第 3 章

类 与 对 象

抽象和封装是面向对象程序设计的两个重要特征，类是对象的抽象，对象是类的实例（或具体化）。在面向对象中，封装就是把描述对象属性的数据和加工处理这些数据的操作放在同一对象中，并使得对数据的访问处理只能通过对象本身来进行，程序的其他部分不能直接访问和处理对象的私有数据。封装相对于 C 语言中的局部变量、结构体等更好地实现了信息隐藏，在程序设计时应充分、合理地运用面向对象的封装机制。

通过本章的学习，可以编写出基于对象的 C++ 程序。基于对象的 C++ 程序以类和对象为基础，程序的操作是围绕对象进行的。在此基础上利用继承机制和多态性，就成为面向对象的程序设计（有时不区分基于对象的程序设计和面向对象的程序设计，而把两者合称为面向对象的程序设计）。

3.1 类的声明和对象的定义

在基于对象的 C++ 程序中是一个一个的对象在运行，而对象是由类实例化而得到的，在实际开发过程中是针对类进行编程的，因此首先要正确地理解类和对象的概念以及类和对象的关系。

3.1.1 类和对象的概念及其关系

在 1.2.2 节中已经较详细地讨论了面向对象的基本概念，此处不再赘述。下面回顾一下对象和类的概念及其关系。

1. 对象

对象就是封装了数据及在这些数据之上的操作的封装体，这个封装体有一个名字标识它，而且可以向外界提供一组操作（或服务）。

2. 类

类是对具有相同属性和操作的一组对象的抽象描述。

3. 类和对象的关系

类代表了一组对象的共性和特征，类是对象的抽象，即类忽略对象中具体的属性值而只保留属性。而对象是对类的实例化，即将类中的属性赋以具体的属性值得到一个具体的对象。类和对象的关系就像图纸和房屋的关系，类就像图纸，而对象就好比按照图纸建造的房屋。

特别提醒：

在介绍类和对象时一般都是从对象开始，先引入对象的概念然后再讨论类的概念。而在程序设计中，需要程序员先设计好类，然后在程序中再定义这些类的对象。

总之，类是抽象的，在C++中它是一种自定义的数据类型，而对象是具体的，在C++中它是“类”类型的变量，定义后系统是要为对象分配内存空间的。在进行基于对象和面向对象的程序设计时一定要搞清楚类和对象的概念及它们之间的关系。

3.1.2　类的声明

在C++中，类是一种用户自定义的数据类型，下面先来看声明类的一般形式：

```
class 类名
{public:
    公用成员
      ⋮
protected:
    受保护成员
      ⋮
private:
    私有成员
      ⋮
};
```

在声明类的一般形式中，class是声明类的关键字，后面跟着类名。花括号里是类的成员的声明，包括数据成员和成员函数。花括号后面要有分号，这个细节一定要注意。成员的可访问性分为3类：公用的(public)、受保护的(protected)和私有的(private)。其中关键字public、protected和private称为成员访问限定符，后面加上冒号。由访问限定符限定它后面的成员的访问属性，直到出现下一个访问限定符或者类声明结束为止。访问属性为公用的成员既可以被本类的成员函数访问，也可以在类的作用域内被其他函数访问。访问属性为受保护的成员可以被本类及其派生类的成员函数访问，但不能被类外访问。访问属性为私有的成员只能被本类的成员函数访问而不能被类外访问(类的友元例外)。

在声明类时，这3种访问属性的成员的声明次序是任意的，并且在一个类的声明中这3种访问属性不一定全部都出现，可能只出现两种或一种。某个成员访问限定符在一个类的声明中也可以出现多次。无论出现几次，成员的访问属性都是从一个成员访问限定符出现开始，到出现另一个成员访问限定符或类声明结束为止。如果在类声明的开始处没有写出成员访问限定符，则默认的成员访问属性是私有的。注意，为了使程序更清晰、更易读，应该养成这样的习惯：即每一种成员访问限定符在类的声明中只出现一次。

关于3种成员访问限定符出现的顺序问题，从语法上说无论谁在前谁在后效果都是完全一样的。但是传统的编程习惯是先出现私有成员，后出现公用成员。现在又提出新

的顺序是先写公用成员，后写私有成员，这样做的好处是可以突出能被外界访问的公用成员。这只是习惯问题，也不必强求。

【例 3-1】 声明一个学生类，要求包括学生的学号、姓名、性别、年龄等信息，并且能够从键盘输入学生信息，能够显示学生信息到屏幕。

```
class Student              //声明学生类 Student
{public:                   //以下部分为学生类 Student 的公用成员函数
    void setInfo()
    {    cout <<"Please enter student's No, name, sex, age: ";
         cin >>stuNo >>stuName >>stuSex >>stuAge;
    }
    void show()
    {    cout <<"No. : " <<stuNo <<endl;
         cout <<"Name: " <<stuName <<endl;
         cout <<"Sex: " <<stuSex <<endl;
         cout <<"Age: " <<stuAge <<endl;
    }
private:                   //以下部分为学生类 Student 的私有数据成员
    string stuNo;          //学号
    string stuName;        //姓名
    string stuSex ="男"; //性别
    int stuAge =0;         //年龄
};                         //类声明结束，此处必须有分号
```

例 3-1 中的学生类的学号、姓名、性别、年龄信息都声明为了私有的，这些信息在类外不能够直接访问，这样就可以保证数据的安全性。setInfo 函数和 show 函数都是公用成员函数，这样在类外可以通过学生类对象调用 setInfo 函数从键盘输入学生信息，调用 show 函数显示学生信息到屏幕。

