第3篇

基于对象的程序设计

第8章 类和对象

第9章 关于类和对象的进一步讨论

第10章 运算符重载

第8章 类和对象

8.1 面向对象程序设计方法概述

到目前为止，我们介绍的是C++在面向过程的程序设计中的应用。对于规模比较小的程序，编程者可以直接编写出一个面向过程的程序，详细地描述每一瞬时的数据结构及对其的操作过程。但是当程序规模较大时，就显得力不从心了。C++就是为了解决编写大程序过程中的困难而产生的。

8.1.1 什么是面向对象的程序设计

面向对象的程序设计的思路和人们日常生活中处理问题的思路是相似的。在自然世界和社会生活中，一个复杂的事物总是由许多部分组成的。

当人们生产汽车时，分别设计和制造发动机、底盘、车身和轮子，最后把它们组装在一起。在组装时，各部分之间有一定的联系，以便协调工作。

这就是面向对象的程序设计的基本思路。

为了进一步说明问题，下面先讨论几个有关的概念。

1. 对象

客观世界中任何一个事物都可以看成一个对象**(object)**。

对象可大可小。对象是构成系统的基本单位。

任何一个对象都应当具有这两个要素，即属性**(attribute)**和行为**(behavior)**，它能根据外界给的信息进行相应的操作。一个对象往往是由一组属性和一组行为构成的。一般来说，凡是具备属性和行为这两种要素的，都可以作为对象。

在一个系统中的多个对象之间通过一定的渠道相互联系，如图**8.1**示意。要使某一个对象实现某一种行为(即操作)，应当向它传送相应的消息。对象之间就是这样通过发送和接收消息互相联系的。

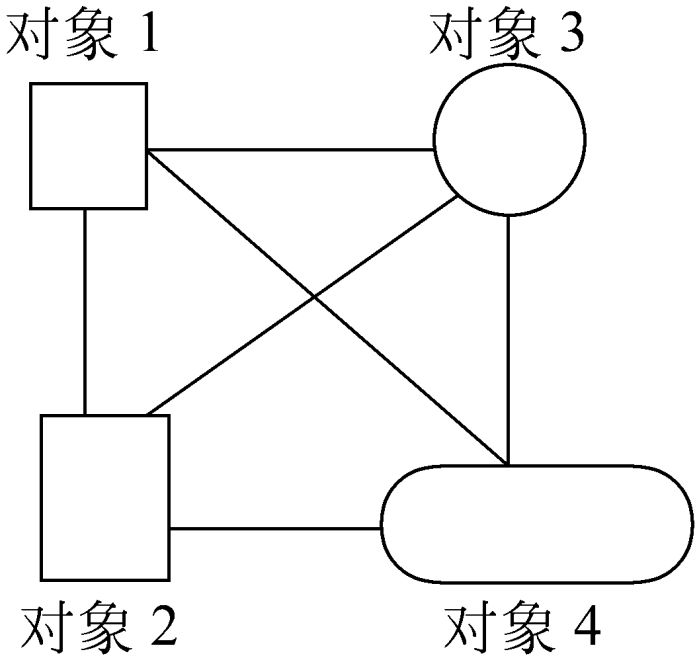

图8.1

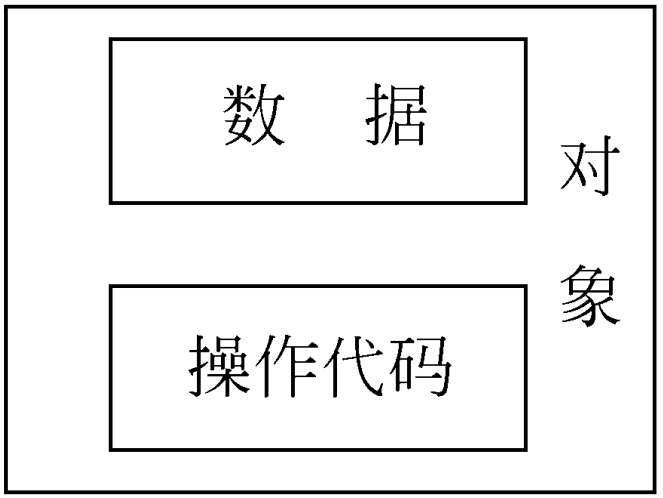

图8.2

面向对象的程序设计采用了以上人们所熟悉的这种思路。使用面向对象的程序设计方法设计一个复杂的软件系统时，首要的问题是确定该系统是由哪些对象组成的，并且设计这些对象。在**C++**中，每个对象都是由数据和函数**(**即操作代码**)**这两部分组成的**,**见图**8.2**。数据体现了前面提到的**“**属性**”**，如一个三角形对象，它的**3**个边长就是它的属性。函数是用来对数据进行操作的，以便实现某些功能，例如可以通过边长计算出三角形的面积，并且输出三角形的边长和面积。计算三角形面积和输出有关数据就是前面提到的行为，在程序设计方法中也称为方法**(method)**。调用对象中的函数就是向该对象传送一个消息**(message)**，要求该对象实现某一行为**(**功能**)**。

2. 封装与信息隐蔽

可以对一个对象进行封装处理，把它的一部分属性和功能对外界屏蔽，也就是说从外界是看不到的，甚至是不可知的。

这样做的好处是大大降低了操作对象的复杂程度。

面向对象程序设计方法的一个重要特点就是“封装性” **(encapsulation)**，所谓“封装”，指两方面的含义： 一是将有关的数据和操作代码封装在一个对象中，形成一个基本单位，各个对象之间相对独立，互不干扰。二是将对象中某些部分对外隐蔽，即隐蔽其内部细节，只留下少量接口，以便与外界联系，接收外界的消息。这种对外界隐蔽的做法称为信息隐蔽**(imformation hiding)**。信息隐蔽还有利于数据安全，防止无关的人了解和修改数据。

C++的对象中的函数名就是对象的对外接口，外界可以通过函数名来调用这些函数来实现某些行为(功能)。这些将在以后详细介绍。

3. 抽象

在程序设计方法中，常用到抽象(abstraction）这一名词。抽象的过程是将有关事物的共性归纳、集中的过程。

抽象的作用是表示同一类事物的本质。C和C++中的数据类型就是对一批具体的数的抽象。

对象是具体存在的，如一个三角形可以作为一个对象，10个不同尺寸的三角形是10个对象。如果这10个三角形对象有相同的属性和行为，可以将它们抽象为一种类型，称为三角形类型。在C++中，这种类型就称为“类(class)”。这10个三角形就是属于同一“类”的对象。类是对象的抽象，而对象则是类的特例，或者说是类的具体表现形式。

4. 继承与重用

如果在软件开发中已经建立了一个名为**A**的“类”，又想另外建立一个名为**B**的“类”，而后者与前者内容基本相同，只是在前者的基础上增加一些属性和行为，只需在类**A**的基础上增加一些新内容即可。这就是面向对象程序设计中的继承机制。利用继承可以简化程序设计的步骤。

