第9章 关于类和对象的进一步讨论

9.1 构造函数

9.1.1 对象的初始化

在建立一个对象时，常常需要作某些初始化的工作，例如对数据成员赋初值。如果一个数据成员未被赋值，则它的值是不可预知的，因为在系统为它分配内存时，保留了这些存储单元的原状，这就成为了这些数据成员的初始值。这种状况显然是与人们的要求不相符的，对象是一个实体，它反映了客观事物的属性(例如时钟的时、分、秒的值)，是应该有确定的值的。

注意： 类的数据成员是不能在声明类时初始化的。

如果一个类中所有的成员都是公用的，则可以在定义对象时对数据成员进行初始化。如

```
class Time
{public:                    //声明为公用成员
 hour;
 minute;
 sec;
};
Time t1={14,56,30};          //将t1初始化为14:56:30
```

这种情况和结构体变量的初始化是差不多的，在一个花括号内顺序列出各公用数据成员的值，两个值之间用逗号分隔。但是，如果数据成员是私有的，或者类中有**private**或**protected**的成员，就不能用这种方法初始化。

在第**8**章的几个例子中，是用成员函数来对对象中的数据成员赋初值的(例如例**8.3**中的**set_time**函数)。从例**8.3**中可以看到，用户在主函数中调用**set_time**函数来为数据成员赋值。如果对一个类定义了多个对象，而且类中的数据成员比较多，那么，程序就显得非常臃肿烦琐。

9.1.2 构造函数的作用

为了解决这个问题，C++提供了构造函数(constructor)来处理对象的初始化。构造函数是一种特殊的成员函数，与其他成员函数不同，不需要用户来调用它，而是在建立对象时自动执行。构造函数的名字必须与类名同名，而不能由用户任意命名，以便编译系统能识别它并把它作为构造函数处理。它不具有任何类型，不返回任何值。构造函数的功能是由用户定义的，用户根据初始化的要求设计函数体和函数参数。

例9.1 在例8.3基础上定义构造成员函数。

```
#include <iostream>
using namespace std;
class Time
{public:
Time( )                  //定义构造成员函数，函数名与类名相同
{hour=0;                 //利用构造函数对对象中的数据成员赋初值
minute=0;
sec=0;
}
void set_time( );          //函数声明
void show_time( );         //函数声明
private:
int hour;                //私有数据成员
int minute;
int sec;
};
```

```
void Time::set_time( )      //定义成员函数，向数据成员赋值
{cin>>hour;
cin>>minute;
cin>>sec;
}
void Time::show_time( )      //定义成员函数，输出数据成员的值
{
 cout<<hour<<":"<<minute<<":"<<sec<<endl;
}
int main( )
{
Time t1;                //建立对象t1，同时调用构造函数t1.Time( )
t1.set_time( );          //对t1的数据成员赋值
t1.show_time( );         //显示t1的数据成员的值
Time t2;                //建立对象t2，同时调用构造函数t2.Time( )
t2.show_time( );         //显示t2的数据成员的值
return 0;
}
```

程序运行的情况为：

10 25 54↙ (从键盘输入新值赋给**t1**的数据成员)

10:25:54 (输出**t1**的时、分、秒值)

0:0:0 (输出**t2**的时、分、秒值)

上面是在类内定义构造函数的，也可以只在类内对构造函数进行声明而在类外定义构造函数。将程序中的第**4~7**行改为下面一行：

```
Time( );          //对构造函数进行声明
```

在类外定义构造函数：

```
Time::Time( )          //在类外定义构造成员函数，要加上类名Time和域限定符“::”
{hour=0;
minute=0;
sec=0;
}
```

有关构造函数的使用，有以下说明：

(1) 在类对象进入其作用域时调用构造函数。

(2) 构造函数没有返回值，因此也不需要在定义构造函数时声明类型，这是它和一般函数的一个重要的不同之点。

(3) 构造函数不需用户调用，也不能被用户调用。

(4) 在构造函数的函数体中不仅可以对数据成员赋初值，而且可以包含其他语句。但是一般不提倡在构造函数中加入与初始化无关的内容，以保持程序的清晰。

(5) 如果用户自己没有定义构造函数，则**C++**系统会自动生成一个构造函数，只是这个构造函数的函数体是空的，也没有参数，不执行初始化操作。

9.1.3 带参数的构造函数

在例**9.1**中构造函数不带参数，在函数体中对数据成员赋初值。这种方式使该类的每一个对象都得到同一组初值(例如例**9.1**中各数据成员的初值均为**0**)。但是有时用户希望对不同的对象赋予不同的初值。

可以采用带参数的构造函数，在调用不同对象的构造函数时，从外面将不同的数据传递给构造函数，以实现不同的初始化。构造函数首部的一般格式为

构造函数名(类型 **1** 形参**1**，类型**2** 形参**2**，…)

前面已说明： 用户是不能调用构造函数的，因此无法采用常规的调用函数的方法给出实参。实参是在定义对象时给出的。定义对象的一般格式为

类名 对象名(实参**1**，实参**2**，…);

例9.2 有两个长方柱，其长、宽、高分别为：(1)12,20,25；(2)10,14,20。求它们的体积。编一个基于对象的程序，在类中用带参数的构造函数。

```
#include <iostream>
using namespace std;
class Box
{public:
Box(int,int,int);        //声明带参数的构造函数
int volume( );           //声明计算体积的函数
 private:
int height;
int width;
int length;
};
Box::Box(int h,int w,int len)   //在类外定义带参数的构造函数
{height=h;
width=w;
length=len;
}
```

```
int Box::volume( )              //定义计算体积的函数
{return(height*width*length);
}

int main( )
{Box box1(12,25,30);          //建立对象box1，并指定box1长、宽、高的值
cout<<"The volume of box1 is "<<box1.volume( )<<endl;
Box box2(15,30,21);          //建立对象box2，并指定box2长、宽、高的值
cout<<"The volume of box2 is "<<box2.volume( )<<endl;
return 0;
}
```

程序运行结果如下：

```
The volume of box1 is 9000
The volume of box2 is 9450
```

可以知道：

（1） 带参数的构造函数中的形参，其对应的实参在定义对象时给定。

（2） 用这种方法可以方便地实现对不同的对象进行不同的初始化。

9.1.4 用参数初始化表对数据成员初始化

在**9.1.3**节中介绍的是在构造函数的函数体内通过赋值语句对数据成员实现初始化。**C++**还提供另一种初始化数据成员的方法——参数初始化表来实现对数据成员的初始化。这种方法不在函数体内对数据成员初始化，而是在函数首部实现。例如例**9.2**中定义构造函数可以改用以下形式：

```
Box::Box(int h,int w,int len):height(h)，width(w)，length(len){ }
```

这种写法方便、简练，尤其当需要初始化的数据成员较多时更显其优越性。甚至可以直接在类体中(而不是在类外)定义构造函数。

9.1.5 构造函数的重载

在一个类中可以定义多个构造函数，以便对类对象提供不同的初始化的方法，供用户选用。这些构造函数具有相同的名字，而参数的个数或参数的类型不相同。这称为构造函数的重载。在第4章第4.6节中所介绍的函数重载的知识也适用于构造函数。

通过下面的例子可以了解怎样应用构造函数的重载。

例**9.3** 在例**9.2**的基础上，定义两个构造函数，其中一个无参数，一个有参数。

```
#include <iostream>
using namespace std;
class Box
{public:
Box( );                                  //声明一个无参的构造函数
Box(int h,int w,int len):height(h),width(w),length(len){ }
//声明一个有参的构造函数，用参数的初始化表对数据成员初始化
int volume( );
private:
int height;
int width;
int length;
};
Box∷Box( )                               //定义一个无参的构造函数
{height=10;
width=10;
length=10;
}
```

```
int Box::volume( )
{return(height*width*length);
}

int main( )
{
Box box1;                              //建立对象box1,不指定实参
cout<<"The volume of box1 is "<<box1.volume( )<<endl;
Box box2(15,30,25);                    //建立对象box2,指定3个实参
cout<<"The volume of box2 is "<<box2.volume( )<<endl;
return 0;
}
```