C++11 标准规定：声明一个类时，可以为类的数据成员提供一个类内初始值。创建对象时，类内初始值将用于初始化数据成员。对类内初始值的限制：或者放在花括号里，或者放在等号右边，记住不能使用圆括号。

3.1.3 对象的定义

声明完类之后，就可以使用类来定义对象了，这个过程称为实例化。下面的语句：

```
Student zhang, wang;
```

定义了两个 Student 类对象 zhang 和 wang。zhang 和 wang 具有相同的属性和操作。

除了这种定义对象的形式之外，C++ 还兼容 C 语言的定义形式：

```
class Student zhang, wang;
```

这种形式与第一种形式是完全等效的。但显然第一种形式更加简便，所以将来遇到或使用最多的就是第一种形式。

除了这种先声明类，然后根据类名再定义对象的方式之外，还有在声明类的同时定义对象和定义无类名的类对象的形式。

1. 在声明类的同时定义对象

【例 3-2】 在声明例 3-1 的学生类的同时定义两个对象 zhang 和 wang。

```
class Student              //声明学生类 Student
{public:                   //以下部分为学生类 Student 的公用成员函数
    void setInfo()
    {    cout <<"Please enter student's No, Name, Sex, Age: ";
         cin >>stuNo >>stuName >>stuSex >>stuAge;
    }
    void show()
    {    cout <<"No. : " <<stuNo <<endl;
         cout <<"Name: " <<stuName <<endl;
         cout <<"Sex: " <<stuSex <<endl;
         cout <<"Age: " <<stuAge <<endl;
    }
private:                   //以下部分为学生类 Student 的私有数据成员
    string stuNo;          //学号
    string stuName;        //姓名
    string stuSex ="男"; //性别
    int stuAge =0;         //年龄
}zhang, wang;
```

其实这种形式的定义在 C 语言的结构体定义中也出现过，在实际的应用过程中并不太常用，只需要了解这种定义对象的形式即可。

2. 不出现类名，直接定义对象

【例 3-3】 直接定义无类名的类对象。

```
class
{public:
    void setInfo()
    {    cout <<"Please enter student's No, Name, Sex, Age: ";
         cin >>stuNo >>stuName >>stuSex >>stuAge;
    }
    void show()
    {    cout <<"No. : " <<stuNo <<endl;
         cout <<"Name: " <<stuName <<endl;
         cout <<"Sex: " <<stuSex <<endl;
         cout <<"Age: " <<stuAge <<endl;
    }
private:
    string stuNo;          //学号
    string stuName;        //姓名
```

```
    string stuSex = "男"; //性别
    int stuAge = 0;       //年龄
}zhang, wang;
```

这与例 3-2 的区别就是去掉了类名，这样 zhang 和 wang 两个对象是同一个类的对象，但是不知道它们所属类的名字。这种方式也不常用，只需知道有这种定义对象的形式就可以了。

特别提醒：

声明类时系统并不分配内存单元，而定义对象时系统会给每个对象分配内存单元，以存储对象的成员信息。

3.2 类的成员函数

类中有两类成员，一类是数据成员，用来描述对象的静态属性；另一类则是成员函数，用来描述对象的动态行为。下面讨论类的成员函数的相关问题。

3.2.1 成员函数的性质

类的成员函数是引入类和对象之后出现的新的函数，以前讲述的函数是不属于任何类的，可以称之为普通函数（或一般函数）。成员函数的定义和用法与普通函数的定义与用法基本一样，定义形式也是：

```
返回值类型 函数名(参数表)
{
    函数体
}
```

它与普通函数的区别在于：成员函数是属于某个类的，是类的一个成员。可以指定成员函数为公用的、受保护的或私有的。在类外调用成员函数时要注意它的访问属性，公用的成员函数才可以被类外任意的调用，私有成员函数在类外是看不到的。成员函数可以访问本类的任何成员，不管它的访问属性是公用的、私有的，还是受保护的，可以访问本类的数据成员，也可以调用本类的其他成员函数。

对于类的成员函数，一般的做法是将需要被类外调用的成员函数声明为公用的，不需要被类外调用的成员函数声明为私有的。有的成员函数并不是为外界调用而设计的，而是为本类中的其他成员函数所调用的，就应该把它们指定为私有的。

类中的公用的成员函数是类的很重要的部分，这些成员函数称为是类的对外接口。如果一个类的数据成员都是私有的，而类中又没有公用的成员函数，这种类是没有办法和外界进行交流的，这种类也是没有意义的。如果类的数据成员都是公用的，则类就失去了信息隐藏的功能，退变成类似结构体的功能，体现不出面向对象的思想。例 3-1 的 Student 类的 show 函数就是一个公用成员函数，是类的对外接口。

3.2.2　在类外定义成员函数

例 3-1 中的 Student 类的 show 函数是在类体中定义的。但在实际的应用中，一般在类的声明中只给出成员函数的原型声明，而成员函数的定义则在类外进行。

C++ 要求成员函数在类外定义时，在函数名的前面要加上类名和作用域运算符进行限定。

【例 3-4】 将例 3-1 中的 Student 类的成员函数改为在类外定义的形式。

```
class Student              //声明学生类 Student
{public:                   //以下部分为学生类 Student 的公用成员函数
    void setInfo();
    void show();
private:                   //以下部分为学生类 Student 的私有数据成员
    string stuNo;          //学号
    string stuName;        //姓名
    string stuSex ="男"; //性别
    int stuAge =0;         //年龄
};                         //类声明结束
void Student::setInfo()  //在类的声明之外定义学生类 Student 的 setInfo 成员函数
{    cout <<"Please enter student's No, Name, Sex, Age: ";
      cin >>stuNo >>stuName >>stuSex >>stuAge;
}
void Student::show()     //在类的声明之外定义学生类 Student 的 show 成员函数
{   cout <<"No. : " <<stuNo <<endl;
    cout <<"Name: " <<stuName <<endl;
    cout <<"Sex : " <<stuSex <<endl;
    cout <<"Age: " <<stuAge <<endl;
}
```