“白马”继承了“马”的基本特征，又增加了新的特征(颜色)，“马”是父类，或称为基类，“白马”是从“马”派生出来的，称为子类或派生类。

C++提供了继承机制，采用继承的方法可以很方便地利用一个已有的类建立一个新的类。这就是常说的“软件重用”**(software reusability)** 的思想。

5. 多态性

如果有几个相似而不完全相同的对象，有时人们要求在向它们发出同一个消息时，它们的反应各不相同，分别执行不同的操作。这种情况就是多态现象。如，在**Windows**环境下，用鼠标双击一个文件对象(这就是向对象传送一个消息)，如果对象是一个可执行文件，则会执行此程序，如果对象是一个文本文件，则启动文本编辑器并打开该文件。

在**C++**中，所谓多态性**(polymorphism)**是指： 由继承而产生的相关的不同的类，其对象对同一消息会作出不同的响应。多态性是面向对象程序设计的一个重要特征，能增加程序的灵活性。

8.1.2 面向对象程序设计的特点

传统的面向过程程序设计是围绕功能进行的，用一个函数实现一个功能。所有的数据都是公用的，一个函数可以使用任何一组数据，而一组数据又能被多个函数所使用（见图**8.3**）。

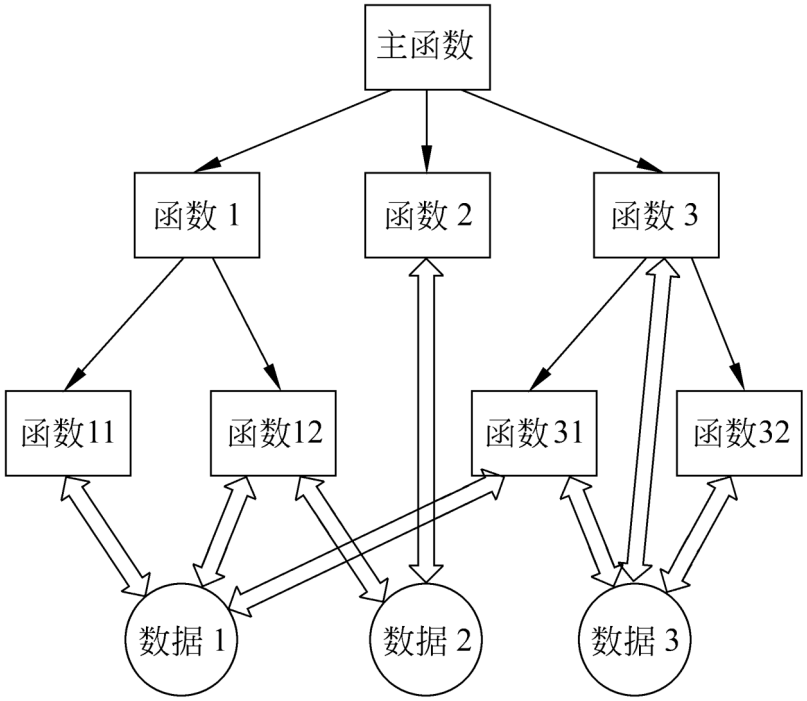

图**8.3**

面向对象程序设计采取的是另外一种思路。它面对的是一个个对象。实际上，每一组数据都是有特定的用途的，是某种操作的对象。也就是说，一组操作调用一组数据。

程序设计者的任务包括两个方面： 一是设计所需的各种类和对象，即决定把哪些数据和操作封装在一起；二是考虑怎样向有关对象发送消息，以完成所需的任务。这时他如同一个总调度，不断地向各个对象发出命令，让这些对象活动起来(或者说激活这些对象)，完成自己职责范围内的工作。各个对象的操作完成了，整体任务也就完成了。显然，对一个大型任务来说，面向对象程序设计方法是十分有效的，它能大大降低程序设计人员的工作难度，减少出错机会。

8.1.3 类和对象的作用

类是C++中十分重要的概念，它是实现面向对象程序设计的基础。类是所有面向对象的语言的共同特征，所有面向对象的语言都提供了这种类型。一个有一定规模的C++程序是由许多类所构成的。

C++支持面向过程的程序设计，也支持基于对象的程序设计，又支持面向对象的程序设计。在本章到第10章将介绍基于对象的程序设计。包括类和对象的概念、类的机制和声明、类对象的定义与使用等。这是面向对象的程序设计的基础。

基于对象就是基于类。与面向过程的程序不同，基于对象的程序是以类和对象为基础的，程序的操作是围绕对象进行的。在此基础上利用了继承机制和多态性，就成为面向对象的程序设计(有时不细分基于对象程序设计和面向对象程序设计，而把二者合称为面向对象的程序设计)。

基于对象程序设计所面对的是一个个对象。所有的数据分别属于不同的对象。

在面向过程的结构化程序设计中，人们常使用这样的公式来表述程序：

程序=算法＋数据结构

算法和数据结构两者是互相独立、分开设计的，面向过程的程序设计是以算法为主体的。在实践中人们逐渐认识到算法和数据结构是互相紧密联系不可分的，应当以一个算法对应一组数据结构，而不宜提倡一个算法对应多组数据结构，以及一组数据结构对应多个算法。基于对象和面向对象程序设计就是把一个算法和一组数据结构封装在一个对象中。因此，就形成了新的观念：

对象 = 算法 + 数据结构

程序 = (对象+对象+对象+...) + 消息　　或：

程序 = 对象s + 消息

“对象s”表示多个对象。消息的作用就是对对象的控制。程序设计的关键是设计好每一个对象，及确定向这些对象发出的命令，使各对象完成相应操作。

8.1.4 面向对象的软件开发

随着软件规模的迅速增大，软件人员面临的问题十分复杂。需要规范整个软件开发过程，明确软件开发过程中每个阶段的任务，在保证前一个阶段工作的正确性的情况下，再进行下一阶段的工作。这就是软件工程学需要研究和解决的问题。

面向对象的软件工程包括以下几个部分：

1. 面向对象分析(object oriented analysis,OOA)

软件工程中的系统分析阶段，系统分析员要和用户结合在一起，对用户的需求作出精确的分析和明确的描述，从宏观的角度概括出系统应该做什么(而不是怎么做)。面向对象的分析，要按照面向对象的概念和方法，在对任务的分析中，从客观存在的事物和事物之间的关系，归纳出有关的对象(包括对象的属性和行为)以及对象之间的联系，并将具有相同属性和行为的对象用一个类(class)来表示。建立一个能反映真实工作情况的需求模型。

2. 面向对象设计(object oriented design,OOD)

根据面向对象分析阶段形成的需求模型，对每一部分分别进行具体的设计，首先是进行类的设计，类的设计可能包含多个层次(利用继承与派生)。然后以这些类为基础提出程序设计的思路和方法，包括对算法的设计。在设计阶段，并不牵涉某一种具体的计算机语言，而是用一种更通用的描述工具(如伪代码或流程图)来描述。