在本程序中定义了两个重载的构造函数，其实还可以定义其他重载构造函数，其原型声明可以为

```
Box::Box(int h);                       //有1个参数的构造函数
Box::Box(int h,int w);                 //有两个参数的构造函数
```

在建立对象时分别给定1个参数和2个参数。

说明：

(1) 调用构造函数时不必给出实参的构造函数，称为默认构造函数**(default constructor)**。显然，无参的构造函数属于默认构造函数。一个类只能有一个默认构造函数。

(2) 如果在建立对象时选用的是无参构造函数，应注意正确书写定义对象的语句。

(3) 尽管在一个类中可以包含多个构造函数，但是对于每一个对象来说，建立对象时只执行其中一个构造函数，并非每个构造函数都被执行。

9.1.6 使用默认参数的构造函数

构造函数中参数的值既可以通过实参传递，也可以指定为某些默认值，即如果用户不指定实参值，编译系统就使形参取默认值。

在第**4**章第**4.8**节中介绍过在函数中可以使用有默认值的参数。在构造函数中也可以采用这样的方法来实现初始化。

例**9.3**的问题也可以使用包含默认参数的构造函数来处理。

例**9.4** 将例**9.3**程序中的构造函数改用含默认值的参数，长、宽、高的默认值均为**10**。

在例**9.3**程序的基础上改写如下：

```
#include <iostream>
using namespace std;
class Box
{public:
Box(int h=10,int w=10,int len=10);       //在声明构造函数时指定默认参数
int volume( );
 private:
int height;
int width;
int length;
};
Box::Box(int h,int w,int len)       //在定义函数时可以不指定默认参数
{height=h;
width=w;
length=len;
}
```

```
int Box::volume( )
{return(height*width*length);
}

int main( )
{
Box box1;                    //没有给实参
cout<<"The volume of box1 is "<<box1.volume( )<<endl;
Box box2(15);                //只给定一个实参
cout<<"The volume of box2 is "<<box2.volume( )<<endl;
Box box3(15,30);             //只给定2个实参
cout<<"The volume of box3 is "<<box3.volume( )<<endl;
Box box4(15,30,20);          //给定3个实参
cout<<"The volume of box4 is "<<box4.volume( )<<endl;
return 0;
}
```

程序运行结果为

The volume of box1 is 1000

The volume of box2 is 1500

The volume of box3 is 4500

The volume of box4 is 9000

程序中对构造函数的定义(第**12~16**行)也可以改写成参数初始化表的形式：

```
Box::Box(int h,int w,int len):height(h),width(w),length(len){ }
```

可以看到： 在构造函数中使用默认参数是方便而有效的，它提供了建立对象时的多种选择，它的作用相当于好几个重载的构造函数。它的好处是： 即使在调用构造函数时没有提供实参值，不仅不会出错，而且还确保按照默认的参数值对对象进行初始化。尤其在希望对每一个对象都有同样的初始化状况时用这种方法更为方便。

说明：

(1) 应该在声明构造函数时指定默认值，而不能只在定义构造函数时指定默认值。

(2) 程序第**5**行在声明构造函数时，形参名可以省略。

(3) 如果构造函数的全部参数都指定了默认值，则在定义对象时可以给一个或几个实参，也可以不给出实参。

(4) 在一个类中定义了全部是默认参数的构造函数后，不能再定义重载构造函数。

9.2 析构函数

析构函数(**destructor**)也是一个特殊的成员函数，它的作用与构造函数相反，它的名字是类名的前面加一个“～”符号。在**C++**中“～”是位取反运算符，从这点也可以想到： 析构函数是与构造函数作用相反的函数。

当对象的生命期结束时，会自动执行析构函数。具体地说如果出现以下几种情况，程序就会执行析构函数： ①如果在一个函数中定义了一个对象(它是自动局部对象)，当这个函数被调用结束时，对象应该释放，在对象释放前自动执行析构函数。

②**static**局部对象在函数调用结束时对象并不释放，因此也不调用析构函数，只在**main**函数结束或调用**exit**函数结束程序时，才调用**static**局部对象的析构函数。③如果定义了一个全局对象，则在程序的流程离开其作用域时(如**main**函数结束或调用**exit**函数) 时，调用该全局对象的析构函数。④如果用**new**运算符动态地建立了一个对象，当用**delete**运算符释放该对象时，先调用该对象的析构函数。

析构函数的作用并不是删除对象，而是在撤销对象占用的内存之前完成一些清理工作，使这部分内存可以被程序分配给新对象使用。程序设计者事先设计好析构函数，以完成所需的功能，只要对象的生命期结束，程序就自动执行析构函数来完成这些工作。

析构函数不返回任何值，没有函数类型，也没有函数参数。因此它不能被重载。一个类可以有多个构造函数，但只能有一个析构函数。

实际上，析构函数的作用并不仅限于释放资源方面，它还可以被用来执行“用户希望在最后一次使用对象之后所执行的任何操作”，例如输出有关的信息。这里说的用户是指类的设计者，因为，析构函数是在声明类的时候定义的。也就是说，析构函数可以完成类的设计者所指定的任何操作。

一般情况下，类的设计者应当在声明类的同时定义析构函数，以指定如何完成“清理”的工作。如果用户没有定义析构函数，**C++**编译系统会自动生成一个析构函数，但它只是徒有析构函数的名称和形式，实际上什么操作都不进行。想让析构函数完成任何工作，都必须在定义的析构函数中指定。

例9.5 包含构造函数和析构函数的C++程序。

```
#include<string>
#include<iostream>
using namespace std;
class Student                          //声明Student类
{public:
student(int n,string nam,char s )          //定义构造函数
{num=n;
name=nam;
sex=s;
cout<<"Constructor called."<<endl;      //输出有关信息
}
~Student( )                            //定义析构函数
{cout<<"Destructor called."<<endl;}      //输出有关信息
void display( )                        //定义成员函数
{cout<<"num: "<<num<<endl;
cout<<"name: "<<name<<endl;
cout<<"sex: "<<sex<<endl<<endl; }
```

```
private:
int num;
char name[10];
char sex;
};

int main( )
{Student stud1(10010,"Wang_li",'f');        //建立对象stud1
stud1.display( );                          //输出学生1的数据
Student stud2(10011,"Zhang_fun",'m');      //定义对象stud2
stud2.display( );                          //输出学生2的数据
return 0;
}
```

程序运行结果如下：

```
Constructor called.          (执行stud1的构造函数)
num: 10010                   (执行stud1的display函数)
name:Wang_li
sex: f

Constructor called.          (执行stud2的构造函数)
num: 10011                   (执行stud2的display函数)
name:Zhang_fun
sex:m
Destructor called.           (执行stud2的析构函数)
Destructor called.           (执行stud1的析构函数)
```

9.3 调用构造函数和析构函数的顺序

在使用构造函数和析构函数时，需要特别注意对它们的调用时间和调用顺序。

在一般情况下，调用析构函数的次序正好与调用构造函数的次序相反： 最先被调用的构造函数，其对应的(同一对象中的)析构函数最后被调用，而最后被调用的构造函数，其对应的析构函数最先被调用。如图**9.1**示意。