如果成员函数放在类内进行定义，这时是不需要在函数名前加上类名进行限定的，因为在类内的成员函数属于哪个类是很显然，不会出现歧义。

注意：

(1) 成员函数在类内定义或是在类外定义，对程序执行的效果基本一样。只是对于较长的成员函数放在类外更有利于程序的阅读。

(2) 在类外定义成员函数时，必须首先在类内写出成员函数的原型声明，然后再在类外定义，这就要求类的声明必须在成员函数定义之前。

(3) 如果在类外有函数定义，但是在函数名前没有类名和作用域运算符进行限定，则该函数被认为是普通函数。

(4) 如果成员函数的函数体很短，如只有两三行，也可以将其定义在类内。

(5) 在类内声明成员函数，在类外定义成员函数，是软件工程中要求的良好的编程风格，因为这样做不仅可以减少类体的长度，而且有利于把类的接口和类的实现细节分离。

3.2.3 inline成员函数

要理解什么是inline成员函数，先看看什么是inline函数。inline函数又称为内置函数或内联函数，该方法的思想是在编译时将被调用函数的代码直接嵌入到调用函数处。

由于inline函数的机制是在编译时将被调用函数的代码嵌入到主调用函数的调用语句处，所以程序员在读程序时仍可以感觉到函数带来的模块性、可读性及代码重用等函数的优点，而在编译之后，在目标程序中就不再存在被调用函数了，被调用函数的代码已经被插入到了调用语句处，提高了程序的执行效率。可以看到，inline函数的机制兼顾了函数和效率两个方面的优点。

同理，inline成员函数就是将类中的成员函数声明为内置的。在类中有不少成员函数的函数体都很小，只有几行代码，这时函数调用的时间开销是非常明显的。为了降低函数调用的时间开销，C++采用内置成员函数的机制，即在函数调用时并不是真正的函数调用，而是将函数的代码嵌入到程序中的函数调用处。

当类中的成员函数是在类内定义时，C++系统会默认该成员函数是inline成员函数，此时不必在函数定义前面加上inline关键字，如果写上inline也是可以的。如前面的例3-1中的show函数的定义就在Student类的内部，虽然show函数前面没有inline关键字，系统也会使用inline成员函数的机制去处理。

如果成员函数定义在类的外部，类内只有成员函数的声明，则在成员函数声明或成员函数定义前必须要有inline关键字。成员函数声明和成员函数定义这两处只要有一处声明为inline即可，都写上inline关键字也可以。但是这时要注意，如果在类外定义inline成员函数，则必须将类的声明和inline成员函数的定义写在同一个文件里，否则编译时无法进行inline成员函数代码的嵌入。这样的话就使得类的接口与类的实现无法分离，不利于信息隐藏。

友情提示：

inline成员函数同样要求成员函数体要简单且调用频繁，如果成员函数体中含有循环或复杂的嵌套的选择结构等，则不要声明为inline成员函数。

3.2.4 成员函数的存储方式

在实例化类得到对象时，系统要给对象分配存储空间，数据和函数都是需要存储空间的。一个类的不同对象其数据成员的值是不一样的，因此必须为每一个对象的数据成员分配存储空间，但是同一个类的不同对象的函数的代码却是相同的，这时再对每一个对象的函数代码分配内存空间，则会造成存储空间的浪费，是没有必要的。所以C++为类的对象分配内存空间时只为对象的数据成员分配内存空间，而将对象的成员函数放在另外一个公共的区域，无论这个类声明多少个对象，这些对象的成员函数在内存中只保存一份。

可以通过如下程序验证上述观点。

【例 3-5】 类的对象占用内存空间情况实验。

```
#include <iostream>
using namespace std;
class Test
{public:
    void show(){  cout <<"char in test is: " <<c <<endl;  }
private:
    char c;
};
int main()
{  Test test;
   cout <<"The size of test is " <<sizeof(test) <<endl;
   return 0;
}
```

程序运行结果如下：

```
The size of test is 1
```

程序中的 sizeof 是长度运算符，用来获得数据类型或变量的长度。可以看到程序运行后的输出结果是 1，这个长度刚好是 Test 类中字符型数据成员 c 所需空间的大小，这也就验证了对象所占空间大小取决于对象的数据成员，与成员函数无关。

既然同一个类的不同对象所用的成员函数是公共的，也就是执行的代码是相同的，为什么执行的结果会不同呢？

【例 3-6】 相同类的不同对象执行相同成员函数输出不同结果。

```
#include <iostream>
using namespace std;
class Test
{public:
    void set(char ch) {  c =ch;  }
    void show(){  cout <<"char in Test is: " <<c <<endl;  }
private:
    char c;
};
int main()
{  Test test1, test2;
   test1. set('a');
   test2. set('b');
   test1. show();
   test2. show();
   return 0;
}
```

