对象。

4.2.2 多继承的定义

在 C++中，要定义具有两个以上基类的派生类，与前面的派生类定义相同，只需将要继承的多个基类使用逗号分隔即可。例如：

```
class c : a, b
{
   //……
};
```

c 是由 a 和 b 作为基类多重继承得来的派生类。在定义派生类时还可定义它的访问控制。例如，上述定义说明，c 类对于其基类 a，b 均为私有派生。若将上述定义改为如下几种形式，c 类对于其基类将具有不同的派生类型：

```
class c : public a,b        //c 类是对 a 类的公有派生，对 b 类的私有派生
{
   //……
};
class c : a, public b       //c 类是对 a 类的私有派生，对 b 类的公有派生
{
   //……
};
class c : public a, public b   //c 类对其基类 a，b 均为公有派生
{
   //……
};
```

在多重继承中，公有派生和私有派生对于基类成员在派生类中的可访问性与单继承的

规则相同。

4.2.3 多继承的构造函数与析构函数

具有多个基类的派生类其构造函数和析构函数与单继承的情况稍有不同，和单继承一样，派生类的构造函数和析构函数都和它们基类的构造函数和析构函数发生关系，主要是在基类的构造和析构的顺序上。

1．构造函数的定义

多继承的构造函数的定义与单继承的相似，只是几个基类的构造函数之间用“，”分隔。例如我们前面讲过的窗口类和滚动条类：

```
class Window    //定义窗口类
{
   //……
public:
   Window(int Top, int Left, int Bottom, int Right);
   ~Window();
};
//-----------------------------------------------------
class Scrollbar    //定义滚动条类
{
   //……
public:
   Scrol1bar(int Top, int Left, int Bottom, int Right);
   ~Scrollbar();
};
//-----------------------------------------------------
class ScrollbarWind : Window,Scrollbar    //定义派生类
{
   //……
public:
   ScrollbafWind(int Top, int Left, int Bottom, int Right);
   ~ScrollbarWind();
};
//-----------------------------------------------------
ScrollbarWind::ScrollbarWind(int Top, int Left, int Bottom, int Right) :
           Window(Top, Left, Bottom, Right),
           Scrollbar(Top, Right-20, Bottom, Right )    //缀上基类的构造函数
{
  //……
```

```
}
```

通过上面的定义,得到了一个带有滚动条的窗口,这个窗口类是由一般窗口类 Window 和滚动条类 Scrollbar 共同派生得来的。

2．构造函数和析构函数的执行顺序

多继承的构造函数的调用顺序与单继承的相同，也是遵从先祖先(基类)，再客人(成员对象)，后自己(派生类)的原则。在多个基类之间则严格按照派生定义时从左到右顺序来排列先后。而析构函数的调用顺序则刚好与构造函数的相反。下面通过一个例子，来显示多继承中构造函数和析构函数的调用顺序：

```
#include <iostream.h>
#include <conio.h>
//-----------------------------------------------------
class x
{
   //……
public:
    x() {cout<<"the constructor of class x! \n";}
    ~x() {cout<<"the destructor of class x! \n";}
   //……
};
//-----------------------------------------------------
class y
{
   //……
public:
    y() {cout<<"the constructor of class y! \n";}
    ~y() {cout<<"the destructor of class y! \n";}
   //……
};
//-----------------------------------------------------
class z : public x,public y
{
   //……
  public:
    z() {cout<<"the constructor of class z! \n";}
    ~z() {cout<<"the destructor of class z! \n";}
   //……
};
```

```
//-----------------------------------------------------
main()
{
    z obj;
    return 1;
    getch();
}
```

运行结果为：

```
the constructor of class x!      //x 类构造函数的执行结果
the constructor of class y!      //y 类构造函数的执行结果
the constructor of class z!      //z 类构造函数的执行结果
the destructor of class z!       //z 类析构函数的执行结果
the destructor of class y!       //y 类析构函数的执行结果
the destructor oj class x!       //x 类析构函数的执行结果
```

在定义派生类 z 时，基类的顺序为 x 在前，y 在后，之间用“，”分隔，因此构造函数的执行顺序应为先 x，再 y，后派生类 z；析构函数的执行顺序正好相反。执行所得到的结果的顺序与此原则是相符的。

在这个例子中，所有的类不需要对成员进行初始化，因此没有显示的声明派生类构造函数和基类构造函数的关系，下面用一个例子来演示多继承的应用。

例子：在一个圆内显示正文，实际上是完成显示一个带字符串的圆，可以有多种方法来实现这个功能，这里我们使用从圆类和正文类多重继承得到一个新的类 Mcircle，而圆和正文本身又都是从 point 类派生出的类。这个问题的类层次如图 2.4 所示。

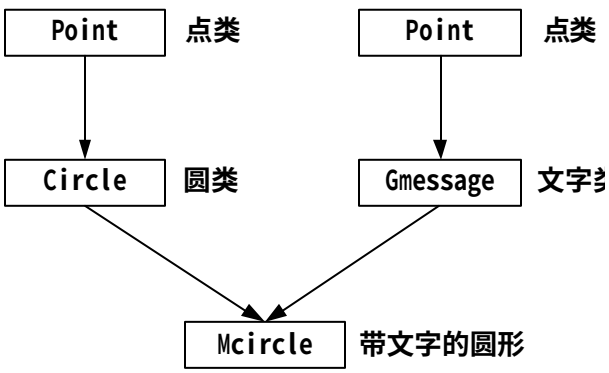


图 2.4 显示文字的圆的继承关系

下面就是这一程序的具体实现：

```
#include <graphics.h>
#include <string.h>
#include <conio.h>
```

```
//-----------------------------------------------------
class point    //定义点类
{
protected:
    int x,y;    //x，y 为 point 类的保护段成员
public:
    point(int initx,int inity);
    //...
};
//-----------------------------------------------------
class circles : point    //定义圆类，它是 point 类的私有派生
{
    int radius;
public:
    circles(int x,int y,int radius);
    void show();
    //...
};
//-----------------------------------------------------
class gmessage : point    //定义正文类
{
    char * msg;
    int font;
    int field;
public:
    gmessage(int mx, int my, int msgfont, int fleldsize, char * text);
    void show();
    //...
};
//-----------------------------------------------------
class mcircle : circles,gmessage    //定义带字符串的圆类
{
public:
    mcircle(int mrx,int mry,int mrradius,int font,char * msg);
    void show();
    //...
};
//-----------------------------------------------------
point::point(int initx, int inity)
```

```
{
    x = initx;
    y = inity;
}
//-----------------------------------------------------
circles::circles(int x,int y,int radius) : point(x,y)
{
    circles::radius = radius;
}
//-----------------------------------------------------
void circles::show()
{
   circle(x,y,radius);
}
//-----------------------------------------------------
gmessage::gmessage(int mx,int my,int msgfont, int fieldsize,char * text)
          : point(mx,my)
{
    font = msgfont;
    field = fieldsize;
    msg = text;
}
//-----------------------------------------------------
void gmessage::show()
{
    int size = field/(8 * _fstrlen(msg));
    settextjustify(CENTER_TEXT,CENTER_TEXT);      //指定文本输出的对齐方式
    settextstyle(font,HORIZ_DIR,size);            //设置文本输出属性
    outtextxy(x,y,msg);                           //输出字符串
}
//-----------------------------------------------------
mcircle::mcircle(int mrx, int mry, int mrradius, int font, char * msg)
          : circles(mrx, mry, mrradius),
            gmessage(mrx, mry, font,2 * mrradius,msg)
{
};
//-----------------------------------------------------
void mcircle::show()
{
```

```
    circles::show();
    gmessage::show();
};
//-------------------------------------------------
main()
{
    int gdriver = DETECT, gmode;
    initgraph(&gdriver,&gmode,"..\\bgi");
    setcolor(12);
    mcircle la(250,250,225,GOTHIC_FONT,"universerr");
    la.show();
    getch();
    closegraph();
    return 1 ;
}
```

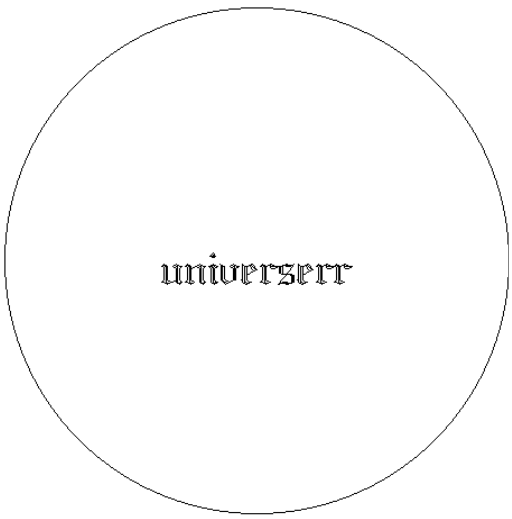


图 2.5 多继承对象的显示结果

这段程序执行后的结果如图 2.5 所示，这是 C++中的一个经典例程，它是在 Dos 的图形模式下运行的，在 Windows 中也可以运行，但是不能在 Windows 2000 和 NT 中运行，本书的光盘中含有所有代码和可执行文件。