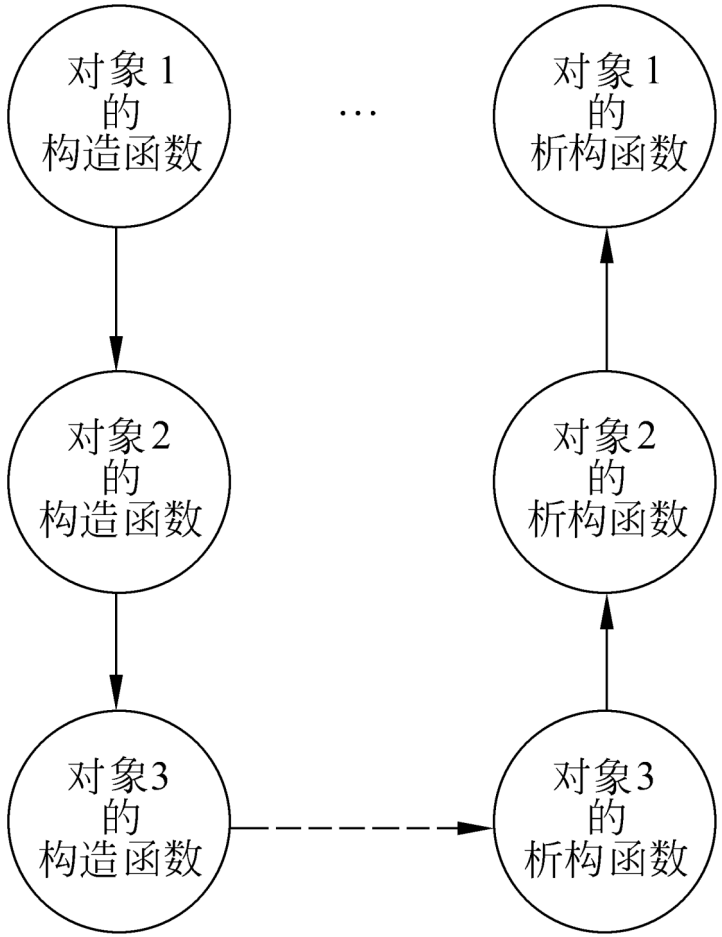

图9.1

但是，并不是在任何情况下都是按这一原则处理的。在第**4**章第**4.11**和**4.12**节中曾介绍过作用域和存储类别的概念，这些概念对于对象也是适用的。对象可以在不同的作用域中定义，可以有不同的存储类别。这些会影响调用构造函数和析构函数的时机。

下面归纳一下什么时候调用构造函数和析构函数：

(1) 在全局范围中定义的对象(即在所有函数之外定义的对象)，它的构造函数在文件中的所有函数(包括**main**函数)执行之前调用。但如果一个程序中有多个文件，而不同的文件中都定义了全局对象，则这些对象的构造函数的执行顺序是不确定的。当**main**函数执行完毕或调用**exit**函数时(此时程序终止)，调用析构函数。

(2) 如果定义的是局部自动对象(例如在函数中定义对象)，则在建立对象时调用其构造函数。如果函数被多次调用，则在每次建立对象时都要调用构造函数。在函数调用结束、对象释放时先调用析构函数。

(3) 如果在函数中定义静态**(static)**局部对象，则只在程序第一次调用此函数建立对象时调用构造函数一次，在调用结束时对象并不释放，因此也不调用析构函数，只在**main**函数结束或调用**exit**函数结束程序时，才调用析构函数。

构造函数和析构函数在面向对象的程序设计中是相当重要的。以上介绍了最基本的、使用最多的普通构造函数，在本章第**9.8**节中将会介绍复制构造函数，在第**10**章第**10.7**节中还要介绍转换构造函数。

9.4 对象数组

数组不仅可以由简单变量组成(例如整型数组的每一个元素都是整型变量)，也可以由对象组成(对象数组的每一个元素都是同类的对象)。

在日常生活中，有许多实体的属性是共同的，只是属性的具体内容不同。例如一个班有**50**个学生，每个学生的属性包括姓名、性别、年龄、成绩等。如果为每一个学生建立一个对象，需要分别取**50**个对象名。用程序处理很不方便。这时可以定义一个“学生类”对象数组，每一个数组元素是一个“学生类”对象。例如

```
Student stud[50];        //假设已声明了Student类，定义stud数组，有50个元素
```

在建立数组时，同样要调用构造函数。如果有**50**个元素，需要调用**50**次构造函数。在需要时可以在定义数组时提供实参以实现初始化。如果构造函数只有一个参数，在定义数组时可以直接在等号后面的花括号内提供实参。如

```
Student stud[3]={60,70,78};     //合法，3个实参分别传递给3个数组元素的构造函数
```

如果构造函数有多个参数，则不能用在定义数组时直接提供所有实参的方法，因为一个数组有多个元素，对每个元素要提供多个实参，如果再考虑到构造函数有默认参数的情况，很容易造成实参与形参的对应关系不清晰，出现歧义性。例如，类**Student**的构造函数有多个参数，且为默认参数：

```
Student：：Student(int=1001,int=18,int=60);   //定义构造函数，有多个参数，且为默认参数
```

如果定义对象数组的语句为

```
Student stud[3]={1005,60,70};
```

在程序中最好不要采用这种容易引起歧义性的方法。

编译系统只为每个对象元素的构造函数传递一个实参，所以在定义数组时提供的实参个数不能超过数组元素个数，如

```
Student stud[3]={60,70,78,45};      //不合法，实参个数超过对象数组元素个数
```

那么，如果构造函数有多个参数，在定义对象数组时应当怎样实现初始化呢？回答是：在花括号中分别写出构造函数并指定实参。如果构造函数有**3**个参数，分别代表学号、年龄、成绩。则可以这样定义对象数组：

```
Student Stud[3]={                    //定义对象数组
Student(1001,18,87),                 //调用第1个元素的构造函数，为它提供3个实参
Student(1002,19,76),                 //调用第2个元素的构造函数，为它提供3个实参
```

```
Student(1003,18,72)          //调用第3个元素的构造函数，为它提供3个实参
};
```

在建立对象数组时，分别调用构造函数，对每个元素初始化。每一个元素的实参分别用括号包起来，对应构造函数的一组形参，不会混淆。

例**9.6** 对象数组的使用方法。

```
#include <iostream>
using namespace std;
class Box
{public:
Box(int h=10,int w=12,int len=15): height(h),width(w),length(len){ }
//声明有默认参数的构造函数，用参数初始化表对数据成员初始化
int volume( );
private:
int height;
int width;
```

```
int length;
};

int Box::volume( )
{return(height*width*length);
}

int main( )
{ Box a[3]={                    //定义对象数组
Box(10,12,15),                 //调用构造函数Box，提供第1个元素的实参
Box(15,18,20),                 //调用构造函数Box，提供第2个元素的实参
Box(16,20,26)                  //调用构造函数Box，提供第3个元素的实参
};
cout<<"volume of a[0] is "<<a[0].volume( )<<endl;   //调用a[0]的volume函数
cout<<"volume of a[1] is "<<a[1].volume( )<<endl;   //调用a[1] 的volume函数
cout<<"volume of a[2] is "<<a[2].volume( )<<endl;   //调用a[2] 的volume函数
}
```

运行结果如下：

volume of a[0] is 1800

volume of a[1] is 5400

volume of a[2] is 8320

9.5 对象指针

9.5.1 指向对象的指针

在建立对象时，编译系统会为每一个对象分配一定的存储空间，以存放其成员。对象空间的起始地址就是对象的指针。可以定义一个指针变量，用来存放对象的指针。如果有一个类：

```
class Time
{public:
int hour;
int minute;
int sec;
void get_time( );
};
```

```
void Time::get_time( )
{cout<<hour<<":"<<minute<<":"<<sec<<endl;}
```

在此基础上有以下语句：

```
Time *pt;              //定义pt为指向Time类对象的指针变量
Time t1;               //定义t1为Time类对象
pt=&t1;                //将t1的起始地址赋给pt
```

这样，**pt**就是指向**Time**类对象的指针变量，它指向对象**t1**。

定义指向类对象的指针变量的一般形式为

类名 *对象指针名;

可以通过对象指针访问对象和对象的成员。如

```
*pt                pt所指向的对象，即t1。
(*pt).hour         pt所指向的对象中的hour成员，即t1.hour
pt->hour           pt所指向的对象中的hour成员，即t1.hour
(*pt).get_time ( )    调用pt所指向的对象中的get_time函数，即t1.get_time
pt->get_time ( )      调用pt所指向的对象中的get_time函数，即t1.get_time
```

9.5.2 指向对象成员的指针

对象有地址，存放对象初始地址的指针变量就是指向对象的指针变量。对象中的成员也有地址，存放对象成员地址的指针变量就是指向对象成员的指针变量。

1. 指向对象数据成员的指针

定义指向对象数据成员的指针变量的方法和定义指向普通变量的指针变量方法相同。例如

```
int *p1;                    //定义指向整型数据的指针变量
```

定义指向对象数据成员的指针变量的一般形式为

数据类型名 *指针变量名;