3. 面向对象编程(object oriented programming, OOP)

根据面向对象设计的结果，用一种计算机语言把它写成程序，显然应当选用面向对象的计算机语言(例如C++)，否则无法实现面向对象设计的要求。

4. 面向对象测试(object oriented test,OOT)

在写好程序后交给用户使用前，必须对程序进行严格的测试。测试的目的是发现程序中的错误并改正它。面向对象测试是用面向对象的方法进行测试，以类作为测试的基本单元。

5. 面向对象维护(object oriented soft maintenance, OOSM)

因为对象的封装性，修改一个对象对其他对象影响很小。利用面向对象的方法维护程序，大大提高了软件维护的效率。

现在设计一个大的软件，是严格按照面向对象软件工程的**5**个阶段进行的，这**5**个阶段的工作不是由一个人从头到尾完成的，而是由不同的人分别完成的。这样，**OOP**阶段的任务就比较简单了，程序编写者只需要根据**OOD**提出的思路用面向对象语言编写出程序即可。在一个大型软件的开发中，**OOP**只是面向对象开发过程中的一个很小的部分。

如果所处理的是一个较简单的问题，可以不必严格按照以上**5**个阶段进行，往往由程序设计者按照面向对象的方法进行程序设计，包括类的设计(或选用已有的类)和程序的设计。

8.2 类的声明和对象的定义

8.2.1 类和对象的关系

每一个实体都是对象。有一些对象是具有相同的结构和特性的。每个对象都属于一个特定的类型。

在**C++**中对象的类型称为类**(class)**。类代表了某一批对象的共性和特征。前面已说明： 类是对象的抽象，而对象是类的具体实例**(instance)**。正如同结构体类型和结构体变量的关系一样，人们先声明一个结构体类型，然后用它去定义结构体变量。同一个结构体类型可以定义出多个不同的结构体变量。

在**C++**中也是先声明一个类类型，然后用它去定义若干个同类型的对象。对象就是类类型的一个变量。可以说类是对象的模板，是用来定义对象的一种抽象类型。

类是抽象的，不占用内存，而对象是具体的，占用存储空间。在一开始时弄清对象和类的关系是十分重要的。

8.2.2 声明类类型

类是用户自己指定的类型。如果程序中要用到类类型，必须自己根据需要进行声明，或者使用别人已设计好的类。C++标准本身并不提供现成的类的名称、结构和内容。

在C++中声明一个类类型和声明一个结构体类型是相似的。

下面是声明一个结构体类型的方法：

```
struct Student                    //声明了一个名为Student的结构体类型
{ int num;
char name[20];
char sex;
};
Student  stud1，stud2;            //定义了两个结构体变量stud1和stud2
```

它只包括数据，没有包括操作。现在声明一个类：

```
class Student                    //以class开头
{ int num;
char name[20];
char sex;                        //以上3行是数据成员
void display( )                  //这是成员函数
{cout<<"num:"<<num<<endl;
cout<<"name:"<<name<<endl;
cout<<"sex:"<<sex<<endl;         //以上4行是函数中的操作语句
}
};
Student  stud1，stud2;           //定义了两个Student 类的对象stud1和stud2
```

可以看到声明类的方法是由声明结构体类型的方法发展而来的。

可以看到，类(class)就是对象的类型。实际上，类是一种广义的数据类型。类这种数据类型中的数据既包含数据，也包含操作数据的函数。

不能把类中的全部成员与外界隔离，一般是把数据隐蔽起来，而把成员函数作为对外界的接口。

可以将上面类的声明改为

```
class Student                    //声明类类型
{ private:                       //声明以下部分为私有的
int num;
char name[20];
char sex;
public:                          //声明以下部分为公用的
void display( )
{cout<<"num:"<<num<<endl;
cout<<"name:"<<name<<endl;
cout<<"sex:"<<sex<<endl; }
};
Student  stud1，stud2;           //定义了两个Student 类的对象
```

如果在类的定义中既不指定**private**，也不指定**public**，则系统就默认为是私有的。

归纳以上对类类型的声明，可得到其一般形式如下：

class 类名

{ private:

私有的数据和成员函数**;**

public:

公用的数据和成员函数**;**

};

private和**public**称为成员访问限定符**(member access specifier)**。

除了**private**和**public**之外，还有一种成员访问限定符**protected**(受保护的)，用**protected**声明的成员称为受保护的成员，它不能被类外访问(这点与私有成员类似)，但可以被派生类的成员函数访问。

在声明类类型时，声明为**private**的成员和声明为**public**的成员的次序任意，既可以先出现**private**部分，也可以先出现**public**部分。如果在类体中既不写关键字**private**，又不写**public**，就默认为**private**。在一个类体中，关键字**private**和**public**可以分别出现多次。每个部分的有效范围到出现另一个访问限定符或类体结束时(最后一个右花括号)为止。但是为了使程序清晰，应该养成这样的习惯： 使每一种成员访问限定符在类定义体中只出现一次。

在以前的C++程序中，常先出现private部分，后出现public部分，如上面所示。现在的C++程序多数先写public部分，把private部分放在类体的后部。这样可以使用户将注意力集中在能被外界调用的成员上，使阅读者的思路更清晰一些。

在C++程序中，经常可以看到类。为了用户方便，常用的C++编译系统往往向用户提供类库(但不属于C++语言的组成部分)，内装常用的基本的类，供用户使用。不少用户也把自己或本单位经常用到的类放在一个专门的类库中，需要用时直接调用，这样就减少了程序设计的工作量。

8.2.3 定义对象的方法

8.2.2节的程序段中，最后一行用已声明的**Student**类来定义对象，这种方法是很容易理解的。经过定义后，**stud1**和**stud2**就成为具有**Student**类特征的对象。**stud1**和**stud2**这两个对象都分别包括**Student**类中定义的数据和函数。

定义对象也可以有几种方法。

1. 先声明类类型，然后再定义对象

前面用的就是这种方法，如

Student stud1，stud2;　　//Student是已经声明的类类型

在**C++**中，声明了类类型后，定义对象有两种形式。

（**1**） **class** 类名 对象名

如 **class Student stud1,stud2;**

把**class**和**Student**合起来作为一个类名，用来定义对象。

（**2**） 类名 对象名

如 **Student stud1，stud2;**

直接用类名定义对象。这两种方法是等效的。第**1**种方法是从**C**语言继承下来的，第**2**种方法是**C++**的特色，显然第**2**种方法更为简捷方便。

2. 在声明类类型的同时定义对象

```
class Student                          //声明类类型
{ public:                              //先声明公用部分
  void display( )
   {cout<<"num:"<<num<<endl;
cout<<"name:"<<name<<endl;
cout<<"sex:"<<sex<<endl; }
private:                               //后声明私有部分
int num;
char name[20];
char sex;
}stud1，stud2;                         //定义了两个Student类的对象
```