4.2.4 虚基类

虚拟基类也是 C++中的一个特点，是和多继承相对应的语言需求，在实际编程中，也会经常用到虚基类。什么是虚拟基类、为什么要使用虚基类、它会给编程带来什么益处，以及编程中如何使用虚基类，我们下面就这个问题简单作一个介绍。

1．为什么要使用虚基类

为什么要使用虚基类，这个问题可以通过下面的例子来解释说明：

```
#include <iostream.h>
//------------------------
class x
{
protected:
   int a;
public:
   x()  {a = 10;}
};
//-------------------------------------------------------
class x1 : public x
{
public:
   x1()
   {
       cout<<a<<"\n";
   }
};
//-------------------------------------------------------
class x2 : public x
{
public:
   x2()
   {
      cout<<a<<"\n";
   }
};
//-------------------------------------------------------
class y : x1, x2
{
public:
   y()
   {
      cout<<a<<"\n";    //错，x1::a 或 x2::a
   }
};
```

```
//-------------------------------------------------
main()
{
    y obj;
    return 1;
}
```

这是一个有问题的程序，问题出在派生类 y 的构造函数的定义上，它试图输出一个它有权访问的变量，表面上看是合理的，但此处的问题不是可访问性的问题，而是二义性的问题，即函数中要输出的 a 变量值，是由 x1 派生路径上来的，还是 x2 派生路径上来的，在这里缺乏明确的说明。

虽然 x1 和 x2 是由共同基类 x 派生而来，但它们所对应的是基类的不同副本，在实际定义对象时，就是对应不同的对象。y 类是 x1 和 x2 的派生类，因此 x 类是 y 类的间接基类，它有两个副本与 y 类相对应，一个是 x1 派生路径上的副本，另一个是 x2 派生路径上的副本。当 y 类要访问这个间接基类 x 时，必须指定要访问的是哪个派生路径上的 x 副本。将上面的例子用一个类层次来表达如图 2.6 所示。但在很多的应用场合，并不希望出现这种情况，而是希望这两个基类的副本合二为一，使 y 类只对应一个间接基类 x，使上面的类层次变为图 2.7 所示。

要实现这种类层次，就需要引出虚基类。

2．虚基类的概念

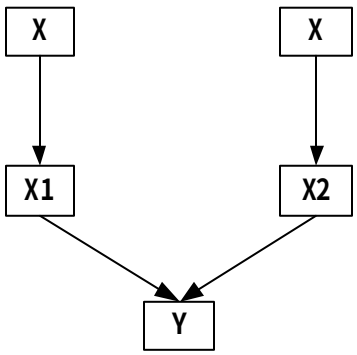


图 2.6 非虚基类的继承结果

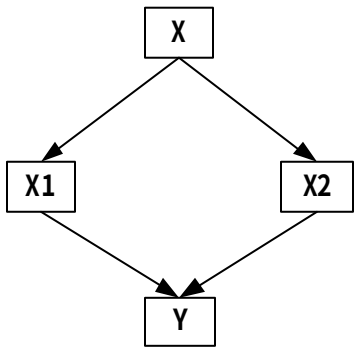


图 2.7 虚基类的继承结果

当在多条继承路径上有一个公共的基类，在这些路径中的某几条路径汇合处，这个公共的基类就会产生多个实例(或多个副本)，若只想保存这个基类的一个实例，可以将这个公共基类说明为虚基类。

定义方法是，在定义派生类时，在需定义为虚基类的基类描述前加上 virtual 关键字。如上例中，可将 x1 和 x2 定义成如下形式：

```
class x1 : virtual public x
{
```

```
    //……
};
//-----------------------------------------------------
class x2 : virtual public x
{
    //……
};
```

这样定义过以后，派生类 y 中的构造函数就不再有问题了。

在现实中，虚基类还有其它的使用价值，例如：当定义图形用户界面时，肯有这样三个类：

```
class port{…}；          图形显示区
class region {…}；       屏幕上的任意区域
class menu{…}；          菜单—选项的集合
```

若从其中派生出两个新类：window 类和 tools 类：

```
class window : public port, public region
{
    //……
};
//-----------------------------------------------------
class tools : public port, public menu
{
    //……
};
```

这两个派生类还可以再派生出新类：

```
class appwind : public window, public tools
{
    //...
}
```

派生 appwind 的目的是为了得到一个带工具条的窗口，但根据上述定义得到的是如图 2.8 所示的结果：

工具条和窗口分别拥有自己的显示区，这并不是所希望的结果。所希望得到的是这两个对象为一个整体，只包含一个显示区，即工具条是包含在窗口之内的。要做到这一点需

要将显示区类 port 定义为虚基类，上述 tools 和 window 的定义变为如下形式：

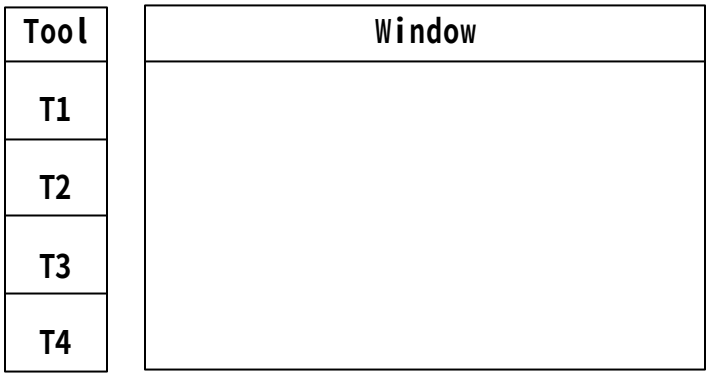


图 2.8 非虚基类的运行结果

```
class window : virtual public port, public region
{
   //……
};
//-----------------------------------------------------
class tools : virtual public port, public menu
{
   //……
};
```

此时窗口即变为如图 2.9 所示的形式：



图 2.9 使用虚基类的运行结果

虚拟基类是一个相对的概念，是否为虚拟基类是在定义派生类的时候决定的，而不是在声明基类的时候决定的，因此同一个基类，在定义它的派生类时，可以作为某些派生类的虚基类的，又可以作为另一些派生类的非虚基类，这种情况是允许的。例如：

```
class b
```

```
{
    //……
};
//-----------------------------------------------------
class x : virtual public b    //b 为 x 派生类的虚基类
{
    //……
};
//-----------------------------------------------------
class y : virtual public b    //b 为 y 派生类的虚基类
{
    //……
};
//-----------------------------------------------------
class z : public b    //b 为 z 派生类的非虚基类
{
    //……
};
//-----------------------------------------------------
class aa : public x, public y, public z
{
    //……
};
```

派生类 aa 的定义，是在三条继承路径的汇合处。aa 与它的间接基类 b 之间对应的关系是，它既是 x、y 继承路径上的一个实例，也是 z 继承路径上的一个实例，可以给出此例的类层次图，如图 2.10。

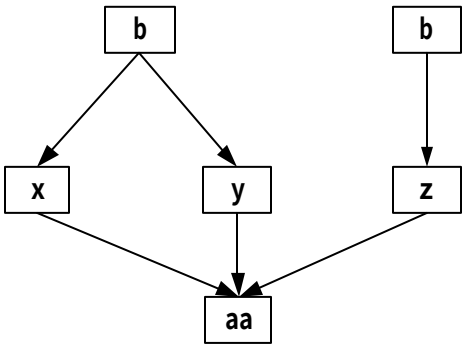


图 2.10 类的层次结构

3．虚基类的初始化

虚基类的初始化与一般的多继承的初始化在语法上是一样的，但构造函数的调用次序不同。

派生类构造函数调用的次序，原则有三：

- 虚基类的构造函数在非虚基类之前调用；
- 若同一层次中包含多个虚基类，这些虚基类的构造函数按它们说明的次序调用；
- 若虚基类由非虚基类派生而来，则仍然先调用基类构造函数，再调用派生类的构造函数。

5 C++中的多态性和重载

多态性是面向对象系统中的又一支柱性概念。在 C++语言中，两个函数的名称完全相同是允许的，而在 C 语言中这是不允许的，即使在不同的程序模块中，只要这两个函数有着某种联系，都是不允许名称相同的。比如我们前面提到的一个类可以有若干个构造函数，而构造函数是和类名相同的，但是在一个类中或者相同的作用域中，这种名称相同的函数，必须有不同的参数，这点表明，C++语言中编译器是根据函数所带的参数来区分不同的函数，而不像 C 语言中必须依靠函数名称来区分不同的函数，这就是 C++语言多态性的一种表现，这是依靠所谓的重载来实现的。

另一方面，在一个类中，允许成员函数具备和其父类“完全相同”的函数，即不仅是函数名称相同，而且参数也完全相同，连返回值也完全相同，这也是一种重载，是派生类对继承父类的成员函数进行重载。而这种重载需要引入“虚函数”的概念。

重载和虚函数是多态性在面向对象系统中两种不同表现形式，既有区别又有联系。重载函数通常在编译时就已经确定该调用那一个正确版本的函数，被称作“早期联编”而虚拟函数是一种特殊的重载函数类型，它在系统运行时才确定调用那一个版本的函数，被称作“滞后联编”。一般来说，“早期联编”提供了执行速度快的优点，而“滞后联编”提供了灵活和高度问题抽象的优点。而这两种编译方式都支持多态性的一般概念。