如果**Time**类的数据成员**hour**为公用的整型数据，则可以在类外通过指向对象数据成员的指针变量访问对象数据成员**hour**。

```
p1=&t1.hour;                //将对象t1的数据成员hour的地址赋给p1，p1指向t1.hour
cout<<*p1<<endl;            //输出t1.hour的值
```

2. 指向对象成员函数的指针

需要提醒读者注意： 定义指向对象成员函数的指针变量的方法和定义指向普通函数的指针变量方法有所不同。

成员函数与普通函数有一个最根本的区别： 它是类中的一个成员。编译系统要求在上面的赋值语句中，指针变量的类型必须与赋值号右侧函数的类型相匹配，要求在以下3方面都要匹配： ①函数参数的类型和参数个数；②函数返回值的类型；③所属的类。

定义指向成员函数的指针变量应该采用下面的形式：

```
void (Time::*p2)( );        //定义p2为指向Time类中公用成员函数的指针变量
```

定义指向公用成员函数的指针变量的一般形式为

数据类型名 (类名::*指针变量名)(参数表列);

可以让它指向一个公用成员函数，只需把公用成员函数的入口地址赋给一个指向公用成员函数的指针变量即可。如

```
p2=&Time::get_time;
```

使指针变量指向一个公用成员函数的一般形式为

指针变量名=&类名::成员函数名;

例9.7 有关对象指针的使用方法。

```
#include <iostream>
using namespace std;
class Time
{public:
Time(int,int,int);
int hour;
int minute;
int sec;
void get_time( );          //声明公有成员函数
};
```

```
Time::Time(int h,int m,int s)
{hour=h;
minute=m;
sec=s;
}
void Time::get_time( )          //定义公有成员函数
{cout<<hour<<":"<<minute<<":" <<sec<<endl;}

int main( )
{Time t1(10,13,56);             //定义Time类对象t1
int *p1=&t1.hour;               //定义指向整型数据的指针变量p1，并使p1指向
t1.hour
cout<<*p1<<endl;                //输出p1所指的数据成员t1.hour
t1.get_time( );                 //调用对象t1的成员函数get_time
Time *p2=&t1;                   //定义指向Time类对象的指针变量p2，并使p2指向t1
p2->get_time( );                //调用p2所指向对象(即t1)的get_time函数
void (Time::*p3)( );            //定义指向Time类公用成员函数的指针变量p3
p3=&Time::get_time;             //使p3指向Time类公用成员函数get_time
(t1.*p3)( );                    //调用对象t1中p3所指的成员函数(即t1.get_time( ))
}
```

程序运行结果为

10 **(main函数第4行的输出)**
10:13:56 **(main函数第5行的输出)**
10:13:56 **(main函数第7行的输出)**
10:13:56 **(main函数第10行的输出)**

可以看到为了输出**t1**中**hour,minute**和**sec**的值，可以采用**3**种不同的方法。

说明：

(1) 从**main**函数第**9**行可以看出： 成员函数的入口地址的正确写法是： **&**类名**::**成员函数名。

(2) main函数第**8**、**9**两行可以合写为一行：

```
void (Time::*p3)( )=&Time::get_time;        //定义指针变量时指定其指向
```

9.5.3 this 指针

在第8章中曾经提到过： 每个对象中的数据成员都分别占有存储空间，如果对同一个类定义了n个对象，则有n组同样大小的空间以存放n个对象中的数据成员。但是，不同对象都调用同一个函数代码段。

那么，当不同对象的成员函数引用数据成员时，怎么能保证引用的是所指定的对象的数据成员呢？假如，对于例9.6程序中定义的Box类，定义了3个同类对象a,b,c。如果有a.volume( )，应该是引用对象a中的height，width和length，计算出长方体a的体积。如果有b.volume( )，应该是引用对象b中的height，width和length，计算出长方体b的体积。而现在都用同一个函数段，系统怎样使它分别引用a或b中的数据成员呢？

在每一个成员函数中都包含一个特殊的指针，这个指针的名字是固定的，称为**this**。它是指向本类对象的指针，它的值是当前被调用的成员函数所在的对象的起始地址。例如，当调用成员函数**a.volume**时，编译系统就把对象**a**的起始地址赋给**this**指针，于是在成员函数引用数据成员时，就按照**this**的指向找到对象**a**的数据成员。例如**volume**函数要计算**height*width*length**的值，实际上是执行：

```
(this->height)*(this->width)*(this->length)
```

由于当前**this**指向**a**，因此相当于执行：

```
(a.height)*(a.width)*(a.length)
```

这就计算出长方体**a**的体积。同样如果有**b.volume( )**，编译系统就把对象**b**的起始地址赋给成员函数**volume**的**this**指针，显然计算出来的是长方体**b**的体积。

this指针是隐式使用的，它是作为参数被传递给成员函数的。本来，成员函数**volume**的定义如下：

```
int Box::volume( )
{return (height*width*length);
}
```

C++把它处理为

```
int Box::volume(Box *this)
{return(this->height * this->width * this->length);
}
```

即在成员函数的形参表列中增加一个**this**指针。在调用该成员函数时，实际上是用以下方式调用的：

```
a.volume(&a);
```

将对象**a**的地址传给形参**this**指针。然后按**this**的指向去引用其他成员。

需要说明：这些都是编译系统自动实现的，编程序者不必人为地在形参中增加**this**指针，也不必将对象**a**的地址传给**this**指针。

在需要时也可以显式地使用**this**指针。例如在**Box**类的**volume**函数中，下面两种表示方法都是合法的、相互等价的。

```
return(height * width * length);                    //隐含使用this指针
return(this->height * this->width * this->length);  //显式使用this指针
```

可以用***this**表示被调用的成员函数所在的对象，***this**就是**this**所指向的对象，即当前的对象。例如在成员函数**a.volume( )**的函数体中，如果出现***this**，它就是本对象**a**。上面的**return**语句也可写成

```
return((*this).height * (*this).width * (*this).length);
```

注意***this**两侧的括号不能省略，不能写成***this.height**。

所谓“调用对象**a**的成员函数**f**”，实际上是在调用成员函数**f**时使**this**指针指向对象**a**，从而访问对象**a**的成员。在使用“调用对象**a**的成员函数**f**”时，应当对它的含义有正确的理解。

9.6 共用数据的保护

C++虽然采取了不少有效的措施(如设private保护)以增加数据的安全性，但是有些数据却往往是共享的，人们可以在不同的场合通过不同的途径访问同一个数据对象。有时在无意之中的误操作会改变有关数据的状况，而这是人们所不希望出现的。

既要使数据能在一定范围内共享，又要保证它不被任意修改，这时可以使用const，即把有关的数据定义为常量。

9.6.1 常对象

在定义对象时指定对象为常对象。常对象必须要有初值，如

```
Time const t1(12,34,46);              //t1是常对象
```

这样，在所有的场合中，对象**t1**中的所有成员的值都不能被修改。凡希望保证数据成员不被改变的对象，可以声明为常对象。

定义常对象的一般形式为

类名 **const** 对象名**[(**实参表列**)];**

也可以把**const**写在最左面：

const 类名 对象名**[(**实参表列**)];**

二者等价。

如果一个对象被声明为常对象，则不能调用该对象的非**const**型的成员函数(除了由系统自动调用的隐式的构造函数和析构函数)。例如，对于例**9.7**中已定义的**Time**类，如果有

```
const Time t1(10,15,36);        //定义常对象t1
t1.get_time( );                 //企图调用常对象t1中的非const型成员函数，非法
```

这是为了防止这些函数会修改常对象中数据成员的值。不能仅依靠编程者的细心来保证程序不出错，编译系统充分考虑到可能出现的情况，对不安全的因素予以拦截。

现在，编译系统只检查函数的声明，只要发现调用了常对象的成员函数，而且该函数未被声明为**const**，就报错，提请编程者注意。

引用常对象中的数据成员很简单，只需将该成员函数声明为**const**即可。如

```
void get_time( ) const;        //将函数声明为const
```

这表示**get_time**是一个**const**型函数，即常成员函数。常成员函数可以访问常对象中的数据成员，但仍然不允许修改常对象中数据成员的值。

有时在编程时有要求，一定要修改常对象中的某个数据成员的值，**ANSI C++**考虑到实际编程时的需要，对此作了特殊的处理，对该数据成员声明为**mutable**，如

```
mutable int count;
```