在定义**Student**类的同时，定义了两个**Student** 类的对象。

3. 不出现类名，直接定义对象

```
class                          //无类名
{private:                       //声明以下部分为私有的
 ┆
 public:                       //声明以下部分为公用的
 ┆
}stud1，stud2;                 //定义了两个无类名的类对象
```

直接定义对象，在C++中是合法的、允许的，但却很少用，也不提倡用。在实际的程序开发中，一般都采用上面3种方法中的第1种方法。在小型程序中或所声明的类只用于本程序时，也可以用第2种方法。

在定义一个对象时，编译系统会为这个对象分配存储空间，以存放对象中的成员。

8.2.4 类和结构体类型的异同

C++增加了class类型后，仍保留了结构体类型(struct)，而且把它的功能也扩展了。C++允许用struct来定义一个类型。如可以将前面用关键字class声明的类类型改为用关键字struct：

```
struct Student                    //用关键字struct来声明一个类类型
{private:                         //声明以下部分为私有的
int num;                          //以下3行为数据成员
char name[20];
char sex;
public:                           //声明以下部分为公用的
 void display( )                  //成员函数
{cout<<"num:"<<num<<endl;
cout<<"name:"<<name<<endl;
cout<<"sex:"<<sex<<endl; }
};
Student  stud1，stud2;            //定义了两个Student类的对象
```

为了使结构体类型也具有封装的特征，**C++**不是简单地继承**C**的结构体，而是使它也具有类的特点，以便于用于面向对象程序设计。用**struct**声明的结构体类型实际上也就是类。

用**struct**声明的类，如果对其成员不作**private**或**public**的声明，系统将其默认为**public**。如果想分别指定私有成员和公用成员，则应用**private**或**public**作显式声明。而用**class**定义的类，如果不作**private**或**public**声明，系统将其成员默认为**private**，在需要时也可以自己用显式声明改变。

如果希望成员是公用的，使用**struct**比较方便，如果希望部分成员是私有的，宜用**class**。建议尽量使用**class**来建立类，写出完全体现**C++**风格的程序。

8.3 类的成员函数

8.3.1 成员函数的性质

类的成员函数(简称类函数)是函数的一种，它的用法和作用和第4章介绍过的函数基本上是一样的，它也有返回值和函数类型，它与一般函数的区别只是：它是属于一个类的成员，出现在类体中。它可以被指定为**private**(私有的)、**public**(公用的)或**protected**(受保护的)。在使用类函数时，要注意调用它的权限(它能否被调用)以及它的作用域(函数能使用什么范围中的数据和函数)。例如私有的成员函数只能被本类中的其他成员函数所调用，而不能被类外调用。

成员函数可以访问本类中任何成员(包括私有的和公用的)，可以引用在本作用域中有效的数据。

一般的做法是将需要被外界调用的成员函数指定为**public**，它们是类的对外接口。但应注意，并非要求把所有成员函数都指定为**public**。有的函数并不是准备为外界调用的，而是为本类中的成员函数所调用的，就应该将它们指定为**private**。这种函数的作用是支持其他函数的操作，是类中其他成员的工具函数**(utility function)**，类外用户不能调用这些私有的工具函数。

类的成员函数是类体中十分重要的部分。如果一个类中不包含成员函数，就等同于**C**语言中的结构体了，体现不出类在面向对象程序设计中的作用。

8.3.2 在类外定义成员函数

在前面已经看到成员函数是在类体中定义的。也可以在类体中只写成员函数的声明，而在类的外面进行函数定义。如

```
class Student
{ public:
void display( );                    //公用成员函数原型声明
private:
int num;
string name;
char sex;                           //以上3行是私有数据成员
};
void Student::display( )            //在类外定义display类函数
{cout<<"num:"<<num<<endl;           //函数体
cout<<"name:"<<name<<endl;
cout<<"sex:"<<sex<<endl;
}
Student stud1,stud2;                //定义两个类对象
```

注意： 在类体中直接定义函数时，不需要在函数名前面加上类名，因为函数属于哪一个类是不言而喻的。但成员函数在类外定义时，必须在函数名前面加上类名，予以限定**(qualifed)**，“**::**”是作用域限定符**(field qualifier)**或称作用域运算符，用它声明函数是属于哪个类的。

如果在作用域运算符“**::**”的前面没有类名，或者函数名前面既无类名又无作用域运算符“**::**”，如

::display( )　　或　display( )

则表示**display**函数不属于任何类，这个函数不是成员函数，而是全局函数，即非成员函数的一般普通函数。

类函数必须先在类体中作原型声明，然后在类外定义，也就是说类体的位置应在函数定义之前，否则编译时会出错。

虽然函数在类的外部定义，但在调用成员函数时会根据在类中声明的函数原型找到函数的定义（函数代码），从而执行该函数。

在类的内部对成员函数作声明，而在类体外定义成员函数，这是程序设计的一种良好习惯。如果一个函数，其函数体只有**2~3**行，一般可在声明类时在类体中定义。多于**3**行的函数，一般在类体内声明，在类外定义。

8.3.3 inline 成员函数

关于内置(inline)函数，已在第4章第4.5节中作过介绍。类的成员函数也可以指定为内置函数。

在类体中定义的成员函数的规模一般都很小，而系统调用函数的过程所花费的时间开销相对是比较大的。调用一个函数的时间开销远远大于小规模函数体中全部语句的执行时间。为了减少时间开销，如果在类体中定义的成员函数中不包括循环等控制结构，C++系统会自动将它们作为内置(inline)函数来处理。也就是说，在程序调用这些成员函数时，并不是真正地执行函数的调用过程(如保留返回地址等处理)，而是把函数代码嵌入程序的调用点。这样可以大大减少调用成员函数的时间开销。

C++要求对一般的内置函数要用关键字**inline**声明，但对类内定义的成员函数，可以省略**inline**，因为这些成员函数已被隐含地指定为内置函数。如

```
class Student
{public:
void display( )
{cout<<"num:"<<num<<endl;
cout<<"name:"<<name<<endl;
cout<<"sex:"<<sex<<endl;
}
private:
int num;
string name;
char sex;
};
```

其中第**3**行

```
void display( )
```

也可以写成

```
inline void display( )
```

将**display**函数显式地声明为内置函数。以上两种写法是等效的。对在类体内定义的函数，一般都省写**inline**。

应该注意的是： 如果成员函数不在类体内定义，而在类体外定义，系统并不把它默认为内置**(inline)**函数，调用这些成员函数的过程和调用一般函数的过程是相同的。如果想将这些成员函数指定为内置函数，应当用**inline**作显式声明。如

```
class Student
{ public:
inline void display( );              //声明此成员函数为内置函数
```

```
private:
int num;
string name;
char sex;
};
inline void Student::display( )        // 在类外定义display函数为内置函数
{cout<<"num:"<<num<<endl;
cout<<"name:"<<name<<endl;
cout<<"sex:"<<sex<<endl;
}
```