下面就重载和虚函数分别作以讲解。

5.1 函数重载

重载概念体现了面向对象程序设计的多态性，而且把更大的灵活性和扩展性添加到该语言中。通过为函数和运算符创建附加定义而使它们的名字可以重载，也就是说相同名字的函数或运算符在不同的场合可以表现出不同的行为，实际上是运行的不同函数版本。

两个以上的函数，取同一名字，只要使用不同类型的参数或参数个数不同，编译器便知道在什么情况下该调用哪个函数，这就叫函数重载。为什么要引入函数重载，可以通过一个例子来说明。有四个函数：

```
iplus()        整数相加；
dplus()        实数相加；
```

fplus()　　浮点数相加；
cplus()　　字符串相加。

若用 C 语言来处理，必须给四个函数规定四个不同的函数名。程序员使用这些函数完成两个变量之间的加法时，必须调用合适的函数，也就是说，程序员所必须记住的是四件事，而不是一件事。

在面向对象程序设计中，由于它支持重载，所以用 C++处理时，虽然只给这四个函数起一个共同的名字 plus，但它们的参数类型是不同的。当程序员调用此函数时，只需在其中带入实参，编译器就根据实参类型来确定到底调用哪个重载函数。因此，程序员只需记住 plus 是将两个参量相加的函数这一件事即可，剩下的则都是系统的事情。上述例子我们可以用下面的程序行实现：

```
#include <string.h>
#include <iostream.h>
int plus(int x,int y)                    //定义两个整数相加
{
   return x + y;
}
//-----------------------------------------------------
float plus(float x,float y)              //定义两个浮点数相加
{
   return x + y;
}
//-----------------------------------------------------
double plus(double x,double y)           //定义两个双精度数相加
{
   return x + y;
};
//-----------------------------------------------------
char * plus(char *x,char *y)
{  //定义两个字符串相叠加
   return strcat(x, y);
}
//-----------------------------------------------------
void main()
{
   int i = 12, j = 345;
   float x1 = 1.2, y1 = 4.5;
   double x2 = 24.5, y2 = 635.4,
```

```
    char * strl= "C++ " ,*str2="Builder";
    cout<<plus(i, j)<< "\n";
    cout<<plus(xl, y1)<< "\n";
    cout<<plus(x2, y2)<< "\n ";
    cout<<plus(strl, str2)<< "\n";
}
```

在 main()中的 plus 函数的四次调用，实际上是调用的四个不同的重载版本，由系统根据传递的不同参数类型来决定调用哪个重载版本。例如 plus(i，j)，因为 i，j 为整型变量，所以当执行到此处时系统将调用两个整数相加的重载版本 int plus(int，int)；plus(strl，str2)，因为 strl、str2 为字符串变量，执行到此处时系统将调用两个字符串相叠加的重载版本 char *plus(char *，char *)。从上面可以看出，利用重载概念，程序员在调用函数时，书写非常方便。

5.1.1 构造函数重载

在前面介绍类的构造函数时，曾经讲过在定义构造函数时可定义多个，只是它们的参数的个数或类型取的不同，实际上这就是构造函数的重载，因为那时还没有介绍重载概念，所以没有使用“重载”这个术语。对构造函数进行重载定义，可使系统有几个不同的途径对对象进行初始化。下面举一个计时器的例子。

定义一个计时器类，对计时器对象初始化就是给其数据成员赋初值。在创建对象时，构造函数传递的参数可以用一个整数表示初始的秒数；也可以用数字串表示；还可以传递两个参数，一个表示秒数，一个表示分钟数。这个构造函数既可以先将它们转换成总秒数后再赋值；还可以不带参数，使初始值为 0。有这么多的情况需要构造函数处理，这只有使用构造函数重载才能解决，此程序如下所示：

```
#include <stdlib.h>
class Timer                          //定义计时器类
{
    int Seconds;
public:
    Timer();                         //声明四个重载构造函数
    Timer(char * t);
    Timer(int t);
    Timer(int, int);
    int GetTimer();                  //此函数返回 seconds 的值
    //...
};
//-----------------------------------------------------
Timer::Timer()                       //定义无参数构造函数，给 seconds 清 0
```

```
{
    Seconds = 0;
}
//-----------------------------------------------------
Timer::Timer(int t)              //定义含一个整型参数的构造函数
{
    Seconds = t;
}
//-----------------------------------------------------
Timer::Timer(char * t)           //定义一个含一个数字串的构造函数
{
    Seconds = atoi(t);
}
//-----------------------------------------------------
Timer::Timer(int min, int sec)   //定义一个含两个整型参数的构造函数
{
    Seconds = min*60 + sec;
}
//-----------------------------------------------------
int Timer::GetTimer()            //取私有变量 seconds 的值
{
    return Seconds;
}
//-----------------------------------------------------
main()
{
    Timer tl,t2("123"),t3(23),t4(2,34);
    cout<<"object tl: "<<t1.GetTimer() <<"\n";
    cout<<"object t2: "<<t2.GetTimer() <<"\n";
    cout<<"object t3: "<<t3.GetTimer() <<"\n";
    cout<<"object t4: "<<t4.GetTimer() <<"\n";
    return 1;
}
```

在 main()函数中定义了四个对象。t1 对象没有传递参数，所以在创建它时调用的是无参数的构造函数 Timer::Timer();t2 对象在被定义时传递了一个数字串参数，所以在创建 t2 对象时调用的是构造函数 Timer::Timer(char *)；t3 对象在被定义时传递了一个整型参数，在创建 t3 对象时调用的是构造函数 Timer::Timer(int)；t4 对象在被定义时传递了两个整型参数，在创建 t4 对象时调用的是构造函数

Timer::Timer(int，int)。

5.1.2 类成员函数重载

在类中除了构造函数可以重载外，一般的成员函数也可以重载，其重载原则与构造函数相同，重载函数之间靠所包含的参数的类型与数量不同来进行区分。

下面给出一个日期类的例子，其中包含 year，month，day 为它的数据成员。在构造函数中既包含直接传递三个整数给这三个成员赋值的版本，又包含传递一个日期字符串的版本。在取得日期的函数中也采用了重载，它既可以返回三个量，又可以返回一个字符串。实现程序如下：

```
#include <stdlib.h>
#include <iostream.h>
#include <string.h>
//-----------------------------------------------------
class Date    //定义日期类
{
   int Year;
   int Month;
   int Day;
public:
   Date() {Year = 0; Month = 0; Day = o;};
   Date(int，int，int);
   Date(char *);
   void GetDate(char *)；
   void GetDate(int *,int *,int *);
};
//-----------------------------------------------------
Date::Date(int yy，int mm，int dd)  //传递整型参数的构造函数
{
   Year = yy;
   Month = mm;
   Day = dd;
}
//-----------------------------------------------------
Date::Date(char * str)         //传递字符串的构造函数
{
   char * ss = new char[5];    //定义一个字符串，并为其分配存储
   ss[0] = str[0];           //取字符串前两个字符，为月份
   ss[1] = str[1];
```

```
    ss[2] = '\n';
    Month = atoi(ss);          //将字符串转换成整数
    ss[0] = str[3];            //取字符串中的第 4，5 个字符，为日期
    ss[1] = str[4];
    Day = atoi(ss);
    ss[0] = 'l';
    ss[1] = '9';
    ss[2] = str[6];            //取字符串中的第 7，8 个字符，为年份
    ss[3] = str[7];
    ss[4] = '\n';
    Year = atoi(ss);
    delete ss;                 //释放分配的存储
}
//-----------------------------------------------------
void Date::GetDate(char * str)
{
    char * ss = new char[5];
    itoa(Month, ss, 10);       //将 month 转换成字符串给 ss，10 为进制
    if(Month<10)
    {
       ss[1] = ss[0];          //若月份小于 10，在第一位加上 0
       ss[0] = '0';
       ss[2] = '\0';
    }
    strcpy(str, ss);           //字符串赋值
    itoa(Day, ss, 10);         //将 day 转换成字符串给 ss
    strcat(str, '/');          //在日期字符串中的月份后加入分隔符
    if(day < 10)
    {
       ss[1] = ss[0];          //日小于 10，第一位为 0
       ss[0] = '0';
       ss[2] = '\n';
    }
    strcat(str, ss);           //将 ss 加到 str 后面
    itoa(Year,ss,lo);          //将 year 转换成字符串给 ss
    ss[0] = '/';
    ss[1] = ss[2];
    ss[2] = ss[3];
    ss[3] = '\0';
```

```
    strcat(str, ss);          //将表示年份的字符串加到日期表达式后面
    delete ss;                //将分配的存储释放
}
//-------------------------------------------------
void Date::GetDate(int  *yy, int  *mm, int  *dd)
{   //将年、月、日值分别赋给三个整数
    *yy = Year;
    *mm = Month ;
    *dd = Day;
}
//-------------------------------------------------
main()
{
    int *yy = new int, *mm = new int, *dd = new int;
    char * date = new char[9];
    Date da1,da2(2002,12,27),da3("03/01/96");
    da2.GetDate(datt);
    da3.GetDate(yy, mm, dd);
    cout<<date<<"\n";
    cout<<*yy<<" "<<*mm<<" "<<*dd<<"\n";
    delete yy,mm,dd,date;
    return 1;
}
```