把**count**声明为可变的数据成员，这样就可以用声明为**const**的成员函数来修改它的值。

9.6.2 常对象成员

可以将对象的成员声明为**const**，包括常数据成员和常成员函数。

1. 常数据成员

其作用和用法与一般常变量相似，用关键字**const**来声明常数据成员。常数据成员的值是不能改变的。有一点要注意： 只能通过构造函数的参数初始化表对常数据成员进行初始化。如在类体中定义了常数据成员**hour**：

```
const int hour;                    //声明hour为常数据成员
```

不能采用在构造函数中对常数据成员赋初值的方法。

在类外定义构造函数，应写成以下形式：

```
Time::Time(int h):hour(h){}        //通过参数初始化表对常数据成员hour初始化
```

常对象的数据成员都是常数据成员，因此常对象的构造函数只能用参数初始化表对常数据成员进行初始化。

2. 常成员函数

前面已提到：一般的成员函数可以引用本类中的非**const**数据成员，也可以修改它们。如果将成员函数声明为常成员函数，则只能引用本类中的数据成员，而不能修改它们，例如只用于输出数据等。如

```
void get_time( ) const;        //注意const的位置在函数名和括号之后
```

const是函数类型的一部分，在声明函数和定义函数时都要有**const**关键字，在调用时不必加**const**。常成员函数可以引用**const**数据成员，也可以引用非**const**的数据成员。**const**数据成员可以被**const**成员函数引用，也可以被非**const**的成员函数引用。具体情况可以用书中表**9.1**表示。

怎样利用常成员函数呢？

(1) 如果在一个类中，有些数据成员的值允许改变，另一些数据成员的值不允许改变，则可以将一部分数据成员声明为**const**，以保证其值不被改变，可以用非**const**的成员函数引用这些数据成员的值，并修改非**const**数据成员的值。

(2) 如果要求所有的数据成员的值都不允许改变，则可以将所有的数据成员声明为**const**，或将对象声明为**const(**常对象**)**，然后用**const**成员函数引用数据成员，这样起到“双保险”的作用，切实保证了数据成员不被修改。

(3) 如果已定义了一个常对象，只能调用其中的**const**成员函数，而不能调用非**const**成员函数(不论这些函数是否会修改对象中的数据)。这是为了保证数据的安全。如果需要访问对象中的数据成员，可将常对象中所有成员函数都声明为**const**成员函数，但应确保在函数中不修改对象中的数据成员。

不要误认为常对象中的成员函数都是常成员函数。常对象只保证其数据成员是常数据成员，其值不被修改。如果在常对象中的成员函数未加**const**声明，编译系统把它作为非**const**成员函数处理。

还有一点要指出： 常成员函数不能调用另一个非**const**成员函数。

9.6.3 指向对象的常指针

将指针变量声明为**const**型，这样指针值始终保持为其初值，不能改变。如

```
Time t1(10,12,15),t2;          //定义对象
Time * const ptr1;             //const位置在指针变量名前面，规定ptr1的值是常值
ptr1=&t1;                      //ptr1指向对象t1，此后不能再改变指向
ptr1=&t2;                      //错误，ptr1不能改变指向
```

定义指向对象的常指针的一般形式为

类名 * const 指针变量名;

也可以在定义指针变量时使之初始化，如将上面第2,3行合并为

```
Time * const ptr1=&t1;         //指定ptr1指向t1
```

请注意： 指向对象的常指针变量的值不能改变，即始终指向同一个对象，但可以改变其所指向对象(如**t1**)的值。

如果想将一个指针变量固定地与一个对象相联系(即该指针变量始终指向一个对象)，可以将它指定为**const**型指针变量。

往往用常指针作为函数的形参，目的是不允许在函数执行过程中改变指针变量的值，使其始终指向原来的对象。

9.6.4 指向常对象的指针变量

为了更容易理解指向常对象的指针变量的概念和使用，首先了解指向常变量的指针变量，然后再进一步研究指向常对象的指针变量。

下面定义了一个指向常变量的指针变量

ptr： const char *ptr;

注意**const**的位置在最左侧，它与类型名**char**紧连，表示指针变量**ptr**指向的**char**变量是常变量，不能通过**ptr**来改变其值的。

定义指向常变量的指针变量的一般形式为

const 类型名 ***指针变量名;**

说明：

(1) 如果一个变量已被声明为常变量，只能用指向常变量的指针变量指向它，而不能用一般的(指向非**const**型变量的)指针变量去指向它。

(2) 指向常变量的指针变量除了可以指向常变量外，还可以指向未被声明为**const**的变量。此时不能通过此指针变量改变该变量的值。如果希望在任何情况下都不能改变**c1**的值，则应把它定义为**const**型。

(3) 如果函数的形参是指向非**const**型变量的指针，实参只能用指向非**const**变量的指针，而不能用指向**const**变量的指针，这样，在执行函数的过程中可以改变形参指针变量所指向的变量(也就是实参指针所指向的变量)的值。

如果函数的形参是指向**const**型变量的指针，在执行函数过程中显然不能改变指针变量所指向的变量的值，因此允许实参是指向**const**变量的指针，或指向非**const**变量的指针。

使用形参和实参的对应关系见书中表**9.2**。

以上的对应关系与在**(2)**中所介绍的指针变量和其所指向的变量的关系是一致的： 指向常变量的指针变量可以指向**const**和非**const**型的变量，而指向非**const**型变量的指针变量只能指向非**const**的变量。

以上介绍的是指向常变量的指针变量，指向常对象的指针变量的概念和使用是与此类似的，只要将“变量”换成“对象”即可。

(1) 如果一个对象已被声明为常对象，只能用指向常对象的指针变量指向它，而不能用一般的(指向非**const**型对象的)指针变量去指向它。

(2) 如果定义了一个指向常对象的指针变量，并使它指向一个非**const**的对象，则其指向的对象是不能通过指针来改变的。如果希望在任何情况下**t1**的值都不能改变，则应把它定义为**const**型。

(3) 指向常对象的指针最常用于函数的形参，目的是在保护形参指针所指向的对象，使它在函数执行过程中不被修改。

请记住这样一条规则： 当希望在调用函数时对象的值不被修改，就应当把形参定义为指向常对象的指针变量，同时用对象的地址作实参(对象可以是**const**或非**const**型)。如果要求该对象不仅在调用函数过程中不被改变，而且要求它在程序执行过程中都不改变，则应把它定义为**const**型。

(4) 如果定义了一个指向常对象的指针变量，是不能通过它改变所指向的对象的值的，但是指针变量本身的值是可以改变的。

9.6.5 对象的常引用

过去曾介绍： 一个变量的引用就是变量的别名。实质上，变量名和引用名都指向同一段内存单元。如果形参为变量的引用名，实参为变量名，则在调用函数进行虚实结合时，并不是为形参另外开辟一个存储空间(常称为建立实参的一个拷贝)，而是把实参变量的地址传给形参(引用名)，这样引用名也指向实参变量。

例9.8 对象的常引用。

```
#include <iostream>
using namespace std;
class Time
{public:
Time(int,int,int);
int hour;
int minute;
int sec;
};
Time∷Time(int h,int m,int s)  //定义构造函数
{hour=h;
minute=m;
sec=s;
}

void fun(Time &t)           //形参t是Time类对象的引用
{t.hour=18;}

int main( )
{Time t1(10,13,56);          // t1是Time类对象
```

```
fun(t1);                    //实参是Time类对象，可以通过引用来修改实参t1的值
cout<<t1.hour<<endl;        //输出t1.hour的值为18
return 0;
}
```

如果不希望在函数中修改实参**t1**的值，可以把引用变量**t**声明为**const**(常引用)，函数原型为

```
void fun(const Time &t);
```

则在函数中不能改变**t**的值，也就是不能改变其对应的实参**t1**的值。

在**C++**面向对象程序设计中，经常用常指针和常引用作函数参数。这样既能保证数据安全，使数据不能被随意修改，在调用函数时又不必建立实参的拷贝。用常指针和常引用作函数参数，可以提高程序运行效率。