在第**4**章第**4.5**节曾提到过，在函数的声明或函数的定义两者之一作**inline**声明即可。值得注意的是：如果在类体外定义**inline**函数，则必须将类定义和成员函数的定义都放在同一个头文件中(或者写在同一个源文件中)，否则编译时无法进行置换(将函数代码的拷贝嵌入到函数调用点)。但是这样做，不利于类的接口与类的实现分离，不利于信息隐蔽。虽然程序的执行效率提高了，但从软件工程质量的角度来看，这样做并不是好的办法。

只有在类外定义的成员函数规模很小而调用频率较高时，才将此成员函数指定为内置函数。

8.3.4 成员函数的存储方式

用类去定义对象时，系统会为每一个对象分配存储空间。如果一个类包括了数据和函数，要分别为数据和函数的代码分配存储空间。按理说，如果用同一个类定义了**10**个对象，那么就需要分别为**10**个对象的数据和函数代码分配存储单元，如图**8.4**所示。

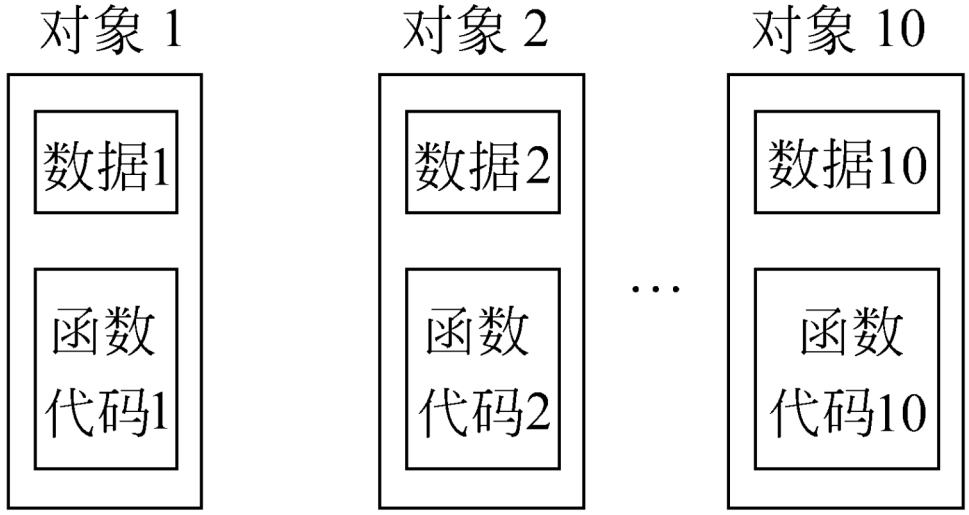

图**8.4**

能否只用一段空间来存放这个共同的函数代码段，在调用各对象的函数时，都去调用这个公用的函数代码。如图**8.5**所示。

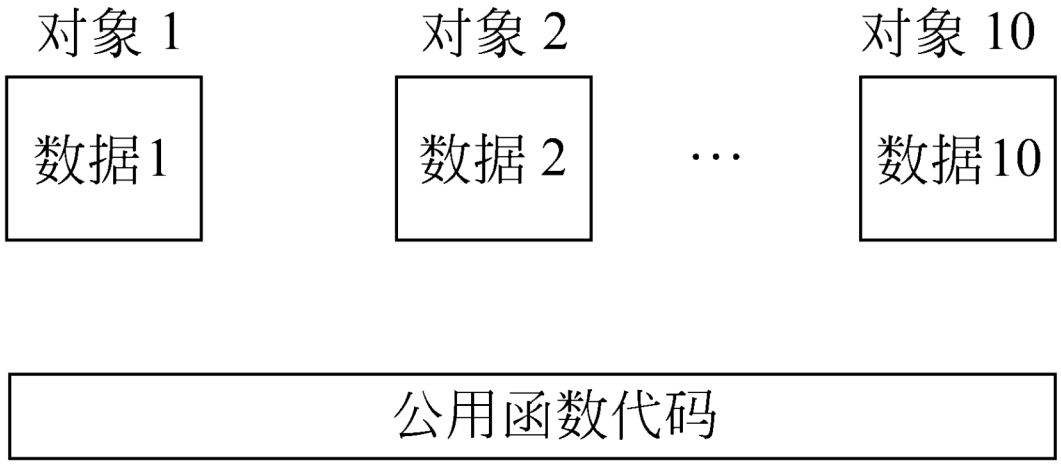

图8.5

显然，这样做会大大节约存储空间。**C++**编译系统正是这样做的，因此每个对象所占用的存储空间只是该对象的数据部分所占用的存储空间，而不包括函数代码所占用的存储空间。如果声明了一个类：

```
class Time
{public:
int hour;
int minute;
int sec;
void set( )
{cin>>a>>b>>c;}
};
```

可以用下面的语句来输出该类对象所占用的字节数：

```
cout<<sizeof(Time)<<endl;
```

输出的值是**12**。这就证明了一个对象所占的空间大小只取决于该对象中数据成员所占的空间，而与成员函数无关。函数代码是存储在对象空间之外的。如果对同一个类定义了**10**个对象，这些对象的成员函数对应的是同一个函数代码段，而不是**10**个不同的函数代码段。

需要注意的是： 虽然调用不同对象的成员函数时都是执行同一段函数代码，但是执行结果一般是不相同的。不同的对象使用的是同一个函数代码段，它怎么能够分别对不同对象中的数据进行操作呢？原来**C++**为此专门设立了一个名为**this**的指针，用来指向不同的对象。

需要说明：

(1) 不论成员函数在类内定义还是在类外定义，成员函数的代码段都用同一种方式存储。

(2) 不要将成员函数的这种存储方式和**inline**(内置)函数的概念混淆。

(3) 应当说明： 常说的“某某对象的成员函数”，是从逻辑的角度而言的，而成员函数的存储方式，是从物理的角度而言的，二者是不矛盾的。

8.4 对象成员的引用

在程序中经常需要访问对象中的成员。访问对象中的成员可以有3种方法：

- 通过对象名和成员运算符访问对象中的成员；
- 通过指向对象的指针访问对象中的成员；
- 通过对象的引用变量访问对象中的成员。

8.4.1 通过对象名和成员运算符访问对象中的成员

例如在程序中可以写出以下语句：

```
stud1.num=1001;          //假设num已定义为公用的整型数据成员
```

表示将整数**1001**赋给对象**stud1**中的数据成员**num**。其中“.”是成员运算符，用来对成员进行限定，指明所访问的是哪一个对象中的成员。注意不能只写成员名而忽略对象名。

访问对象中成员的一般形式为

对象名.成员名

不仅可以在类外引用对象的公用数据成员，而且还可以调用对象的公用成员函数，但同样必须指出对象名，如

```
stud1.display( );          //正确，调用对象stud1的公用成员函数
display( );                //错误，没有指明是哪一个对象的display函数
```