在 main()函数中，首先定义了三个对象，da1 对象被创建时调用无参量构造函数，da2 对象在被创建时调用了整型参量的构造函数 Date(int，int，int)，da3 对象被创建时调用了字符串参量的构造函数 Date(char *)。da2.GetDate(date)函数的调用，因为 Date 为字符串变量，所以此时调用的是 GetDate(char *)重载版本；da3.GetDate(yy，mm，dd)函数的调用，因为 yy，mm，dd 均为整型指针变量，所以此时调用的是 GetDate(int *，int *，int *)重载版本。

5.1.3 类以外的一般函数重载

在 C++中，除了类中的各成员函数允许重载以外，类以外的一般函数也允许重载，规则都一样。例如，写个求最大值函数 max 的重载版本，用它来比较两个数(可以是整数、实数、字符串)的大小，并返回其中较大的一个。程序行如下：

```
#include <iostream.h>
#include <string.h>
int max(int x,int y)              //比较两个整数大小
```

```
{
    return x > y ? x : y;
}
//-----------------------------------------------------
double max(double x,double y)    //比较两个实数大小
{
    return x > y? x : y;
}
//-----------------------------------------------------
char * max(char *x,char*y)       //比较两个字符串大小
{
    return strcmp(x,y) > 0? x: y;
}
//-----------------------------------------------------
main()
{
    cout<<max(9,4)<<"\n";
    cout<<max(4.5,7.4)<<"\n";
    cout<<max("windows","MS-Dos")<<"\n";
    return 1;
}
```

在 main()函数中，max(9,4)中的 9 和 4 均为整数，因此它调用的是 max 函数的整数版本 int max(int,int);max(4.5,7.4)中的 4.5 和 7.4 为实数，因此它调用的是 max 函数的实数版本 double max(double,double);最后一个函数调用 max("windows","MS-Dos")，其中的参数为字符串，因此它调用的是 max 函数的字符串版本 char *max(char *,char *)。

5.2 运算符重载

在 C++中，除了函数可以重载之外，运算符也是可以重载，运算符的重载不仅增强了 C++语言的可扩充性也提高了使用的灵活性。最著名的例子是 C++中提供的复数运算，复数是有实部和虚部两个部分组成的，通常在数学运算库中，复数的加、减、乘、除等运算都不能使用 C 语言中的“+、-、*、/”四个运算符，而在 C++中，通过对运算符的重载，可以使复数运算向整数、浮点数一样使用“+、-、*、/”来运算。

在 C++中，说明一个类就是说明一个新类型。因此，类对象和变量一样，可以作为参数传递，也可以作为返回类型。编译器为基本数据类型定义了许多运算符，例如"+"运算符：

```
int x, y;
y = x + y;
```

这是表达两个整数相加的方法，非常简单，但是对于一个类这样的表达式是无效的，例如对类“x”要实现特定的“加法”运算，需要使用类似下面的代码：

```
class x
{
public:
   int y;
   //...
};
//-----------------------------------------------------
main()
{
   x a,b;               //定义了两个对象
   a.y = a.y + b.y;     //将两个对象的数据内容相加
   //...
   return 1;
}
```

这样表达不如前面的简洁，也不直观；而且还存在一个问题，若类 x 中的 y 为私有成员，上述表达式 a.y = a.y + b.y 就是错误的。

因此，人们为了表达上的方便，希望已预定义的运算符(如+，-，*，/等)，也可以在特定类的对象上以新的含义进行解释。如上面的表达式则希望变为 a = a + b，这就需要用重载运算符来解决。

在 C++中，大多数系统预定义的运算符都能被重载。例如：

```
+       -       *(乘法) /       %       &
~       !       =       <       >       +=      -=      *=
/=      %=      ^=      &=      |=      <<      >>      >>=
<<=     ==      !=      <=  >=  &&      ||      ++
--      []      ()      new     delete
```

也有一些运算符是不能被重载的，如：

```
.       ::      *(指针) ?:      #
```

重载运算符时，不能改变它们的优先级，不能改变这些运算符所需操作数的数目。实际上被重载的运算符也是通过调用特定的函数来完成其运算的，但是使用了重载的运算符不仅在表达方式上简单、直观了很多，也减少了出错的机会。

5.2.1 用成员函数重载运算符

在 C++中，用成员函数重载运算符就是将运算符重载定义成一个类的成员函数的形式，通常将重载运算符的成员函数称为运算符函数。

(1) 在类定义体中声明运算符函数

声明重载运算符时，写成如下形式：

type operator@(**参数表**)

其中，type **为返回类型；**operator **是运算符重载时不可缺少的关键字；@为所要重载的运算符符号；参数表中罗列的是该运算符所需要的操作数。**

用成员函数重载运算符的函数参数表中，若运算符是一元的，则参数表为空，此时当前对象作为运算符的单操作数；若运算符是二元的，则参数表中有一个操作数，此时当前对象作为此运算符的左操作数，参数表中的操作数作为此运算符的右操作数。

现在举一个例子说明：

```
class x
{
   //...
   int operator + (x a);
};
```

在类定义中声明一个重载运算符"+"，返回类型为 int，它具有两个操作数，一个是当前对象，另一个是对象 a**。**

(2) 定义运算符函数

运算符函数的定义与一般的成员函数相似，对类成员的访问与一般成员函数相同，格式如下：

type x::operator@(**参数表**)

*//**定义的操作***

其中 x **是可以重载此运算符的类名，其它符号的含义与上面相同。**

上面在 x **类中重载"+"的例子也可定义为下面的形式：**

```
int x::operator + (x a)
{
   return a.x + x;
}
```

(3) 重载运算符的使用

运算符重裁定义后是为了方便使用，定义后的重载运算符使用起来就像原运算符一样方便。只是要注意一点，即它的操作数一定要是定义它的特定类的对象，如上面重载“ +”运算符的例子可用一个 main()**来使用它：**

```
main()
{
   x a1,b1;
   y a2,b2;
   a1 = a1 + b1;    //正确
```

```
    a2 = a2 + b2;    //错误
    //...
}
```

从例子中看出，al = a1 + b1 是正确的，因为 a1，b1 是 x 类的对象，在 x 类中已重载了“+”运算符，因此可直接使用。而 a2 = a2 + b2 是错误的，这是因为 a2，b2 是 y 类的对象，而不是 x 类的对象，在 y 中没有定义“+”重载，因此不能使用。在 x 类中对“+”运算符有了重载定义，因此对此类对象进行相加时就像做一般的加法一样简单。

5.2.2 用友元重载运算符

在 C++中，还可以把重载运算符定义成某个类的友元函数的形式，而且这种方法用得更多。在定义格式上与用成员函数定义相似，但稍有差别。

用友元重载运算符的函数也称运算符函数。它与用成员函数重载运算符的函数之不同，在于后者本身是类中的成员函数；而它是类的友元函数，是独立于类外的一般函数。

(1) 在类定义体中声明重载运算符

在类定义体中，要声明用友元重载的运算符时与一般的友元函数有些相似，可采用如下形式：

friend type operator@(参数表)；

与用成员函数定义的方法相比较，只是在前面多了一个关键字**friend**，其它项目含义相同。

在为用友元函数重载运算符的参数表填写操作数时，要注意友元函数是不属于任何类的，它没有 this 指针，这与成员函数完全不同。若重载的是一元运算符，则在参数表中有一个操作数用来充当这唯一的操作数；若重载的是二元运算符，则在参数表中有两个操作数。也就是说在用友元定义重载运算符时，所有的操作数均需要用参数来传递。

看下面一个声明的例子：

```
class point
{
    int x, y;
public:
    //...
}
friend point operator + (point p1,point p2);
```

因为是用友元重载的运算符"+"，而"+"又是一个二元运算符，所以需要传递两个参数，这里传递的是 p1 和 p2。

(2) 定义重载运算符

定义用友元重载的运算符与定义一般的友元函数相似，格式如下：

```
type operator @(参数表)
```

例如上面 point 类中用友元重载的"+"运算符的定义为：

```
point operator + (point p1, point p2)
{
    point p;
    p.x = p1.x + p2.x;
    p.y = p1.y + p2.y;
    return p;
}
```

(3) 重载运算符的使用

用成员函数定义的重载运算符和用友元定义的重载运算符，使用起来没有什么差别，因此在这里不再重复。

下面代码将示范分别使用成员函数和友员函数来为类 TOPClass 重载“+”和“*”运算符，TOPClass 包含了两个成员：x、y，“+”运算将两个操作数的 x、y 分别相加作为结果的 x 和 y，“*”运算将两个操作数的 x 相乘作为结果的 x，y 相乘作为结果的 y，代码如下：

```
class TOPClass;
TOPClass operator* (TOPClass Cls1,TOPClass Cls2);

class TOPClass
{
public:
    int x;
    int y;
    TOPClass operator + (TOPClass TheClass);
    friend TOPClass operator*(TOPClass Cls1,TOPClass Cls2);
};

TOPClass operator *(TOPClass Cls1,TOPClass Cls2)
{
    TOPClass Result;
    Result.x = Cls1.x * Cls2.x;
    Result.y = Cls1.y * Cls2.y;
    return Result;
```

```
}

TOPClass TOPClass:: operator + (TOPClass TheClass)
{
    TOPClass Result;
    Result.x = x + TheClass.x;
    Result.y = y + TheClass.y;
    return Result;
}
```