9.6.6 const型数据的小结

表9.3

形式	含义
Time const t1;	**t1**是常对象，其值在任何情况下都不能改变
void Time∷fun( )const	**fun**是**Time**类中的常成员函数，可以引用，但不能修改本类中的数据成员
Time * const p;	**p**是指向**Time**对象的常指针，**p**的值(即**p**的指向)不能改变
const Time *p;	**p**是指向**Time**类常对象的指针，其指向的类对象的值不能通过指针来改变
Time &t1=t;	**t1**是**Time**类对象**t**的引用，二者指向同一段内存空间

9.7 对象的动态建立和释放

用前面介绍的方法定义的对象是静态的，在程序运行过程中，对象所占的空间是不能随时释放的。但有时人们希望在需要用到对象时才建立对象，在不需要用该对象时就撤销它，释放它所占的内存空间以供别的数据使用。这样可提高内存空间的利用率。

在第7章7.1.7节中介绍了用**new**运算符动态地分配内存，用**delete**运算符释放这些内存空间。这也适用于对象，可以用**new**运算符动态建立对象，用**delete**运算符撤销对象。

如果已经定义了一个**Box**类，可以用下面的方法动态地建立一个对象：

```
new Box;
```

编译系统开辟了一段内存空间，并在此内存空间中存放一个**Box**类对象，同时调用该类的构造函数，以使该对象初始化(如果已对构造函数赋予此功能的话)。但是此时用户还无法访问这个对象，因为这个对象既没有对象名，用户也不知道它的地址。这种对象称为无名对象，它确实是存在的，但它没有名字。

用**new**运算符动态地分配内存后，将返回一个指向新对象的指针的值，即所分配的内存空间的起始地址。用户可以获得这个地址，并通过这个地址来访问这个对象。需要定义一个指向本类的对象的指针变量来存放该地址。如

```
Box *pt;                    //定义一个指向Box类对象的指针变量pt
pt=new Box;                 //在pt中存放了新建对象的起始地址
```

在程序中就可以通过**pt**访问这个新建的对象。如

```
cout<<pt->height;          //输出该对象的height成员
cout<<pt->volume( );       //调用该对象的volume函数，计算并输出体积
```

C++还允许在执行**new**时，对新建立的对象进行初始化。如

```
Box *pt=new Box(12,15,18);
```

这种写法是把上面两个语句(定义指针变量和用**new**建立新对象)合并为一个语句，并指定初值。这样更精炼。新对象中的**height**，**width**和**length**分别获得初值**12,15,18**。

调用对象既可以通过对象名，也可以通过指针。用**new**建立的动态对象一般是不用对象名的，是通过指针访问的，它主要应用于动态的数据结构，如链表。访问链表中的结点，并不需要通过对象名，

而是在上一个结点中存放下一个结点的地址，从而由上一个结点找到下一个结点，构成链接的关系。

在执行**new**运算时，如果内存量不足，无法开辟所需的内存空间，目前大多数**C++**编译系统都使**new**返回一个**0**指针值。只要检测返回值是否为**0**，就可判断分配内存是否成功。**ANSI C++**标准提出，在执行**new**出现故障时，就“抛出”一个“异常”，用户可根据异常进行有关处理。但**C++**标准仍然允许在出现**new**故障时返回**0**指针值。当前，不同的编译系统对**new**故障的处理方法是不同的。

在不再需要使用由**new**建立的对象时，可以用**delete**运算符予以释放。如

```
delete pt;          //释放pt指向的内存空间
```

这就撤销了**pt**指向的对象。此后程序不能再使用该对象。如果用一个指针变量**pt**先后指向不同的动态对象，应注意指针变量的当前指向，以免删错了对象。

在执行**delete**运算符时，在释放内存空间之前，自动调用析构函数，完成有关善后清理工作。

9.8 对象的赋值和复制

9.8.1 对象的赋值

如果对一个类定义了两个或多个对象，则这些同类的对象之间可以互相赋值，或者说，一个对象的值可以赋给另一个同类的对象。这里所指的对象的值是指对象中所有数据成员的值。

对象之间的赋值也是通过赋值运算符“=”进行的。本来，赋值运算符“=”只能用来对单个的变量赋值，现在被扩展为两个同类对象之间的赋值，这是通过对赋值运算符的重载实现的。实际这个过程是通过成员复制来完成的，即将一个对象的成员值一一复制给另一对象的对应成员。对象赋值的一般形式为

对象名1 = 对象名2;

注意对象名1和对象名2必须属于同一个类。

例如

```
Student stud1,stud2;          //定义两个同类的对象
 ┆
stud2=stud1;                  //将stud1赋给stud2
```

通过下面的例子可以了解怎样进行对象的赋值。

例9.9 对象的赋值。

```
#include <iostream>
using namespace std;
class Box
{public:
Box(int=10,int=10,int=10);          //声明有默认参数的构造函数
int volume( );
private:
int height;
int width;
```

```
int length;
};

Box::Box(int h,int w,int len)
{height=h;
width=w;
length=len;
}

int Box::volume( )
{return(height*width*length);      //返回体积
}

int main( )
{Box box1(15,30,25),box2;                    //定义两个对象box1和box2
cout<<"The volume of box1 is "<<box1.volume( )<<endl;
box2=box1;                                  //将box1的值赋给box2
cout<<"The volume of box2 is "<<box2.volume( )<<endl;
return 0;
}
```

运行结果如下：

The volume of box1 is 11250

The volume of box2 is 11250

说明：

(1) 对象的赋值只对其中的数据成员赋值，而不对成员函数赋值。

(2) 类的数据成员中不能包括动态分配的数据，否则在赋值时可能出现严重后果。

9.8.2 对象的复制

有时需要用到多个完全相同的对象。此外，有时需要将对象在某一瞬时的状态保留下来。这就是对象的复制机制。用一个已有的对象快速地复制出多个完全相同的对象。如

Box box2(box1);

其作用是用已有的对象**box1**去克隆出一个新对象**box2**。

其一般形式为

类名 对象**2**(对象**1**);

用对象**1**复制出对象**2**。

可以看到：它与前面介绍过的定义对象方式类似，但是括号中给出的参数不是一般的变量，而是对象。在建立对象时调用一个特殊的构造函数——复制构造函数(**copy constructor**)。这个函数的形式是这样的：

```
//The copy constructor definition.
Box∷Box(const Box& b)
{height=b.height;
width=b.width;
length=b.length;
}
```

复制构造函数也是构造函数，但它只有一个参数，这个参数是本类的对象(不能是其他类的对象)，而且采用对象的引用的形式(一般约定加**const**声明，使参数值不能改变，以免在调用此函数时因不慎而使对象值被修改)。

此复制构造函数的作用就是将实参对象的各成员值一一赋给新的对象中对应的成员。

回顾复制对象的语句

Box box2(box1);

这实际上也是建立对象的语句，建立一个新对象**box2**。由于在括号内给定的实参是对象，因此编译系统就调用复制构造函数(它的形参也是对象)，而不会去调用其他构造函数。实参**box1**的地址传递给形参**b(b**是**box1**的引用)，因此执行复制构造函数的函数体时，将**box1**对象中各数据成员的值赋给**box2**中各数据成员。

如果用户自己未定义复制构造函数，则编译系统会自动提供一个默认的复制构造函数，其作用只是简单地复制类中每个数据成员。

C++还提供另一种方便用户的复制形式，用赋值号代替括号，如

```
Box box2=box1;        //用box1初始化box2
```

其一般形式为

类名 对象名1 = 对象名2;

可以在一个语句中进行多个对象的复制。如

```
Box box2=box1,box3=box2;
```

按box1来复制box2和box3。可以看出： 这种形式与变量初始化语句类似，请与下面定义变量的语句作比较：

```
int a=4,b=a;
```

这种形式看起来很直观，用起来很方便。但是其作用都是调用复制构造函数。

请注意对象的复制和**9.8.1**节介绍的对象的赋值在概念上和语法上的不同。对象的赋值是对一个已经存在的对象赋值，因此必须先定义被赋值的对象，才能进行赋值。而对象的复制则是从无到有地建立一个新对象，并使它与一个已有的对象完全相同(包括对象的结构和成员的值)。