由于没有指明对象名，编译时把**display**作为普通函数处理。

应该注意所访问的成员是公用的**(public)**还是私有的**(private)**。只能访问**public**成员，而不能访问**private**成员，如果已定义**num**为私有数据成员，下面的语句是错误的：

```
stud1.num=10101;           //num是私有数据成员，不能被外界引用
```

在类外只能调用公用的成员函数。在一个类中应当至少有一个公用的成员函数，作为对外的接口，否则就无法对对象进行任何操作。

8.4.2 通过指向对象的指针访问对象中的成员

在第7章第7.1.5节中介绍了指向结构体变量的指针，可以通过指针引用结构体中的成员。用指针访问对象中的成员的方法与此类似。如果有以下程序段：

```
class Time
{public:                //数据成员是公用的
int hour;
int minute;
};
Time t,*p;              //定义对象t和指针变量p
p=&t;                   //使p指向对象t
cout<<p->hour;          //输出p指向的对象中的成员hour
```

在p指向t的前提下，p->hour，(*p).hour和t.hour三者等价

8.4.3 通过对象的引用变量来访问对象中的成员

如果为一个对象定义了一个引用变量，它们是共占同一段存储单元的，实际上它们是同一个对象，只是用不同的名字表示而已。因此完全可以通过引用变量来访问对象中的成员。

如果已声明了**Time**类，并有以下定义语句：

```
Time t1;              //定义对象t1
Time &t2=t1;          //定义Time类引用变量t2，并使之初始化为t1
cout<<t2.hour;        //输出对象t1中的成员hour
```

由于**t2**与**t1**共占同一段存储单元(即**t2**是**t1**的别名)，因此**t2.hour**就是**t1.hour**。

本章第**8.6**节的例**8.2**中的程序**(b)**,介绍的是引用变量作为形参的情况，读者可以参考。

8.5 类的封装性和信息隐蔽

8.5.1 公用接口与私有实现的分离

从前面的介绍已知： C++通过类来实现封装性，把数据和与这些数据有关的操作封装在一个类中，或者说，类的作用是把数据和算法封装在用户声明的抽象数据类型中。

在声明了一个类以后，用户主要是通过调用公用的成员函数来实现类提供的功能(例如对数据成员设置值，显示数据成员的值，对数据进行加工等)。因此，公用成员函数是用户使用类的公用接口(public interface)，或者说是类的对外接口。

当然并不一定要把所有成员函数都指定为**public**(公用)的，但这时这些成员函数就不是公用接口了。在类外虽然不能直接访问私有数据成员，但可以通过调用公用成员函数来引用甚至修改私有数据成员。

用户可以调用公用成员函数来实现某些功能，而这些功能是在声明类时已指定的，用户可以使用它们而不应改变它们。实际上用户往往并不关心这些功能是如何实现的细节，而只需知道调用哪个函数会得到什么结果，能实现什么功能即可。

通过成员函数对数据成员进行操作称为类的实现，为了防止用户任意修改公用成员函数，改变对数据进行的操作，往往不让用户看到公用成员函数的源代码，显然更不能修改它，用户只能接触到公用成员函数的目标代码(详见**8.5.2**节)。

可以看到： 类中被操作的数据是私有的，实现的细节对用户是隐蔽的，这种实现称为私有实现**(private implementation)**。这种“类的公用接口与私有实现的分离”形成了信息隐蔽。

软件工程的一个最基本的原则就是将接口与实现分离，信息隐蔽是软件工程中一个非常重要的概念。它的好处在于：

(1) 如果想修改或扩充类的功能，只需修改本类中有关的数据成员和与它有关的成员函数，程序中类外的部分可以不必修改。

(2) 如果在编译时发现类中的数据读写有错，不必检查整个程序，只需检查本类中访问这些数据的少数成员函数。

8.5.2 类声明和成员函数定义的分离

在面向对象的程序开发中，一般做法是将类的声明(其中包含成员函数的声明)放在指定的头文件中，用户如果想用该类，只要把有关的头文件包含进来即可，不必在程序中重复书写类的声明，以减少工作量，节省篇幅，提高编程的效率。

由于在头文件中包含了类的声明，因此在程序中就可以用该类来定义对象。由于在类体中包含了对成员函数的声明，在程序中就可以调用这些对象的公用成员函数。为了实现上一节所叙述的信息隐蔽，对类成员函数的定义一般不放在头文件中，而另外放在一个文件中。

例如，可以分别写两个文件：

```
//student.h              (这是头文件，在此文件中进行类的声明)
class Student                        //类声明
{ public:
void display( );                 //公用成员函数原型声明
private:
int num;
char name[20];
char sex;
};
//student.cpp                        //在此文件中进行函数的定义
#include <iostream>
#include "student.h"                 //不要漏写此行，否则编译通不过
void Student::display( )             //在类外定义display类函数
{cout<<"num:"<<num<<endl;
cout<<"name:"<<name<<endl;
cout<<"sex:"<<sex<<endl;
}
```

为了组成一个完整的源程序，还应当有包括主函数的源文件：

```
//main.cpp                    主函数模块
#include <iostream>
#include "student.h"              //将类声明头文件包含进来
int main( )
{Student stud;                    //定义对象
stud.display( );                  //执行stud对象的display函数
return 0;
}
```

这是一个包括3个文件的程序，组成两个文件模块：一个是主模块**main.cpp**，一个是**student.cpp**。在主模块中又包含头文件**student.h**。在预编译时会将头文件**student.h**中的内容取代**#include "student.h"**行。

请注意： 由于将头文件**student.h**放在用户当前目录中，因此在文件名两侧用双撇号包起来**("student.h")**而不用尖括号**(<student.h>)**，否则编译时会找不到此文件。