5.3 虚函数

虚函数是重载的另一种表现形式，它是一种动态的重载方式，它提供了一种更为灵活的多态性机制。虚函数允许函数调用与函数体之间的联系在运行时才建立，也就是在运行时才决定如何动作，即所谓的“动态连接”。当一般的类型对应于不同的类型变种时，这个能力显得尤其重要。下面先介绍对象指针，然后介绍虚函数的引入及其使用。

5.3.1 对象指针

在 C++中，对象除可以直接引用外，还可以通过对象指针来引用。在前面虽然没有正式介绍对象指针，但在许多例子中已经采用了对象指针。

1. 一般对象的指针

指向一般对象的指针与指向一般变量的指针，其定义和使用语法都是相同的。例如：

```
#include <iostream.h>
c1ass class1
{
   //...
public：
    void show()
    {cout<<"this is the class of class1!\n";}
};
class class2
{
   //...
public：
    void show()
    { cout<<"this is the class of class2!\n";}
};
main()
```

```
{
    class1 cl, *ptrl;          //定义classl类对象c1和classl类指针ptrl
    class2 c2, *Ptr2;          //定义class2类对象c2和class2类指针ptr2
    ptrl = &c1;                //将ptrl指针指向cl对象
    ptr2 = &c2;                //将ptr2指针指向c2对象
    ptr1->show();
    ptr2->show();
    return 1;
}
```

运行结果为：

```
this is the class of classl!
this is the class of class2!
```

2． 引入派生类后的对象指针

前面介绍的一般对象的指针，它们各自独立，之间没有什么联系，相互不能混用。引入派生类后，由于派生类是由基类派生出来的，派生类和基类之间是息息相关的，因此指向派生类和基类的指针也是相关的。

在引入了派生的概念后，任何被说明为指向基类对象的指针都可以指向它的公有派生类。下面看一个例子：

```
#include <iostream.h>
#include <string.h>
class String
{
    char *Name;
    int Length；
public：
    String(char *Str)
    {
        Length  = strlen(Str)；
        Name = new char[Length + 1]；
        strcpy(Name,Str)；
    }
    void show()
         {cout<<Name<<"\n"；}
};
class de_string : public String
{
    int age;
```

```
public：
    de_string(char *str，int age) : String(str)
    {
        de_string::age = age；
    }
    void show()
    {
        string::show();
        cout<<"the age is："<<age<<"\n"；
    }
};
main()
{
   string s1(“Smith”),*ptrl;     //定义string类对象s1及string类指针ptrl
                          //de_string s2(“Jean”，20), *ptr2;
                          //定义de_string类对象s2及de_string类指针ptr2
    ptrl = &s1；          //将ptrl指向s1对象
    ptrl->show();         //调用string类的成员函数
    ptrl->&s2;            //将ptrl指向string类的派生类de_string的对象s2
    ptrl->show()；        //调用s2对象所属的基类的成员函数show()
    ptr2 = &s2；          //将指针ptr2指向de_string类对象s2
    ptr2->show()；        //调用de_string类的成员函数show()
    return 1;
}
```

运行结果为：

```
Smith
Jean
Jcan
the age is：20
```

从例子中看出，虽然 ptrl 指针已经指向了 s2 对象(ptr1 = &s2)，但是它所调用的函数(ptrl->show())仍然是其基类对象的成员函数，这是使用时要注意的问题。在使用引入派生类之后的对象指针时要注意下面几个问题：

- 可以用一个声明让指向基类对象的指针指向它的公有派生的对象。若试图指向它的私有派生的对象则是被禁止的。
- 不能将一个声明为指向派生类对象的指针指向其基类的一个对象。
- 声明为指向基类对象的指针，当其指向派生类对象时，只能利用它来直接访问派生类中从基类继承来的成员，不能直接访问公有派生类中特定的成员。

这三点是使用对象指针所需要注意的，如果希望通过指向派生类的基类指针可以派生

类的特定成员，可以将其显式的类型转换为派生类指针来实现。例如：

```
class Point
{
protected:
    int x, y;
public:
    Point(int x, int y)
      {Point::x = x;point::y = y;}
    void Show()
      {putpixel(x, y, getcolor());}
    void Show1() {  cout<<"x = "<<x<<" y = "<<y<<"\n";}
};
class Circles : public Point
{
    int Radius;
public:
    Circles(int x, int y, int radius) : Point(x, y)
       {circles::Radius = radius;}
    void Show()
       {Circle(x,y,radius);}
    void Show2()
    {
       Show1()
       cout<<"Radius = "<<Radius<<"\n";
    }
};
main()
{
    Point obj1(100,100),*ptr;          //定义对象 objl 及指向 point 类的指针 ptr
    Circles obj2(200,200,100);         //定义 obj2 对象
    ptr = &obj1;                       //将指针 ptr 指向 objl 对象
    int gdrivef = DETECT,gmode;
    initgraph(&gdriver,&gmode,' ');
    setcolor(13);
    ptr->Show();          //调用 objl 对象所属类 point 中的成员函数在屏幕上画一个点
    ptr = &obj2;          //将指针 ptr 指向其派生类对象 obj2
    ptr->Show();          //调用 obj2 对象从基类继承来的成员函数，在屏幕上画一个点
    ((Circles *)ptr)->Show();          //调用 obj2 对象的特定成员 show()，画一个圆
```

```
    getch();
    closegraph();
    ptr = &obj1;
    ptr->Show1();
    ptr = &obj2;
    ((Circles *)Ptr)->Show2();       //调用 obj2 对象的成员函数 show2()
    getch();
    return 1;
}
```

灵活的指针应用是C语言的一大特色,因而在C++中对象的指针也是非常重要的概念，而且具有非常广泛的用途，尤其是在 C++ Builer 中，所有的 VCL 类库必须是动态分配的内存，因此对 VCL 对象的使用必须使用对象指针。

5.3.2 虚函数的必要性

为什么要引入虚函数，我们先来看一个例子：

```
#include <iostream.h>
class base
{
public:
    void who()
      {cout<<"this isthe class of base! \n";}
};
class derive1 : public base
{
public:
    void who()
      {cout<<"\this is the class of derivel! \n";}
};
class derive2 : public base
{
public:
    void who()
       {cout<<"this is the class of derive2! \n";}
};
main()
{
    base obj1,*p;
    derive1 obj2;
```

```
    derive2 obj3;
    p = &obj1;        //指针 p 指向 base 类对象 objl
    p->who();
    p = *obj2;        //指针 p 指向 derivel 类对象 obj2
    p->who();
    p = &obj3;        //指针 p 指向 derive2 类对象 obj3
    p->who();
    obj2.who();
    obj3.who();
    return 1;
}
```

此例在 main()函数中定义了二个基类对象 obj1，和两个派生类对象 obj2，obj3，又定义了一个指向基类对象的指针 p。此程序的意图是用 p 指针分别指向不同的对象，以便执行不同对象所对应的类的成员函数。当 p 指向 objl 对象时，则 p->who()调用 base 类的成员函数 who()；当 p 指向 obj2 对象时，我们希望 p->who()调用 derivel 类的成员函数 who()；当 p 指向 obj3 对象时，希望 p->who()调用 derive2 类的成员函数 who()。

此程序执行后实际得到的结果是：

```
this is the class of base!          ①
this is the class of base!          ②
this is the class of base!          ③
this is the class of derivel!       ④
this is the clasl of derive2!       ⑤
```

在结果中，①，④，⑤，与所预想的相符，而②和③却不是希望得到的。这说明，不管指针 p 当前指向哪个对象(是基类对象还是派生类对象)，p->who()调用的都是基类中定义的 who()函数版本。也就是说，通过指针引起的普通成员函数调用，仅仅与指针的类型有关，而与此刻正指向什么对象无关。在这种情况下，必须显式的调用派生类中的函数成员，例如：

obj2.who()或 obj3.who()或者是采用对指针的强制类型转换的方法，如：

```
((derie1 *)p)->who()，或((derive2 *)p)->who()
```

本来使用对象指针是为了表达一种动态的性质，即当指针指向不同对象时执行不同的操作，现在看来并没有起到这种作用。要实现这种功能，就需要引入虚函数的概念，上面的功能只需将基类的 who()函数声明为虚函数即可。

5.3.3 虚函数的定义及使用

1．虚函数的定义

虚函数是引入了派生概念以后，用来表现基类和派生类的成员函数之间的一种关系的。虚函数定义是在基类中进行的，它是在基类中需要定义为虚函数的成员函数的声明中冠以关键字 virtual，从而提供了一种接口界面。在基类中的某个成员函数被声明为虚函数后，此虚函数就可以在一个或多个派生类中被重新定义。在派生类中重新定义时，其函数原型，包括返回类型、函数名、参数个数、参数类型顺序，都必须与基类中的原型完全相同。如下面的例子：

```
#include <iostream.h>
class base
{
   //...
public：
   virtual void who()    //定义虚函数
      {cout<<"base\n";}
};
class first : public base
{
   //...
public:
   void who()    //重新定义虚函数
       {cout<<"the first derivation \n";}
};
class second : public base
{
   //...
public:
   void who()    //重新定义虚函数
       { cout<<" the second derivation\n";}
};
main()
{
   base obj1,*ptr;
   first obj2;
   second obj3;
   ptr = &obj1;
   ptr->who();    //调用 base 类的 who()版本
```

```
    ptr = &obj2;
    ptr->who();      //调用 first 类的 who()版本
    ptr = &obj3;
    ptr->who();      //调用 second 类的 who()版本
    return 1 ;
}
```