程序运行结果如下：

```
char in Test is: a
char in Test is: b
```

从程序中可以看到，对象 test1 和 test2 都执行了 show 成员函数，两个对象执行的代码都是“cout <<"char in Test is：" <<c <<endl;”，为什么输出结果却是不同的呢？其实，两个对象执行的代码中访问的数据成员是不一样的，不同对象成员函数中访问的数据成员都是对象自己的，那么又怎么实现不同对象的相同的代码访问不同的对象的数据成员呢？C++ 利用隐藏的 this 指针，this 指针隐藏的指向调用该成员函数的对象的地址，而代码中的访问数据成员的变量名前面实际上是有 this 指针的，只不过没有显式的写出来，即代码行“cout <<"char in Test is：" <<c <<endl;”实际上等价于“cout <<"char in Test is：" <<this ->c <<endl;”。当用 test1 对象调用 show 函数时，this 指针就指向 test1 对象；当用 test2 对象调用 show 函数时，this 指针就指向 test2 对象。关于 this 指针在后面还会有详细的讨论。

3.3 对象成员的访问

如何在类外对对象中的公用成员进行访问？方法主要有 3 种：

(1) 通过对象名和成员运算符来访问对象的成员。

(2) 通过指向对象的指针来访问对象的成员。

(3) 通过对象的引用来访问对象的成员。

3.3.1 通过对象名和成员运算符来访问对象的成员

通过对象名和成员运算符访问对象中的成员的一般形式为：

```
对象名.成员名
```

其中的“.”是成员运算符，作用是对成员进行限定，指明成员是属于哪个对象的成员。因此，在使用对象的成员时一定要写清楚成员所属的对象，如果只写成员名，系统则会误认为是一个普通的变量或函数。例如，例 3-6 中 main 函数里的“test1.show();”语句就是访问 test1 对象的公用成员函数 show。

特别提醒：

通过对象名和成员运算符访问对象中的成员可以是公用的数据成员，也可以是公用的成员函数，无论是数据成员还是成员函数，要求其访问属性必须是公用的。

3.3.2 通过指向对象的指针来访问对象的成员

要通过指向对象的指针访问对象中的成员，可以使用 C++ 的“->”运算符方便直观地进行。“->”称为指向运算符，该运算符可以通过指向对象的指针访问对象的成员。

【例 3-7】 通过指向对象的指针访问对象中的成员。

```
#include <iostream>
using namespace std;
class Test
{public:
    void set(char ch) {  c =ch;  }
    void show(){  cout <<"char in Test is: " <<c <<endl;   }
private:
    char c;
};
int main()
{  Test test1;
   test1. set('a');
   Test * pTest =&test1;
   test1. show();
   pTest ->show();
   return 0;
}
```

程序运行结果如下：

```
char in Test is: a
char in Test is: a
```

从例 3-7 中可以看到“test1. show();”的效果和“pTest ->show()”的效果完全一样，pTest ->show()表示调用 pTest 指针当前所指向的对象(本例中是 test1 对象)中的成员函数 show。另外，除这种形式外，(* pTest). show()是另外一种通过指针访问其所指向对象成员的形式，因为(* pTest)就是指针所指向的对象，在本例中，(* pTest)就是 test1 对象，(* pTest). show()就等价于 test1. show()。总之，在 pTest 指向 test1 对象的情况下，pTest ->show()、(* pTest). show()和 test1. show()等 3 种书写形式是等价的。

3.3.3　通过对象的引用来访问对象的成员

对象的引用和普通变量的引用在本质上是一样的，对象的引用就是给对象起了一个别名，使用引用名和使用对象名访问的都是同一个对象。因此，通过引用访问对象的成员的方法与通过对象名来访问对象的成员是完全相同的。

【例 3-8】　通过对象的引用访问对象的成员。

```
#include <iostream>
using namespace std;
class Test
{public:
    void set(char ch) {  c =ch;  }
    void show() {   cout <<"char in Test is: " <<c <<endl;   }
private:
    char c;
```

```
};
int main()
{  Test test1;
   test1. set('a');
   Test &refTest =test1;
   test1. show();
   refTest. show();
   return 0;
}
```

程序运行结果如下：

```
char in Test is: a
char in Test is: a
```

3.4 构造函数与析构函数

3.4.1 构造函数

1. 构造函数的任务

构造函数是类的一个特殊成员函数，构造函数的任务是初始化类对象的数据成员。无论何时只要类的对象被创建，就会执行构造函数。看下面的例子：

【例 3-9】 构造函数举例。

```
#include<iostream>
using namespace std;
#include<string>
class SalesData          //声明图书交易记录类 SalesData
{public:
    SalesData()          //定义 SalesData 类的构造函数，构造函数名与类名相同
    {                    //利用构造函数对数据成员赋初值
        bookNo ="null";  unitsSold =0;  revenue =0. 0;
        cout <<"SalesData's constructor is executed!" <<endl;
    }
    void show()          //输出图书基本销售记录：书号、销售册数、总销售金额、销售均价
    {   cout <<"bookNo =" <<bookNo <<endl;
        cout <<"unitsSold =" <<unitsSold <<endl;
        cout <<"revenue =" <<revenue <<endl;
        if( unitsSold !=0 )   cout <<"avgPrice =" <<avgPrice() <<endl;
    }
    double avgPrice(){  return revenue / unitsSold;  }     //计算图书销售均价
private:
    string bookNo;       //书号
    int unitsSold ;      //销售出的册数
```

```
    double revenue;       //总销售金额
};
int main()
{   SalesData book1;      //创建 book1 对象
    cout <<"The information of book1 is: " <<endl;
    book1. show();
    SalesData book2;      //创建 book2 对象
    cout <<"The information of book2 is: " <<endl;
    book2. show();
    return 0;
}
```