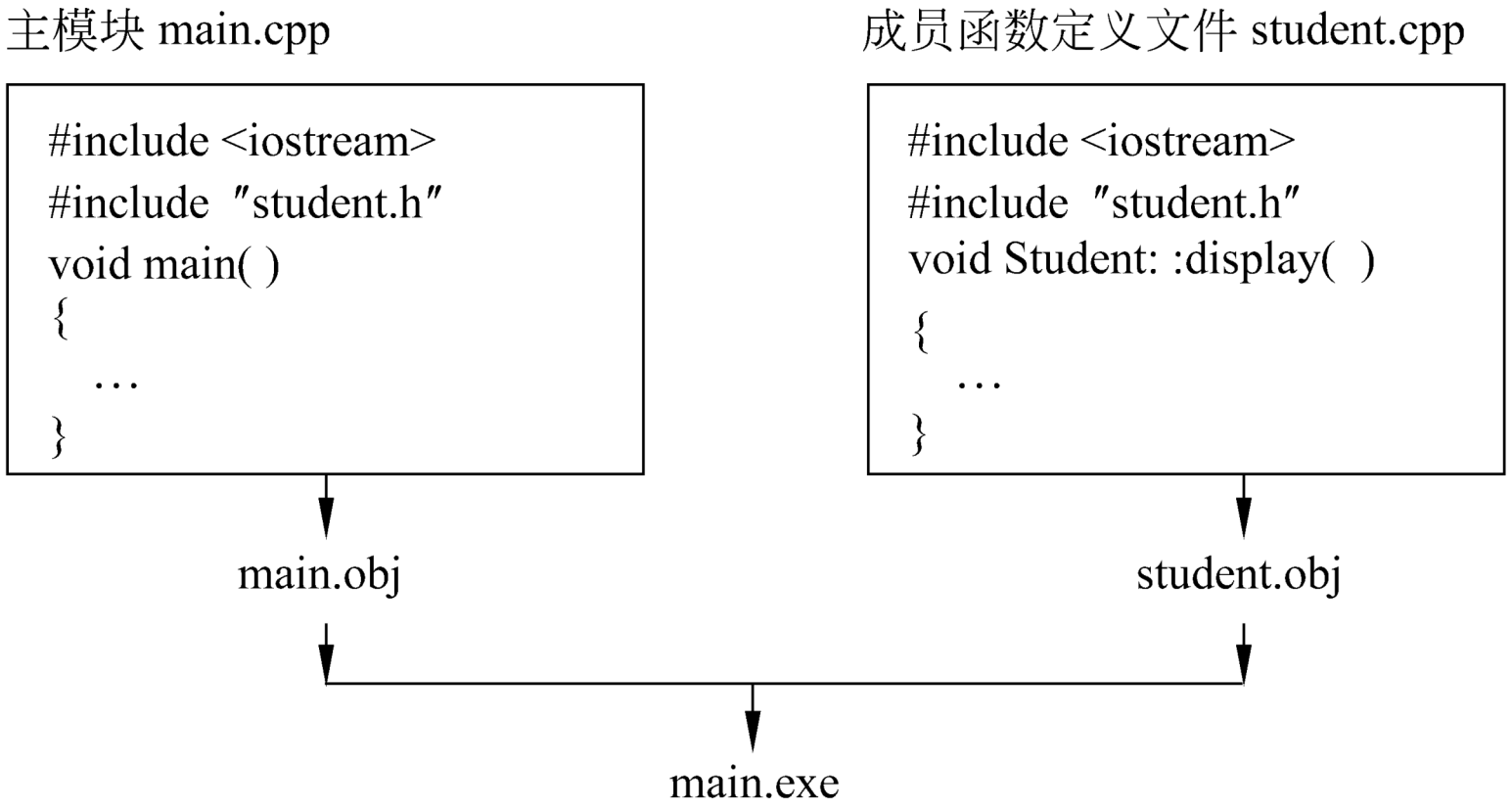

图**8.6**。

在运行程序时调用**stud**中的**display**函数，输出各数据成员的值。

如果一个类声明多次被不同的程序所选用，每次都要对包含成员函数定义的源文件(如上面的**student.cpp)**进行编译，这是否可以改进呢？的确，可以不必每次都对它重复进行编译，而只需编译一次即可。把第一次编译后所形成的目标文件保存起来，以后在需要时把它调出来直接与程序的目标文件相连接即可。这和使用函数库中的函数是类似的。

这也是把成员函数的定义不放在头文件中的一个好处。

在实际工作中，并不是将一个类声明做成一个头文件，而是将若干个常用的功能相近的类声明集中在

类库有两种： 一种是**C++**编译系统提供的标准类库；一种是用户根据自己的需要做成的用户类库，提供给自己和自己授权的人使用，这称为自定义类库。在程序开发工作中，类库是很有用的，它可以减少用户自己对类和成员函数进行定义的工作量。

类库包括两个组成部分： **(1)**类声明头文件； **(2)**已经过编译的成员函数的定义，它是目标文件。用户只需把类库装入到自己的计算机系统中**(**一般装到**C++**编译系统所在的子目录下**)**，并在程序中用**#include**命令行将有关的类声明的头文件包含到程序中，就可以使用这些类和其中的成员函数，顺利地运行程序。

这和在程序中使用C++系统提供的标准函数的方法是一样的，例如用户在调用sin函数时只需将包含声明此函数的头文件包含到程序中，即可调用该库函数，而不必了解sin函数是怎么实现的(函数值是怎样计算出来的)。当然，前提是系统已装了标准函数库。在用户源文件经过编译后，与系统库(是目标文件)相连接。

在用户程序中包含类声明头文件，类声明头文件就成为用户使用类的公用接口，在头文件的类体中还提供了成员函数的函数原型声明，用户只有通过头文件才能使用有关的类。用户看得见和接触到的是这个头文件，任何要使用这个类的用户只需包含这个头文件即可。包含成员函数定义的文件就是类的实现。请特别注意：类声明和函数定义一般是分别放在两个文本中的。

由于要求接口与实现分离，为软件开发商向用户提供类库创造了很好的条件。开发商把用户所需的各种类的声明按类放在不同的头文件中，同时对包含成员函数定义的源文件进行编译，得到成员函数定义的目标代码。软件商向用户提供这些头文件和类的实现的目标代码(不提供函数定义的源代码)。用户在使用类库中的类时，只需将有关头文件包含到自己的程序中，并且在编译后连接成员函数定义的目标代码即可。

由于类库的出现，用户可以像使用零件一样方便地使用在实践中积累的通用的或专用的类，这就大大减少了程序设计的工作量，有效地提高了工作效率。

8.5.3 面向对象程序设计中的几个名词

类的成员函数在面向对象程序理论中被称为“方法”(method)，“方法”是指对数据的操作。一个“方法”对应一种操作。显然，只有被声明为公用的方法（成员函数）才能被对象外界所激活。外界是通过发“消息”来激活有关方法的。所谓“消息”，其实就是一个命令，由程序语句来实现。前面的**stud.display( );**就是向对象**stud**发出的一个“消息”，通知它执行其中的**display**“方法”（即**display**函数）。上面这个语句涉及**3**个术语： 对象、方法和消息。**stud**是对象，**display( )**是方法，语句“**stud.display( );**”是消息。

8.6 类和对象的简单应用举例

例8.1 最简单的例子。

```
#include <iostream>
using namespace std;
class Time                    //定义Time类
{public:                      //数据成员为公用的
 int hour;
 int minute;
 int sec;
};

int main( )
{ Time t1;                    //定义t1为Time类对象
cin>>t1.hour;                 //输入设定的时间
cin>>t1.minute;
cin>>t1.sec;
cout<<t1.hour<<":"<<t1.minute<<":"<<t1.sec<<endl;    //输出时间
return 0;
}
```

运行情况如下：

1232 43↙

12:32:43

注意：

(1) 在引用数据成员**hour**，**minute**，**sec**时不要忘记在前面指定对象名。

(2) 不要错写为类名，如写成**Time.hour,Time.minute,Time.sec**是不对的。因为类是一种抽象的数据类型，并不是一个实体，也不占存储空间，而对象是实际存在的实体，是占存储空间的，其数据成员是有值的，可以被引用的。