运行结果为：

```
base
the first derivation
the second derivation
```

分析一下上面这个程序：

①在基类中对 void who()进行了虚函数声明 virtual，在其派生类中就可以重新定义它。

②在派生类 first 和 second 中分别重新定义 void who()函数，此虚函数在派生类中重新定义时不再需要 virtual 声明，此声明只在其基类出现一次。在 void who()函数被重新定义时，其函数的原型与基类中的函数原型必须完全相同。

③在 main()函数中，定义了一个指向基类类型的指针，它也被允许指向其派生类。在执行过程中，不断改变它所指向的对象，p->who()就能调用不同的版本。虽然都是 p->who()语句，但是当 p 指向不同的对象，所对应的执行动作就不同。可见用虚函数充分体现了多态性，并且，因为 p 指针指向哪个对象是在执行过程中确定的，所以体现的又是一种动态的多态性。

2 虚函数与重载函数的关系

在一个派生类中重新定义基类的虚函数是函数重载的另一种特殊形式，但它不同于一般的函数重载。一般的函数重载，函数的返回类型及所带的参量可以不同，仅是函数名相同即可，而虚拟函数要求函数名称和参数以及返回值必须严格的一致，否则将会出错或者失去虚特性。

- 假如仅仅返回类型不同，其余均相同，系统会当作出错处理，因为仅仅返回类型不同的函数本质上是含糊的，不是虚拟函数也不是重载函数。
- 函数原型不同，仅函数名相同。此时，系统会将它认为是一般的函数重载，将丢失虚特性。

下面看一个例子：

```
#include <iostream.h>
class base
{
public：
    virtual void f1()
        {cout<<"function of base !  \n";}
    virtual void f2()
```

```
        {cout<<"f2 function of base ! \n";}
    virtual void f3()
        {cout<<" f3 function of base! \n";}
    void f4()
        {cout<<" f4 function of base! \n";}
};
class derive : public base
{
    void f1()
        {cout<<" f1 function of derive! \n";}
    void f2(int x)
        {cout<<"f2 function of derive!  \n";}
    int f3();       //错误，只有返回类型不同，应去掉
    void f4()
        {cout<< "f4 function of derive! \n";}
};
main()
{
    base obj1,* ptr;
    derive obj2;
    ptr = &obj1;
    ptr->f1();
    ptr->f2();
    ptr->f3();
    ptr = &obj2;
    ptr->f1();
    ptr->f2();
    ptr->f4();
    return 1;
}
```

此例在基类中定义了三个虚函数 f1()，f2()，f3()，这三个函数在派生类中被重新定义时，f1()符合规则，它仍为虚函数；f2()增加了一个整型参量，变为 f2(int x)，因此它丢失了虚特性，变为一般的重载函数；f3()的返回类型由 void 变为 int，系统指出冲突的出错信息。基类中的 f4()函数和派生类中的 f4()函数都为一般的重载函数。

在 main()函数中，定义了一个基类指针 ptr，在执行过程中当指向基类对象时，用它调用的函数均为基类的成员函数；当 ptr 指向派生类对象 obj2 时，ptr->f1()执行的是派生类中的成员函数，这是因为 f1()为虚函数；ptr->f2()执行的是基类的成员函数，因为 f2()函数丢失了虚特性，按照一般的重载函数来处理；ptr->f4()执行的是基

类的成员函数，因为 f4()为重载函数。

3．多继承中的虚函数

在多继承中由于派生类是由多个基类派生而来的，因而，虚函数的使用就不像单继承那样简单。看下面的例子：

```
#include <iostream.h>
class a
{
public:
   virtual void f()    //定义 f()为虚函数
      {cout<<"class a \n";}
};
class b
{
public:
   void f()            //此 f()函数为一般函数
       {cout<<"class b \n";}
};
class aa : public a,public b
{
public:
   void f()
      {cout<<"class aa \n";}
};
main()
{
   a obj1,*ptr1;
   b obj3,*ptr2;
   aa obj3;
   ptr1 = &obj1;            //将指针 ptrl 指向 a 类对象
   ptr1->f();               //调用 a 类的 f 门
   ptr2 = &ohj2;            //将指针 ptr2 指向 b 类对象
   ptr2->f();               //调用 b 类的 f()
   ptr1 = &obj3;            //将指针 ptrl 指向 a 类的派生类 aa 类的对象
   ptr1->f();               //调用 aa 类的 f()，此时的 f()为虚函数
   ptr2 = &obj3;            //将指针 ptr2 指向 b 类的派生类 aa 类对象
   ptr2->f();   //调用 b 类中的 f()，因为此处 f()为非虚函数，而 ptr2 又为 b 类指针
   return 1;
}
```

从上面的例子看出，派生类 aa 中的函数 f()在不同的场合，呈现不同的性质。若相对于 a 派生路径，由于在 a 中的 f()函数前有关键字 virtual，所以它是一个虚函数；若相对于 b 派生路径，在 b 中的 f()函数为一般函数，所以此时它只能是一个重载函数。

当 a 类指针指向 aa 类对象 obj3 时，函数 f()就呈现出虚特性；当 b 类指针指向 aa 类对象 obj3 时，函数只呈现一般的重载特性。

运行结果如下：

```
c1ass a
class b
class aa
class b
```

若一个派生类，它的多个基类中有公共的基类，在公共基类中定义一个虚函数，则多级派生以后仍可以重新定义虚函数，也就是说虚特性是可以传递的。看下面的例子：

```
#include <iostream.h>
c1ass a
{
public:
    virtual void f()
       {cout<<"class a \n";}
};
class a1 : public a
{
public:
    void f()
        {cout<<"class al\n";}
};
class a2 : public a
{
public:
    void f()
       {cout<<"class a2 \n";)
};
class aa : public al, public a2
{
public:
    void f()
       {cout<<"class aa \n";}
};
```

```
main()
{
    a1 *ptr1;            //定义 al 类指针 ptrl
    a2 *ptr2;            //定义 a2 类指针 ptr2
    aa obj;              //定义 aa 类对象 obj
    ptr1 = &obj;         //a1 类指针 ptrl 指向 aa 类对象 obj
    ptr1->f();           //调用 aa 类的 f()
    ptr2 = &obj;         //a2 类指针 ptr2 指向 aa 类对象 obj
    ptr2->f();           //调用 aa 类的 f()
    return 1;
}
```

运行结果为：

```
class aa
c1ass aa
```

从例子中看出，虚特性是可以传递的。a 类作为 al 和 a2 类的直接基类，它的成员函数 f()被声明为虚函数，则 al 和 a2 类中的 f()都具有虚特性，即均为虚函数；而 aa 类为 a1 和 a2 类的派生类，因此它的成员函数 5()也为虚函数。若 a 类中 f()函数声明时的 virtual 省掉，则运行结果将变为：

```
class a1
class a2
```

4．基类构造函数调用虚函数

当一个基类的构造函数调用一个虚函数时，会出现意想不到的结果。看下面的例子：

```
#include <iostream.h>
class base
{
public：
    base()
     {
        cout<<"the constfuctor of base\n";
        clone()；}
    virtllal void clone()
        {cout<<"the clone function of base\n";}
};
class derive : public base
{
public:
    derive()
```

```
        {cout<<"the constructor of derive\n";}
    void clone()
        {cout<<"the clone function ofderive\n";}
};
main()
{
    derive obj;
    base *ptr = &obj;
    ptr->clone();
    return 1;
}
```

运行结果为：

```
the constructor of base
the clone function of base
the constructof of derive
the clone function of derive
```

输出结果的第二行是出乎意料的，下面来分析一下此程序的运行过程。

首先创建一个派生类对象 obj，在创建时先调用基类构造函数。基类构造函数首先输出第一行结果，然后调用 clone()函数，因为此函数为虚函数，并且当前定义对象为 derive 类对象。照理此时应执行 clone()在 derive 类中的版本，但此时，派生类对象 obj 正在创建中，只做了部分的初始化，当编译器遇到要调用虚函数 clone()时，它无法与派生类中的 clone()版本联编(因为对象还不完整)，编译器只能调用基类的 clone()版本来代替它，因此出现第二行的意外结果。之后再执行派生类的构造函数，输出第三行结果。执行到 ptr->clone()时，派生类对象早已圆满创建，因此就可调用 clone()函数的派生类版本，得到第四行的输出结果。

在构造函数中调用虚拟函数时，一定要注意对象的构造顺序，否则可能会得到意想不到的结果，在 C++ Builder 中 VCL 的构造顺序和 Ansi C++有所不同，调用虚拟函数的结果可能也会有所不同，因此在 C++ Builder 中，除非是必须的，一般不要在构造函数中调用虚拟函数。