程序运行结果如下：

```
SalesData's constructor is executed!
The information of book1 is:
bookNo =null
unitsSold =0
revenue =0
SalesData's constructor is executed!
The information of book2 is:
bookNo =null
unitsSold =0
revenue =0
```

由于在 main 函数中创建了 SalesData 类的两个对象 book1、book2，因此类 SalesData 的构造函数被调用了 2 次。

构造函数不仅具有成员函数的特性，而且还具有它自身的特点。对构造函数总结如下。

(1) 构造函数名与类名相同，与其他函数不一样的是，构造函数没有返回类型，在定义构造函数时在函数名前什么也不能加(加 void 也不可以)；除此之外类似于其他的函数，构造函数也有一个(可能为空的)参数列表和一个(可能为空的)函数体。

(2) 构造函数可以被重载，一个类可以包含多个构造函数，不同的构造函数之间在形参数量或形参类型上有所不同。稍后介绍。

(3) 构造函数不需要用户调用，它是由系统在创建对象时自动调用的。鉴于此，构造函数要声明为 public 访问属性的。

(4) 构造函数不能声明为 const 的。当我们创建一个 const 对象(3.7.1 节介绍)时，直到构造函数完成初始化过程，对象才能真正取得其“常量”属性。因此，构造函数在 const 对象的构造过程中可以向其写值。

(5) 构造函数的作用是在创建对象时对对象的数据成员进行初始化，一般在构造函数的函数体里写对其数据成员初始化的语句，但是也可以在其中加上和初始化无关的其他语句。这虽然在语法上没有错误，但是在实际编程中不提倡这样做。例 3-9 中的构造函数的函数体中的输出语句只是为了演示构造函数的执行时机，除此之外，没有其他任何

用途。

（6）C++系统在创建对象时必须执行一个构造函数。你可能会有这样的疑问：前面的例3-5～例3-8中的Test类并没有定义任何构造函数，可使用了Test类对象的程序仍然可以正常的编译和运行？这是因为，如果用户自己没有定义构造函数，编译器会隐式地提供一个构造函数，称之为合成的默认构造函数，该构造函数的形参表和函数体皆为空，它按如下规则初始化类的数据成员：

① 如果存在类内的初始值，用它来初始化成员，这是C++11标准中新增的类内初始化。

② 否则，默认初始化该成员，其初始化规则与普通变量的默认初始化规则[①]相同。

合成的默认构造函数只适合非常简单的类，对于大多数普通的类来说，必须定义它自己的默认构造函数（该构造函数或者形参表为空，如例3-9中的构造函数，或者形参不为空但全部参数都有默认实参），原因有三：

第一个原因也是最容易理解的一个原因就是编译器只有在发现类不包含任何构造函数的情况下才会替我们生成一个默认的构造函数。一旦我们定义了一些其他的构造函数，那么除非我们再定义一个默认的构造函数，否则该类将没有默认构造函数。

第二个原因是对于某些类来说，合成的默认构造函数可能执行错误的操作。对于含有内置类型或复合类型成员的类，或者在类的内部初始化这些成员，或者定义一个自己的默认构造函数。否则，用户在创建类的对象时就可能得到未定义的值。

第三个原因是有的编译器不能为某些类合成默认的构造函数。例如，如果类中包含一个其他类类型的成员且这个成员的类型没有默认构造函数，那么编译器将无法初始化该成员。对于这样的类来说，我们必须自定义默认构造函数，否则该类将没有默认的构造函数可用。

2. 定义自己的构造函数

对于例3-9中的SalesData类，自定义不同的构造函数，给类实例化对象时提供不同的初始化方法。

【例3-10】 构造函数重载。

```
#include<iostream>
using namespace std;
#include<string>
class SalesData                                    //声明图书交易记录类 SalesData
{public:
    SalesData() =default;                          //无参的默认构造函数
```

① 如果定义变量时没有指定初值，则变量被默认初始化，此时变量被赋予一个“默认值”。默认值到底是什么由变量的类型决定，同时定义变量的位置也会对此有影响。如果是内置类型的变量未被显示初始化，它的值由定义的位置决定。定义于任何函数体之外的全局变量均被初始化为0，而定义于函数体（块）内部的局部变量的值随机。定义在块内部的局部变量有一个例外，即如果是static型变量，即使不显式地给予初始值，也会被默认初始化为零。