(3) 如果删去主函数的**3**个输入语句，即不向这些数据成员赋值，则它们的值是不可预知的。

例8.2 引用多个对象的成员。

（1） 程序(a)

```
#include <iostream>
using namespace std;
class Time
{public:
int hour;
int minute;
int sec;
};
int main( )
{Time t1;                                    /定义对象t1
cin>>t1.hour;                                //向t1的数据成员输入数据
cin>>t1.minute;
cin>>t1.sec;
cout<<t1.hour<<":"<<t1.minute<<":"<<t1.sec<<endl;   //输出t1中数据成员的值
```

```
Time t2;                                    /定义对象t2
cin>>t2.hour;                               //向t2的数据成员输入数据
cin>>t2.minute;
cin>>t2.sec;
cout<<t2.hour<<":"<<t2.minute<<":"<<t2.sec<<endl;  //输出t2中数据成员的值
return 0;
}
```

运行情况如下：

```
1032  43↙
10:32:43
22  32  43↙
22:32:43
```

程序是清晰易懂的，但是在主函数中对不同的对象一一写出有关操作，会使程序冗长。为了解决这个问题，可以使用函数来进行输入和输出。见程序**(b)**。

（2） 程序(b)

```
#include <iostream>
using namespace std;
class Time
{public:
int hour;
int minute;
int sec;
};

int main( )
{
void set_time(Time&);    //函数声明
void show_time(Time&);    //函数声明
Time t1;                 //定义t1为Time类对象
set_time(t1);            //调用set_time函数，向t1对象中的数据成员输入数据
show_time(t1);           //调用show_time函数，输出t1对象中的数据
```

```
Time t2;                //定义t2为Time类对象
set_time(t2);           //调用set_time函数，向t2对象中的数据成员输入数据
show_time(t2);          //调用show_time函数，输出t2对象中的数据
return 0;
}

void set_time(Time& t)        //定义函数set_time，形参t是引用变量
{
 cin>>t.hour;                 //输入设定的时间
 cin>>t.minute;
 cin>>t.sec;
}

void show_time(Time& t)       //定义函数show_time，形参t是引用变量
{
 cout<<t.hour<<":"<<t.minute<<":"<<t.sec<<endl;    //输出对象中的数据
}
```

运行情况与程序(a)相同。

（3） 程序(c)

可以对上面的程序作一些修改，数据成员的值不再由键盘输入，而在调用函数时由实参给出，并在函数中使用默认参数。将程序(b)第8行以下部分改为

```
int main( )
{
 void set_time(Time&,int hour=0,int minute=0,int sec=0);  //函数声明
void show_time(Time&);                                    //函数声明
Time t1;
set_time(t1,12,23,34);                    //通过实参传递时、分、秒的值
show_time(t1);
Time t2;
set_time(t2);                             //使用默认的时、分、秒的值
show_time(t2);
return 0;
}
```

```
void set_time(Time& t,int hour,int minute,int sec)
{
t.hour=hour;
t.minute=minute;
t.sec=sec;
}

void show_time(Time& t)
{
cout<<t.hour<<":"<<t.minute<<":"<<t.sec<<endl;
}
```

程序运行时的输出为

12:23:34 (t1中的时、分、秒)

0:0:0 (t2中的时、分、秒)

以上两个程序中定义的类都只有数据成员，没有成员函数，这显然没有体现出使用类的优越性。在下面的例子中，类体中就包含了成员函数。

例8.3 将例8.2的程序改用含成员函数的类来处理。

```
#include <iostream>
using namespace std;
class Time
{public:
void set_time( );          //公用成员函数
void show_time( );         //公用成员函数
 private:                  //数据成员为私有
int hour;
int minute;
int sec;
};
int main( )
{
Time t1;            //定义对象t1
t1.set_time( );     //调用对象t1的成员函数set_time，向t1的数据成员输入数据
t1.show_time( );    //调用对象t1的成员函数show_time，输出t1的数据成员的值
Time t2;            //定义对象t2
```

```
t2.set_time( );       //调用对象t2的成员函数set_time，向t2的数据成员输入数据
t2.show_time( );      //调用对象t2的成员函数show_time，输出t2的数据成员的值
return 0;
}

void Time::set_time( )      //在类外定义set_time函数
{
 cin>>hour;
 cin>>minute;
 cin>>sec;
}

void Time::show_time( )      //在类外定义show_time函数
{
 cout<<hour<<":"<<minute<<":"<<sec<<endl;
}
```

运行情况与例**8.2**中的程序**(a)**相同。

注意：

(1) 在主函数中调用两个成员函数时，应指明对象名**(t1,t2)**。表示调用的是哪一个对象的成员函数。

(2) 在类外定义函数时，应指明函数的作用域(如**void Time∶∶set_time( )**)。在成员函数引用本对象的数据成员时，只需直接写数据成员名，这时**C++**系统会把它默认为本对象的数据成员。也可以显式地写出类名并使用域运算符。

(3) 应注意区分什么场合用域运算符“**::**”，什么场合用成员运算符“**.**”，不要搞混。

例**8.4** 找出一个整型数组中的元素的最大值。

这个问题可以不用类的方法来解决，现在用类来处理，读者可以比较不同方法的特点。

```
#include <iostream>
using namespace std;
class Array_max                    //声明类
{public:                          //以下3行为成员函数原型声明
void set_value( );                 //对数组元素设置值
void max_value( );                 //找出数组中的最大元素
void show_value( );                //输出最大值
private:
int array[10];                     //整型数组
int max;                          //max用来存放最大值
};

void Array_max::set_value( )      //成员函数定义，向数组元素输入数值
{ int i;
for (i=0;i<10;i++)

cin>>array[i];
}
```

```
void Array_max::max_value( )     //成员函数定义，找数组元素中的最大值
{int i;
max=array[0];
for (i=1;i<10;i++)
if(array[i]>max) max=array[i];
}

void Array_max::show_value( )    //成员函数定义，输出最大值
{cout<<"max="<<max;}

int main( )
{Array_max  arrmax;      //定义对象arrmax
arrmax.set_value( );     //调用arrmax的set_value函数，向数组元素输入数值
arrmax.max_value( );     //调用arrmax的max_value函数，找出数组元素中的最大值
arrmax.show_value( );    //调用arrmax的show_value函数，输出数组元素中的最大值
return 0;
}
```

运行结果如下：

12 12 39 -34 17 134 045 -91 76↙ **(输入10个元素的值)**

max=134 **(输入10个元素中的最大值)**

请注意成员函数定义与调用成员函数的关系，定义成员函数只是设计了一组操作代码，并未实际执行，只有在被调用时才真正地执行这一组操作。

可以看出： 主函数很简单，语句很少，只是调用有关对象的成员函数，去完成相应的操作。在大多数情况下，主函数中甚至不出现控制结构(判断结构和循环结构)，而在成员函数中使用控制结构。在面向对象的程序设计中，最关键的工作是类的设计。所有的数据和对数据的操作都体现在类中。只要把类定义好，编写程序的工作就显得很简单了。