6 面向对象的设计过程

前面将 C++语言中的一些特点作了一个简单的介绍，这和面向对象的大部分思想是一致的，通过这些内容的学习，可以对面向对象的程序设计有一个大致的了解。接下来就几种比较经典的程序设计方法作一个简单的介绍。

6.1 几种软件设计方法简介

60 年代中期开始爆发了众所周知的软件危机。为了克服这一危机，在 1968、1969 年连续召开的两次著名的 NATO 会议上提出了软件工程这一术语，并在以后不断发展、完善。与此同时，软件研究人员也在不断探索新的软件开发方法。至今已形成多种软件开发方法。

在这些软件设计方法中，有的是近些年才出现的，而有些是已经被人们运用了数十年的方法。这些设计方法本身并不是相互对立，也不是简单的补充关系，而是每种设计方法都包含了各自的核心思想，在一个软件的开发过程，可能是多重设计方法共同存在，相互补充的。以下对这些设计方法作一简单介绍。

6.1.1 Parnas 方法

最早的软件开发方法是由 D．Parnas 在 1972 年提出的。由于当时软件在可维护性和可靠性方面存在着严重问题，因此 Parnas 提出的方法是针对这两个问题的。首先，Parnas 提出了信息隐蔽原则：在概要设计时列出将来可能发生变化的因素，并在模块划分时将这些因素放到个别模块的内部。这样，在将来由于这些因素变化而需修改软件时，只需修改这些个别的模块，其它模块不受影响。信息隐蔽技术不仅提高了软件的可维护性，而且也避免了错误的蔓延，改善了软件的可靠性。现在信息隐蔽原则已成为软件工程学中的一条重要原则。

Parnas 提出的第二条原则是在软件设计时应对可能发生的种种意外故障采取措施。软件是很脆弱的，很可能因为一个微小的错误而引发严重的事故，所以必须加强防范。如在分配使用设备前，应该取设备状态字，检查设备是否正常。此外，模块之间也要加强检查，防止错误蔓延。

Parnas 对软件开发提出了深刻的见解。遗憾的是，他没有给出明确的工作流程。所以这一方法不能独立使用，只能作为其它方法的补充。但这中方法实际上始终存在于我们的软件开发过程中。不光是驱动程序、操作系统，就是在一般的应用程序中，开发也通常遵循这一规则。

6.1.2 SASD 方法

1978 年，E．Yourdon 和 L．L．Constantine 提出了结构化方法，即 SASD 方法，也可称为面向功能的软件开发方法或面向数据流的软件开发方法。1979 年 TomDeMarco 对此方法作了进一步的完善。

Yourdon 方法是 80 年代使用最广泛的软件开发方法。它首先用结构化分析（SA）对软件进行需求分析，然后用结构化设计（SD）方法进行总体设计，最后是结构化编程（SP）。这一方法不仅开发步骤明确，SA、SD、SP 相辅相成，一气呵成，而且给出了两类典型的软件结构（变换型和事务型），便于参照，使软件开发的成功率大大提高，从而深受软件开发人员的青睐。

6.1.3 面向数据结构的软件开发方法

(1) Jackson 方法

1975 年，M．A．Jackson 提出了一类至今仍广泛使用的软件开发方法。这一方法从目标系统的输入、输出数据结构入手，导出程序框架结构，再补充其它细节，就可得到完整的程序结构图。这一方法对输入、输出数据结构明确的中小型系统特别有效，如商业应用中的文件表格处理。该方法也可与其它方法结合，用于模块的详细设计。

Jackson 方法有时也称为面向数据结构的软件设计方法。

(2) Warnier 方法

1974 年，J．D．Warnier 提出的软件开发方法与 Jackson 方法类似。差别有三点：一是它们使用的图形工具不同，分别使用 Warnier 图和 Jackson 图；另一个差别是使用的伪码不同；最主要的差别是在构造程序框架时，Warnier 方法仅考虑输入数据结构，而 Jackson 方法不仅考虑输入数据结构，而且还考虑输出数据结构。

6.1.4 问题分析法

PAM 问题分析法。PAM（Problem Analysis Method）是 80 年代末由日立公司提出的一种软件开发方法。

PAM 方法希望能兼顾 Yourdon 方法、Jackson 方法和自底向上的软件开发方法的优点，而避免它们的缺陷。它的基本思想是：考虑到输入、输出数据结构，指导系统的分解，在系统分析指导下逐步综合。这一方法的具体步骤是：从输入、输出数据结构导出基本处理框；分析这些处理框之间的先后关系；按先后关系逐步综合处理框，直到画出整个系统的 PAD 图。从上述步骤中可以看出，这一方法本质上是综合的自底向上的方法，但在逐步综合之前已进行了有目的的分解，这个目的就是充分考虑系统的输入、输出数据结构。

PAM 方法的另一个优点是使用 PAD 图。这是一种二维树形结构图，是到目前为止最好的详细设计表示方法之一，远远优于 NS 图和 PDL 语言。

这一方法在日本较为流行，软件开发的成功率也很高。由于在输入、输出数据结构与整个系统之间同样存在着鸿沟，这一方法仍只适用于中小型问题。

6.1.5 面向对象的软件开发方法

随着 OOP（面向对象编程）向 OOD（面向对象设计）和 OOA（面向对象分析）的发展，最终形成面向对象的软件开发方法 OMT(Object Modelling Technique)。这是一种自底向上和自顶向下相结合的方法，而且它以对象建模为基础，从而不仅考虑了输入、输出数据结构，实际上也包含了所有对象的数据结构。所以 OMT 彻底实现了 PAM 没有完全实现的目标。不仅如此，OO 技术在需求分析、可维护性和可靠性这三个软件开发的关键环节和质量指标上有了实质性的突破，彻底地解决了在这些方面存在的严重问题，从而宣告了软件危机末日的来临。

面向对象技术是软件技术的一次革命，在软件开发史上具有里程碑的意义。面向对象的基本思想是以对象为核心，通过对事物本身特性的抽象，产生可维护性高、可复用性强、容易扩展的程序代码，也是本章主要所述的内容，后面将对面向对象的软件开发技术进行

更详细一些的讲述。

6.1.6 可视化开发方法

可视化开发是 90 年代软件界最大的两个热点之一。随着图形用户界面的兴起，用户界面在软件系统中所占的比例也越来越大，有的甚至高达 60～70 %。产生这一问题的原因是图形界面元素的生成很不方便。为此 Windows 提供了应用程序设计接口 API（Application Programming Interface），它包含了 600 多个函数，极大地方便了图形用户界面的开发。但是在这批函数中，大量的函数参数和使用数量更多的有关常量，使基于 Windows API 的开发变得相当困难。

为此许多公司都推出了可视化编程工具，如：Mecrosoft 的 VisualC++、Visual Basic，Borland 公司的 Delphi、C++ Builder，IBM 公司的 Power Builder 以及 JAVA Beans 等。大多数的可视化开发环境都将 API 的各部分用对象类进行封装，并为这些定义了许多成员函数。利用子类对父类的继承性，以及实例对类的函数的引用，应用程序的开发可以省却大量类的定义，省却大量成员函数的定义或只需作少量修改以定义子类。

可视化开发就是在可视开发工具提供的图形用户界面上，通过操作界面元素，诸如菜单、按钮、对话框、编辑框、单选框、复选框、列表框和滚动条等，由可视开发工具自动生成应用软件。

这类应用软件的工作方式是事件驱动（Events Driver Program）。对每一事件，由系统产生相应的消息，再传递给相应的消息响应函数。这些消息响应函数是由可视开发工具在生成软件时自动装入的。

可视开发工具应提供两大类服务。一类是生成图形用户界面及相关的消息响应函数。通常的方法是先生成基本窗口，并在它的外面以图标形式列出所有其它的界面元素，让开发人员挑选后放入窗口指定位置。在逐一安排界面元素的同时，还可以用鼠标拖动，以使窗口的布局更趋合理。

另一类服务是为各种具体的子应用的各个常规执行步骤提供规范窗口，它包括对话框、菜单、列表框、组合框、按钮和编辑框等，以供用户挑选。开发工具还应为所有的选择（事件）提供消息响应函数。

由于要生成与各种应用相关的消息响应函数，因此，可视化开发只能用于相当成熟的应用领域，如目前流行的可视化开发工具基本上都是多用于关系数据库的开发。对一般的应用，目前的可视化开发工具只能提供用户界面的可视化开发。至于消息响应函数（或称脚本），则仍需用通常的高级语言（3GL）编写。只有在数据库领域才提供 4GL，使消息响应函数的开发大大简化。

从原理上讲，与图形有关的所有应用都可采用可视化开发方式。而且目前不光是在 Windows 系列操作系统出现了大量的可视化开发环境，在其他的操作系统中也逐渐出现了可视化的开发环境,如：Linux 中有 Borland 公司基于 Object Pascal 语言的 Kylix，通常被人们成为 Linux 中的 Delphi。

实际上可视化编程也是面向对象的一种设计语言，但比面向对象似乎更高级了一些，

本书将要讲的面向状态正是在可视化开发方法中产生的，但它并不局限于可视化的开发过程，而且也并不是所有的可视化开发环境都支持面向状态的设计方法。

6.2 软件的生命周期

软件的生命周期指的是软件从开发一直到不再使用时的这一段存在时间，一般来讲在结构化程序设计以及之前的软件设计中，软件的生命周期是呈现瀑布状，阶梯形式，通常可以分作 5 个阶段如图 2.13 所示：