未初始化变量的默认值不一定符合我们程序的实际需求，使用未初始化变量的值是一种错误的编程行为并且很难调试。尽管大多数编译器都能对一部分使用未初始化变量的行为提出警告，但严格来说，编译器并未被要求检查此类错误。

```
    SalesData(const string &No): bookNo(No){}  //带1个const string &参数的构造函数
    SalesData(istream &input);                  //带1个istream &参数的构造函数
    //带3个参数(书号、销售册数、销售价格)的构造函数
    SalesData(const string &No, int n, double price): bookNo(No), unitsSold(n),
              revenue(n*price){ }
    void show()              //输出图书基本销售记录:书号、销售册数、总销售金额、销售均价
    {     cout <<"bookNo =" <<bookNo <<endl;
          cout <<"unitsSold =" <<unitsSold <<endl;
          cout <<"revenue =" <<revenue <<endl;
          if( unitsSold !=0 )   cout <<"avgPrice =" <<avgPrice() <<endl;
    }
    double avgPrice(){  return revenue / unitsSold;  }     //计算图书销售均价
    string isbn() {  return bookNo;  };        //声明获取书号函数,并返回书号
    //一个combine成员函数,用于将一个SalesData对象加到另一个对象上
    SalesData& combine(const SalesData &);
private:
    string bookNo ="null";                     //书号
    int unitsSold =0;                          //销售出的册数
    double revenue =0. 0;                      //总销售金额
};
int main()
{   SalesData book1;
    book1. show(); cout <<endl;
    SalesData book2("1002");
    book2. show(); cout <<endl;
    SalesData book3("1003",2,35. 0) ;
    book3. show(); cout <<endl;
    cout<<"请输入同一本图书的销售记录:书号、销售册数、销售单价" <<endl;
    SalesData book4(cin), book5(cin);
    if ( book4. isbn() ==book5. isbn() )
    {   SalesData total =book4. combine(book5);
        cout <<"此本图书的销售记录:书号、销售册数、总销售金额 销售均价" <<endl;
        total. show(); cout <<endl;
    }
    else
    {   cout <<"您输入的两本书编号不同,";
        cout <<"它们的销售记录:书号、销售册数、总销售金额、销售均价分别为:" <<endl;
        book4. show(); cout <<endl;
        book5. show(); cout <<endl;
    }
    return 0;
}
SalesData::SalesData(istream &input)           //构造函数的类外定义
{  double price;
```

```
    input >>bookNo >>unitsSold >>price;
    revenue =unitsSold * price;
}
SalesData& SalesData::combine(const SalesData &rhs) // combine 成员函数的类外定义
{   //把 rhs 对象的数据成员 unitsSold 的值累加到当前对象的数据成员 unitsSold
    unitsSold +=rhs. unitsSold;
    //把 rhs 对象的数据成员 revenue 的值累加到当前对象的数据成员 revenue
    revenue +=rhs. revenue;
    return * this;                                  //返回当前对象的引用
}
```

例 3-10 的 SalesData 类提供有如下 4 个不同的构造函数：

(1) SalesData() = default。

该构造函数是一个无参的默认构造函数。定义这个构造函数的目的仅仅是因为我们既需要其他形式的构造函数，也需要默认的构造函数。我们希望这个构造函数的作用完全等同于系统提供的合成默认构造函数。

在 C++ 11 标准中，如果我们需要合成默认构造函数的行为，那么可以通过在参数列表后面写上=default 来要求编译器生成构造函数。

(2) SalesData(const string &No): bookNo(No){ }。

该构造函数为带一个 const string & 形参的构造函数，形参 No 表示书号。该构造函数使用初始化列表[①]对数据成员 bookNo 初始化，初始值为形参 No 的值。对于数据成员 unitsSold 和 revenue 则没有显式初始化。当一个数据成员被构造函数初始化列表忽略时，它将以与合成默认构造函数相同的方式隐式初始化。在此例中，这样的成员使用类内初始值初始化。该构造函数的唯一目的就是为数据成员 bookNo 赋初值，一旦没有其他任务需要执行，函数体也就为空了。

(3) SalesData(istream &input)。

该构造函数也为带一个 istream & 形参的构造函数，形参 input 表示输入流对象，与上面带一个 const string & 形参的构造函数的形参类型不同。向该构造函数传入 cin 对象，可以实现从键盘输入新创建对象的数据成员 bookNo 和 unitsSold 的初始值，并根据从键盘输入的销售单价 price 计算数据成员 revenue 的值(unitsSold * price)。

(4) SalesData(const string &No, int n, double price): bookNo(No), unitsSold(n), revenue(n * price){ }。

该构造函数为带 3 个参数的构造函数，第 1 个形参 No 表示书号，第 2 个形参 n 表示图书销量册数，第 3 个形参 price 表示图书销售价格，它同样采用构造函数的初始化列表对数据成员初始化，使用形参 No 的值初始化数据成员 bookNo，使用形参 n 的值初始化数据成员 unitsSold，使用形参 n 和 price 的乘积初始化数据成员 revenue。该构造函数的

① 构造函数的初始化列表为函数参数表后面的冒号及冒号和花括号之间的代码，它负责为新创建对象的一个或多个数据成员赋初值。它是成员名字的一个列表，每个名字后面紧跟括号括起来的(或者在花括号内的)成员初始值。不同成员的初始化通过逗号分隔开来。

函数体也为空。

带参数的构造函数声明的一般格式为：

```
构造函数名(参数表);
```

这里的参数表和普通函数的参数表是一样的，由参数类型和形参名组成，多个形参之间通过“,”分隔。

实参是在定义对象时给出的，一般格式如下：

```
类名 对象名(实参表);
```

这里的实参表的参数的类型和个数要和形参表中的对应起来。

在例 3-10 程序中，创建了 5 个 SalesData 类的对象 book1、book2、book3、book4、book5。创建 book1 对象时未提供实参，系统会自动调用无参的默认构造函数初始化 book1 数据成员的值(bookNo 的值为 null，unitsSold 的值为 0，revenue 的值为 0.0)；创建 book2 对象时给出了一个字符串实参，系统会自动调用带有一个字符串形参的构造函数初始化 book2 数据成员的值(bookNo 的值为 1002，unitsSold 的值为 0，revenue 的值为 0.0)；创建 book3 对象时给出了 3 个实参，系统会自动调用带有 3 个形参的构造函数初始化 book3 数据成员的值(bookNo 的值为 1003，unitsSold 的值为 2，revenue 的值为 70(2 * 35.0))；创建 book4 和 book5 对象时给出了一个输入流对象 cin 实参，系统会自动调用带有一个 istream& 形参的构造函数，完成从键盘输入这两个对象的数据成员 bookNo 和 unitsSold 的初始值，并根据从键盘输入的销售单价 price 计算这两个对象的数据成员 revenue 的值(unitsSold * price)。如果两次输入的书号相同，程序实现同一本图书两次交易记录的累加(销售册数累加、总销售金额累加)。