可以对例**9.7**程序中的主函数作一些修改：

```
int main( )
{Box box1(15,30,25);                          //定义box1
cout<<"The volume of box1 is "<<box1.volume( )<<endl;
Box box2=box1,box3=box2;                     //按box1来复制box2,box3
cout<<"The volume of box2 is "<<box2.volume( )<<endl;
cout<<"The volume of box3 is "<<box3.volume( )<<endl;
}
```

执行完第**3**行后，**3**个对象的状态完全相同。

请注意普通构造函数和复制构造函数的区别。

(1) 在形式上

类名(形参表列); //普通构造函数的声明，如**Box(int h,int w,int len);**

类名(类名& 对象名); //复制构造函数的声明，如**Box(Box &b);**

(2) 在建立对象时，实参类型不同。系统会根据实参的类型决定调用普通构造函数或复制构造函数。如

```
Box box1(12,15,16);        //实参为整数，调用普通构造函数
Box box2(box1);            //实参是对象名，调用复制构造函数
```

（**3**） 在什么情况下被调用

普通构造函数在程序中建立对象时被调用。

复制构造函数在用已有对象复制一个新对象时被调用，在以下**3**种情况下需要克隆对象：

① 程序中需要新建立一个对象，并用另一个同类的对象对它初始化，如前面介绍的那样。

② 当函数的参数为类的对象时。在调用函数时需要将实参对象完整地传递给形参，也就是需要建立一个实参的拷贝，这就是按实参复制一个形参，系统是通过调用复制构造函数来实现的，这样能保证形参具有和实参完全相同的值。如

```
void fun(Box b)            //形参是类的对象
{   }

int main( )
{Box box1(12,15,18);
fun(box1);                 //实参是类的对象，调用函数时将复制一个新对象b
return 0;
}
```

③ 函数的返回值是类的对象。在函数调用完毕将返回值带回函数调用处时。此时需要将函数中的对象复制一个临时对象并传给该函数的调用处。如

```
Box f( )                    //函数f的类型为Box类类型
{Box box1(12,15,18);
return box1;                //返回值是Box类的对象
}

int main( )
{Box box2;                  //定义Box类的对象box2
box2=f( );                  //调用f函数，返回Box类的临时对象，并将它赋值给box2
}
```

以上几种调用复制构造函数都是由编译系统自动实现的，不必由用户自己去调用，读者只要知道在这些情况下需要调用复制构造函数就可以了。

9.9 静态成员

如果有n个同类的对象，那么每一个对象都分别有自己的数据成员，不同对象的数据成员各自有值，互不相干。但是有时人们希望有某一个或几个数据成员为所有对象所共有。这样可以实现数据共享。

在第7章中曾介绍过全局变量，它能够实现数据共享。如果在一个程序文件中有多个函数，在每一个函数中都可以改变全局变量的值，全局变量的值为各函数共享。但是用全局变量的安全性得不到保证，由于在各处都可以自由地修改全局变量的值，很有可能偶一失误，全局变量的值就被修改，导致程序的失败。因此在实际工作中很少使用全局变量。

如果想在同类的多个对象之间实现数据共享，也不要用全局对象，可以用静态的数据成员。

9.9.1 静态数据成员

静态数据成员是一种特殊的数据成员。它以关键字**static**开头。例如

```
class Box
{public:
int volume( );
private:
static int height;                    //把height定义为静态的数据成员
int width;
int length;
};
```

如果希望各对象中的**height**的值是一样的，就可以把它定义为静态数据成员，这样它就为各对象所共有，而不只属于某个对象的成员，

所有对象都可以引用它。静态的数据成员在内存中只占一份空间。每个对象都可以引用这个静态数据成员。静态数据成员的值对所有对象都是一样的。如果改变它的值，则在各对象中这个数据成员的值都同时改变了。这样可以节约空间，提高效率。

说明：

(1) 在第**8**章中曾强调： 如果只声明了类而未定义对象，则类的一般数据成员是不占内存空间的，只有在定义对象时，才为对象的数据成员分配空间。但是静态数据成员不属于某一个对象，在为对象所分配的空间中不包括静态数据成员所占的空间。静态数据成员是在所有对象之外单独开辟空间。只要在类中定义了静态数据成员，即使不定义对象，也为静态数据成员分配空间，它可以被引用。

在一个类中可以有一个或多个静态数据成员，所有的对象共享这些静态数据成员，都可以引用它。

(2) 在第**7**章中曾介绍了静态变量的概念： 如果在一个函数中定义了静态变量，在函数结束时该静态变量并不释放，仍然存在并保留其值。现在讨论的静态数据成员也是类似的，它不随对象的建立而分配空间，也不随对象的撤销而释放**(**一般数据成员是在对象建立时分配空间，在对象撤销时释放**)**。静态数据成员是在程序编译时被分配空间的，到程序结束时才释放空间。

(3) 静态数据成员可以初始化，但只能在类体外进行初始化。如

```
int Box::height=10;          //表示对Box类中的数据成员初始化
```

其一般形式为

数据类型类名::静态数据成员名=初值;

不必在初始化语句中加**static**。

注意： 不能用参数初始化表对静态数据成员初始化。如在定义**Box**类中这样定义构造函数是错误的：

```
Box(int h,int w,int len):height(h){ }   //错误，height是静态数据成员
```

如果未对静态数据成员赋初值，则编译系统会自动赋予初值**0**。

(4) 静态数据成员既可以通过对象名引用，也可以通过类名来引用。

请观察下面的程序。

例**9.10** 引用静态数据成员。

```
#include <iostream>
using namespace std;
class Box
{public:
Box(int,int);
int volume( );
static int height;          //把height定义为公用的静态的数据成员
int width;
int length;
};
Box::Box(int w,int len)      //通过构造函数对width和length赋初值
{width=w;
length=len;
}
int Box::volume( )
{return(height*width*length);
}
int Box::height=10;          //对静态数据成员height初始化
```

```
int main( )
{
Box a(15,20),b(20,30);
cout<<a.height<<endl;          //通过对象名a引用静态数据成员
cout<<b.height<<endl;          //通过对象名b引用静态数据成员
cout<<Box::height<<endl;       //通过类名引用静态数据成员
cout<<a.volume( )<<endl;       //调用volume函数，计算体积，输出结果
}
```

上面3个输出语句的输出结果相同(都是10)。这就验证了所有对象的静态数据成员实际上是同一个数据成员。

请注意： 在上面的程序中将height定义为公用的静态数据成员，所以在类外可以直接引用。可以看到在类外可以通过对象名引用公用的静态数据成员，也可以通过类名引用静态数据成员。即使没有定义类对象，也可以通过类名引用静态数据成员。

这说明静态数据成员并不是属于对象的，而是属于类的，但类的对象可以引用它。

如果静态数据成员被定义为私有的，则不能在类外直接引用，而必须通过公用的成员函数引用。

(5) 有了静态数据成员，各对象之间的数据有了沟通的渠道，实现数据共享，因此可以不使用全局变量。全局变量破坏了封装的原则，不符合面向对象程序的要求。

但是也要注意公用静态数据成员与全局变量的不同，静态数据成员的作用域只限于定义该类的作用域内(如果是在一个函数中定义类，那么其中静态数据成员的作用域就是此函数内)。在此作用域内，可以通过类名和域运算符“::”引用静态数据成员，而不论类对象是否存在。

9.9.2 静态成员函数

成员函数也可以定义为静态的，在类中声明函数的前面加**static**就成了静态成员函数。如

```
static int volume( );
```

和静态数据成员一样，静态成员函数是类的一部分，而不是对象的一部分。如果要在类外调用公用的静态成员函数，要用类名和域运算符“::”。如

```
Box::volume( );
```

实际上也允许通过对象名调用静态成员函数，如

```
a.volume( );
```

但这并不意味着此函数是属于对象**a**的，而只是用**a**的类型而已。

与静态数据成员不同，静态成员函数的作用不是为了对象之间的沟通，而是为了能处理静态数据成员。

前面曾指出： 当调用一个对象的成员函数(非静态成员函数)时，系统会把该对象的起始地址赋给成员函数的**this**指针。而静态成员函数并不属于某一对象，它与任何对象都无关，因此静态成员函数没有**this**指针。既然它没有指向某一对象，就无法对一个对象中的非静态成员进行默认访问(即在引用数据成员时不指定对象名)。