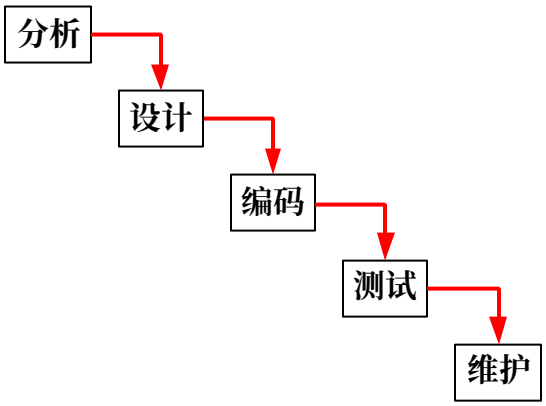


图 2.13 软件开发的生命周期和关系

分析阶段——定义并描述系统的外部特征。

设计阶段——将整个系统划分成为一系列简单的子系统，并定义每个子系统的功能和接口。

编码阶段——用某种程序设计语言将每个子系统写成程序代码，并将各个子系统集成为完整的系统。

测试阶段——对系统进行测试，证明系统达到了预期的目标。

维护阶段——对系统进行修改和扩充，以适应新的应用需求。

而在面向对象的设计方法中则有所不同，软件的生命周期通常被划分为 4 个阶段，如图 2.14 所示：

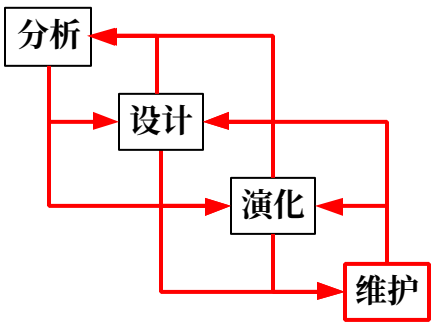


图2.14 面向对象的软件生命周期

分析阶段——在面向对象设计中，把分析看作是从问题域中选出词汇建立类和对象的模型世界。

设计阶段——设计是对问题域的行为进行关键抽象再分解的过程。设计的结果反馈到分析上进行纠正，当关键抽象足够简单不需要再进行分解时即可停止设计。

演化阶段——演化实际上是由编码、测试和集成组合在一起的阶段。

维护阶段——维护是在系统交付使用后的变更活动。

实际上，面向对象的这四个阶段并没有明确的界限，从时间上讲也是交叉进行的，因此严格的划分这四个阶段是没有任何实际意义的。一般来说，进行面向对象的程序设计需要经过以下几个步骤：

(1) 发现对象

前面我们讲过，对象是经过抽象的，具有唯一的名称，可以独力一个事物的实体，因此发现对象就是一个抽象的过程，将相似的或者相同的事物抽象成类，这些类将具有共同特征和行为，类的不同实例就是对象。在现有的开发工具中，大多数提供了许多基本类库，抽象是应该多从已有的类出发，通过派生和继承等组合而形成新的可用的类。

(2) 发现对象之间的关系

在抽象出类和对象之后，就是确定对象和对象之间的关系。将这些对象的特征和行为有机的组合起来就形成了我们程序结构。因此发现对象之间的关系是一个集成或者叫整合的过程，这个过程在一些具有简单功能的对象之间建立一些特定的关系，从而将其整合成为一个具有复杂功能的实体，这就是程序。

(3) 编码

编码就是采用一种计算机语言来将上面这两个过程的结构描述成计算机可以识别的信息，如采用 C++语言或 Delphi 等编写程序代码。

6.3 面向对象开发的特征

上面所说的三个步骤，在实现的过程中可能会复杂的多，而且也可以由不同的方法来实现。比如分析就有“ 综合法 ”和“ 分析法 ”两种截然不同方法。下面就面向对象技术（OMT）中一些常用的方法和特征作一介绍。

(1) 自底向上的归纳

OMT 的第一步是从问题的陈述入手，构造系统模型。从真实系统导出类的体系，即对象模型包括类的属性，与子类、父类的继承关系，以及类之间的关联。类是具有相似属性和行为的一组具体实例（客观对象）的抽象，父类是若干子类的归纳。因此这是一种自底向上的归纳过程。在自底向上的归纳过程中，为使子类能更合理地继承父类的属性和行为，可能需要自顶向下的修改，从而使整个类体系更加合理。由于这种类体系的构造是从具体到抽象，再从抽象到具体，符合人类的思维规律，因此能更快、更方便地完成任务。这与自顶向下的 Yourdon 方法构成鲜明的对照。在 Yourdon 方法中构造系统模型是最困难的一步，因为自顶向下的“ 顶 ”是一个空中楼阁，缺乏坚实的基础，而且功能分解有相当大的任意性，因此需要开发人员有丰富的软件开发经验。而在 OMT 中这一工作可由一般开发人员较快地完成。在对象模型建立后，很容易在这一基础上再导出动态模型和功能模型。这三个模型一起构成要求解的系统模型。

(2) 自顶向下的分解

系统模型建立后的工作就是分解。与 Yourdon 方法按功能分解不同，在 OMT 中通常按服务（Service）来分解。服务是具有共同目标的相关功能的集合，如 I/O 处理、图形处理等。这一步的分解通常很明确，而这些子系统的进一步分解因有较具体的系统模型为依据，也相对容易。所以 OMT 也具有自顶向下方法的优点，即能有效地控制模块的复杂性，同时避免了 Yourdon 方法中功能分解的困难和不确定性。

(3) OMT 的基础是对象模型

每个对象类由数据结构（属性）和操作（行为）组成，有关的所有数据结构（包括输入、输出数据结构）都成了软件开发的依据。因此 Jackson 方法和 PAM 中输入、输出数据结构与整个系统之间的鸿沟在 OMT 中不再存在。OMT 不仅具有 Jackson 方法和 PAM 的优点，而且可以应用于大型系统。更重要的是，在 Jackson 方法和 PAM 方法中，当它们的出发点—输入、输出数据结构（即系统的边界）发生变化时，整个软件必须推倒重来。但在 OMT 中系统边界的改变只是增加或减少一些对象而已，整个系统改动极小。

(4) 需求分析彻底

需求分析不彻底是软件失败的主要原因之一。即使在目前，这一危险依然存在。传统的软件开发方法不允许在开发过程中用户的需求发生变化，从而导致种种问题。正是由于这一原因，人们提出了原型化方法，推出探索原型、实验原型和进化原型，积极鼓励用户改进需求。在每次改进需求后又形成新的进化原型供用户试用，直到用户基本满意，大大提高了软件的成功率。但是它要求软件开发人员能迅速生成这些原型，这就要求有自动生成代码的工具的支持。

OMT 彻底解决了这一问题。因为需求分析过程已与系统模型的形成过程一致，开发人员与用户的讨论是从用户熟悉的具体实例（实体）开始的。开发人员必须搞清现实系统才能导出系统模型，这就使用户与开发人员之间有了共同的语言，避免了传统需求分析中可能产生的种种问题。

(5) 可维护性大大改善

在 OMT 之前的软件开发方法都是基于功能分解的。尽管软件工程学在可维护方面作出了极大的努力，使软件的可维护性有较大的改进。但从本质上讲，基于功能分解的软件是不易维护的。因为功能一旦有变化都会使开发的软件系统产生较大的变化，甚至推倒重来。更严重的是，在这种软件系统中，修改是困难的。由于种种原因，即使是微小的修改也可能引入新的错误。所以传统开发方法很可能会引起软件成本增长失控、软件质量得不到保证等一系列严重问题。正是 OMT 才使软件的可维护性有了质的改善。

OMT 的基础是目标系统的对象模型，而不是功能的分解。功能是对象的使用，它依赖于应用的细节，并在开过程中不断变化。由于对象是客观存在的，因此当需求变化时对象的性质要比对象的使用更为稳定，从而使建立在对象结构上的软件系统也更为稳定。

更重要的是 OMT 彻底解决了软件的可维护性。在 OO 语言中，子类不仅可以继承父类的属性和行为，而且也可以重载父类的某个行为（虚函数）。利用这一特点，我们可以方便地进行功能修改：引入某类的一个子类，对要修改的一些行为（即虚函数或虚方法）进行重载，也就是对它们重新定义。由于不再在原来的程序模块中引入修改，所以彻底解决了软件的可修改性，从而也彻底解决了软件的可维护性。OO 技术还提高了软件的可靠性

和健壮性。

7 小结

本章简单的介绍了面向对象的程序设计方法，其目的是使读者具备面向对象的基本知识，因为面向对象是面向状态的基础。但这不是本书的主要内容，已经熟悉掌握 C++语言的读者可以跳过本章，从第三章阅读，需要作更深入了解的读者，可以参考其他专门介绍面向对象的书籍。本章也介绍了一些常用的程序设计方法，可以帮助读者更深入的理解和使用面向状态技术，需要详细了解的读者应该查阅相关资料。

从下一章开始将进入本书的主题，介绍面向状态的思想和设计方法，其中很多内容都是作者的一些经验的总结，希望能对读者有所帮助。