带参数的构造函数形式可以很方便地实现创建对象时，对不同对象进行不同的初始化。

注意：

(1) 前面曾提到：如果在类的声明中没有书写构造函数，系统会自动生成一个无参的、函数体为空的合成默认构造函数。一旦书写了一个构造函数，系统就不会再生成合成默认构造函数，但是，C++ 11 标准通过引入=default，可以要求编译器为用户生成合成默认构造函数。对于例 3-10 中的 SalesData 类默认构造函数的处理：由于在类内提供有数据成员的类内初始值，使用=default 要求编译器为我们生成合成默认构造函数是合适的。如果在类内未提供数据成员的类内初始值，就需要用户自己写一个无参的默认构造函数。编译器自动生成的合成默认构造函数，自定义的无参或者有参但全部参数都有默认值的构造函数，皆称为默认构造函数。请记住一个类只能有一个默认构造函数，否则构造函数重载会出现二义性。

(2) 在程序中定义对象时一定要注意类中有几个构造函数，它们要求的参数分别是什么样的，如果创建对象时给出的参数表和所有的构造函数的参数表都不匹配，则系统无法创建对象。在设计类时，应尽可能考虑将来创建对象的各种情况，写出多个构造函数。虽然构造函数有多个，但是在创建一个对象时，系统只调用其中的一个。

(3) 使用参数实例化对象时的格式是“类名 对象名(实参表);”，而使用默认构造函

数实例化对象时的格式是“类名 对象名;”，这时若在对象名后加括号则是错误的。如例 3-10 中对 book1 对象的定义时，如果写成“Box book1();”，则是错误的。

3. 构造函数的初始化列表

构造函数的执行分为两个阶段：

(1) 初始化阶段(初始化)。

初始化阶段具体指的是用构造函数初始化列表方式来初始化类中的数据成员。没有在构造函数的初始值列表中显式地初始化的成员，则该成员将在构造函数体之前执行默认初始化。

(2) 普通计算阶段(赋值)。

执行构造函数的函数体，函数体内一般是对类的数据成员进行赋值的操作。

这两个阶段按照顺序依次执行。

如果把例 3-10 中的构造函数：

```
SalesData(const string &No, int n, double price) : bookNo(No), unitsSold(n),
revenue(n*price){ }
```

改为如下的书写形式：

```
SalesData(const string &No, int n, double price)
{  bookNo=No;  unitsSold=n;  revenue=n*price;  }
```

这两个构造函数执行完成后的效果是一样的，数据成员的值相同。区别是上面的代码初始化了它的数据成员，而下面这个版本是对数据成员执行了赋值操作。这一区别到底会有什么深层次的影响完全依赖于数据成员的类型。

对于内置类型，如 int 型、float 型等，使用初始化列表和在构造函数体内赋值差别不是很大，但是对于类类型来说，最好使用初始化列表。因为使用初始化列表少了一次调用默认构造函数的过程，这对于数据密集型的类来说，是非常高效的。所以，一个好的原则是，能使用初始化列表的时候尽量使用初始化列表。

• 初始化列表有时必不可少

除了性能问题之外，有些时候初始化列表是不可或缺的，以下几种情况时必须使用初始化列表：

(1) 常数据成员，因为常量只能初始化不能赋值，所以必须放在初始化列表里面。

(2) 引用类型成员，引用必须在定义的时候初始化，并且不能重新赋值，所以也要写在初始化列表里面。

(3) 没有默认构造函数的类类型成员，因为使用初始化列表可以不必调用默认构造函数来初始化，而是直接调用复制构造函数初始化(复制构造函数的内容在 3.9.1 节介绍)。

• 成员初始化的顺序

初始化列表中成员初始化按照变量定义的先后顺序来初始化，与初始化列表中成员顺序无关。

如果成员初始化依赖其他成员的值，那么要注意初始化顺序。为了避免这个问题，一

般按照定义的顺序来初始化成员。

4. 委托构造函数

委托构造函数是C++11中对C++的构造函数的一项改进，其目的是为了简化构造函数的书写，减少冗余代码。一个委托构造函数可以使用它所属类的其他构造函数执行它自己的初始化过程，或者说它把它自己的一些(或全部)职责委托给了其他构造函数。

下面使用委托构造函数重写例3-10中的SalesData类的构造函数。

```
class SalesData          //声明图书交易记录类 SalesData
{public:
    //受委托的构造函数，也称为目标构造函数
    SalesData(const string &No, int n, double price): bookNo(No), unitsSold(n),
              revenue(n * price){ }
    //其余构造函数全部委托给另一个构造函数
    SalesData(): SalesData("null", 0, 0){ }
    SalesData(const string &No): SalesData(No, 0, 0){ }
    SalesData(istream &input) : SalesData()
    {  double price;
       input >>bookNo >>unitsSold >>price;
       revenue =  unitsSold * price;
    }
    …(此处省略的代码同例 3-10)
private:
    string bookNo;      //书号
    int unitsSold;      //销售出的册数
    double revenue;     //总销售金额
};
```

在这个SalesData类中，除了第一个构造函数外，其他的构造函数都委托了它们的工作，第一个构造函数接受3个实参，使用这些实参初始化数据成员，然后结束工作(函数体为空，所以没有执行动作)。自定义默认构造函数委托接受3个实参的构造函数完成初始化工作，它也无须执行其他任务，故函数体为空。接受一个字符串实参的构造函数同样委托给了使用三参数的构造函数版本，它同样无须执行其他任务，函数体也为空。

接受istream&的构造函数也是委托构造函数，它委托给了默认构造函数，默认构造函数又接着委托给三参数的构造函数。当这些受委托的构造函数执行完后，接着执行istream&构造函数体内的内容。

注意：

(1) 委托构造函数的初始化列表中有且仅有一个对目标构造函数的调用。即，如果委托构造函数的初始化列表中有一个对目标构造函数的调用，则该初始化列表中就不能再有其他东西(即不允许再有其他基类或数据成员的初始化)。

(2) 目标构造函数还可以再委托给另一个构造函数，但不要形成委托环。如，构造函数C1委托给另一个构造函数C2，而C2又委托给C1，这样的代码通常会导致编译错误。

(3) 委托构造函数的函数体中的语句在目标构造函数完全执行后才被执行。

（4）对象的生命期从任意一个构造函数执行完毕开始（对于委托构造的情况，就是最终的目标构造函数执行完毕时），这意味着从委托构造函数体中抛出异常将导致析构函数的自动执行。

3.4.2 析构函数

析构函数是与构造函数相对应的另一个特殊成员函数，其作用与构造函数正好相反。构造函数初始化对象的非 static 数据成员，而析构函数释放对象使用的额外内存资源，并析构对象的非 static 数据成员。

析构函数的函数名是固定的，由波浪线～后跟类名构成。析构函数没有返回值，也不接受参数。由于析构函数不接受参数，因此析构函数无法重载。一个类可以有多个构造函数，但有且仅有一个析构函数。当一个类未定义自己的析构函数时，编译器会为它自动生成一个合成的析构函数。合成析构函数的函数体为空。

1. 析构函数完成什么工作

如同构造函数有一个初始化部分和一个函数体，析构函数也有一个函数体和一个析构部分。在一个构造函数中，成员的初始化是在函数体执行之前完成的，且按照它们在类中出现的顺序进行初始化。在一个析构函数中，首先执行函数体，然后析构成员。成员按初始化顺序的逆序析构。

在析构函数的函数体内，可以书写在对象最后一次使用之后类设计者希望执行的任何收尾工作。通常，析构函数的函数体完成释放对象在生存期分配的所有资源。因为，析构函数的任务不是销毁对象，销毁对象是由系统来进行的，而是在系统销毁对象之前进行一些清理工作，把对象所占用的额外内存空间归还给系统。

析构部分是隐式的。成员析构时发生什么完全依赖于成员的类型。析构类类型的成员系统会自动执行成员对象自己的析构函数。内置类型没有析构函数，因此析构内置类型成员什么也不需要做。注意：隐式析构一个内置指针类型的成员不会 delete 它所指向的对象。

认识到析构函数体自身并不直接析构成员是非常重要的。成员是在析构函数体之后隐含的析构阶段中被析构的。在整个对象被析构过程中，析构函数体是作为成员析构步骤之外的另一部分进行的。

2. 什么时候会调用析构函数

无论何时一个对象被销毁，都会自动调用其析构函数。

（1）对象在离开其作用域时被销毁。

（2）当一个对象被销毁时，其成员被销毁。

（3）容器（无论是标准库容器还是数组）被销毁时，其元素被销毁。

（4）对于动态分配的对象，当对指向它的指针应用 delete 运算符时被销毁。

（5）对于临时对象，当创建它的完整表达式结束时被销毁。

注意：当指向一个对象的引用或指针离开作用域时，析构函数不会执行。

【例 3-11】 简化版的 SalesData 类。

```
#include<iostream>
```

```
using namespace std;
#include<string>
class SalesData          //声明图书交易记录类 SalesData
{public:
    //受委托的构造函数,也称为目标构造函数
    SalesData(const string &No, int n, double price): bookNo(No), unitsSold(n),
              revenue(n*price){ }
    //其余构造函数全部委托给另一个构造函数
    SalesData(): SalesData("null", 0, 0){ }
    SalesData(const string &No): SalesData(No, 0, 0){ }
    SalesData(istream &input) : SalesData()
    {   double price;
        input >>bookNo >>unitsSold >>price;
        revenue =unitsSold * price;
    }
    ~SalesData(){
        //对象被销毁前输出图书基本销售记录:书号、销售册数、总销售金额、销售均价
        cout <<"bookNo =" <<bookNo;
        cout <<", unitsSold =" <<unitsSold;
        cout <<", revenue =" <<revenue;
        if( unitsSold !=0 )
            cout <<", avgPrice =" <<avgPrice();
        cout<<endl;    //换行
    }
    double avgPrice(){  return revenue / unitsSold;  }     //计算图书销售均价
private:
    string bookNo;                                          //书号
    int unitsSold;                                          //销售出的册数
    double revenue;                                         //总销售额
};
int main()
{   SalesData book1("1001", 2,35.0);
    SalesData book2("1002", 5,32.6) ;
    return 0;
}
```

程序运行结果如下：

```
bookNo =1002, unitsSold =5, revenue =163, avgPrice =32. 6
bookNo =1001, unitsSold =2, revenue =70, avgPrice =35
```

限于篇幅，例 3-11 中对 SalesData 类的功能进行了简化，只提供了构造函数、析构函数和计算图书销售均价的函数，并且析构函数的函数体很简单，只是简单输出被销毁对象的相关信息。在 main 函数里创建了两个对象 book1 和 book2，当 main 函数执行完毕时，系统会自动销毁这两个对象，在销毁它们之前系统自动调用这两个对象的析构函数，程序