可以说，静态成员函数与非静态成员函数的根本区别是： 非静态成员函数有**this**指针，而静态成员函数没有**this**指针。由此决定了静态成员函数不能访问本类中的非静态成员。

静态成员函数可以直接引用本类中的静态数据成员，因为静态成员同样是属于类的，可以直接引用。在**C++**程序中，静态成员函数主要用来访问静态数据成员，而不访问非静态成员。假如在一个静态成员函数中有以下语句：

```
cout<<height<<endl;          //若height已声明为static，则引用本类中的静态成员，合法
cout<<width<<endl;           //若width是非静态数据成员，不合法
```

但是，并不是绝对不能引用本类中的非静态成员，只是不能进行默认访问，因为无法知道应该去找哪个对象。如果一定要引用本类的非静态成员，应该加对象名和成员运算符“.”。如

```
cout<<a.width<<endl;         //引用本类对象a中的非静态成员
```

假设**a**已定义为**Box**类对象，且在当前作用域内有效，则此语句合法。

通过例**9.11**可以具体了解有关引用非静态成员的具体方法。

例**9.11** 静态成员函数的应用。

```
#include <iostream>
using namespace std;
class Student                    //定义Student类
{public:
Student(int n,int a,float s):num(n),age(a),score(s){ }      //定义构造函数
void total( );
static float average( );      //声明静态成员函数
private:
int num;
int age;
float score;
static float sum;             //静态数据成员
static int count;             //静态数据成员
};
void Student::total( )                    //定义非静态成员函数
{sum+=score;                              //累加总分
 count++;                                 //累计已统计的人数
}
```

```
float  Student::average( )              //定义静态成员函数
{return(sum/count);
}

float Student::sum=0;                   //对静态数据成员初始化
int Student::count=0;                   //对静态数据成员初始化

int main( )
{Student stud[3]={                      //定义对象数组并初始化
Student(1001,18,70),
Student(1002,19,78),
Student(1005,20,98)
};
int n;
cout<<"please input the number of students:";
 cin>>n;                                //输入需要求前面多少名学生的平均成绩
 for(int i=0;i<n;i++)                   //调用3次total函数
stud[i].total( );
 cout<<"the average score of "<<n<<" students is
"<<Student::average( )<<endl;
//调用静态成员函数
return 0;
}
```

运行结果为

please input the number of students:3↙

the average score of 3 students is 82.3333

说明：

(1) 在主函数中定义了**stud**对象数组，为了使程序简练，只定义它含**3**个元素，分别存放**3**个学生的数据。程序的作用是先求用户指定的**n**名学生的总分，然后求平均成绩**(n**由用户输入**)**。

(2) 在**Student**类中定义了两个静态数据成员**sum(**总分**)**和**count(**累计需要统计的学生人数**)**，这是由于这两个数据成员的值是需要进行累加的，它们并不是只属于某一个对象元素，而是由各对象元素共享的，可以看出： 它们的值是在不断变化的，而且无论对哪个对象元素而言，都是相同的，而且始终不释放内存空间。

(3) total是公有的成员函数，其作用是将一个学生的成绩累加到**sum**中。公有的成员函数可以引用本对象中的一般数据成员(非静态数据成员)，也可以引用类中的静态数据成员。**score**是非静态数据成员，**sum**和**count**是静态数据成员。

(4) average是静态成员函数，它可以直接引用私有的静态数据成员(不必加类名或对象名)，函数返回成绩的平均值。

(5) 在**main**函数中，引用**total**函数要加对象名(今用对象数组元素名)，引用静态成员函数**average**函数要用类名或对象名。

(6) 请思考： 如果不将**average**函数定义为静态成员函数行不行？程序能否通过编译？需要作什么修改？为什么要用静态成员函数？请分析其理由。

(7) 如果想在**average**函数中引用**stud[1]**的非静态数据成员**score**，应该怎样处理？

以上是在例**9.11**的基础上顺便说明静态成员函数引用非静态数据成员的方法，以帮助读者理解。但是在**C++**程序中最好养成这样的习惯： 只用静态成员函数引用静态数据成员，而不引用非静态数据成员。这样思路清晰，逻辑清楚，不易出错。

9.10 友元

在一个类中可以有公用的**(public)**成员和私有的**(private)**成员。在类外可以访问公用成员，只有本类中的函数可以访问本类的私有成员。现在，我们来补充介绍一个例外——友元**(friend)**。

友元可以访问与其有好友关系的类中的私有成员。友元包括友元函数和友元类。

9.10.1 友元函数

如果在本类以外的其他地方定义了一个函数(这个函数可以是不属于任何类的非成员函数，也可以是其他类的成员函数)，在类体中用**friend**对其进行声明，此函数就称为本类的友元函数。友元函数可以访问这个类中的私有成员。

1. 将普通函数声明为友元函数

通过下面的例子可以了解友元函数的性质和作用。

例9.12 友元函数的简单例子。

```
#include <iostream>
using namespace std;
class Time
{public:
Time(int,int,int);
friend void display(Time &);   //声明display函数为Time类的友元函数
 private:                      //以下数据是私有数据成员
int hour;
int minute;
int sec;
};

Time::Time(int h,int m,int s)     //构造函数，给hour,minute,sec赋初值
{hour=h;
```

```
minute=m;
sec=s;
}

void display(Time& t)            //这是友元函数，形参t是Time类对象的引用
{cout<<t.hour<<":"<<t.minute<<":"<<t.sec<<endl;}

int main( )
{ Time t1(10,13,56);
display(t1);
return 0;                        //调用display函数，实参t1是Time类对象
}
```

程序输出结果如下：

```
10:13:56
```

由于声明了**display**是**Time**类的**friend**函数，所以**display**函数可以引用**Time**中的私有成员**hour,minute,sec**。但注意在引用这些私有数据成员时，必须加上对象名，不能写成

```
cout<<hour<<":"<<minute<<":"<<sec<<endl;
```

因为**display**函数不是**Time**类的成员函数，不能默认引用**Time**类的数据成员，必须指定要访问的对象。

2. 友元成员函数

friend函数不仅可以是一般函数(非成员函数)，而且可以是另一个类中的成员函数。见例**9.13**。

例**9.13** 友元成员函数的简单应用。

在本例中除了介绍有关友元成员函数的简单应用外，还将用到类的提前引用声明，请读者注意。

```
#include <iostream>
using namespace std;
class Date;                  //对Date类的提前引用声明
class Time                   //定义Time类
{public:
Time(int,int,int);
void display(Date &);    //display是成员函数，形参是Date类对象的引用
 private:
int hour;
```

```
int minute;
int sec;
};

class Date                          //声明Date类
{public:
Date(int,int,int);
friend void Time::display(Date &);    //声明Time中的display函数为友元成员函数
 private:
int month;
int day;
int year;
};

Time::Time(int h,int m,int s)    //类Time的构造函数
{hour=h;
minute=m;
sec=s;
}
```

```
void Time::display(Date &d)      //display的作用是输出年、月、日和时、分、秒
{cout<<d.month<<"/"<<d.day<<"/"<<d.year<<endl;  //引用Date类对象中的私有数据
cout<<hour<<":"<<minute<<":"<<sec<<endl;       //引用本类对象中的私有数据
}

Date::Date(int m,int d,int y)        //类Date的构造函数
{month=m;
day=d;
year=y;
}

int main( )
{Time t1(10,13,56);            //定义Time类对象t1
Date d1(12,25,2004);          //定义Date类对象d1
t1.display(d1);               //调用t1中的display函数，实参是Date类对象d1
return 0;
}
```

运行时输出：

12/25/2004 (输出**Date**类对象**d1**中的私有数据)

10:13:56 (输出**Time**类对象**t1**中的私有数据)

在本例中定义了两个类**Time**和**Date**。程序第**3**行是对**Date**类的声明，因为在第**7**行和第**16**行中对**display**函数的声明和定义中要用到类名**Date**，而对**Date**类的定义却在其后面。能否将**Date**类的声明提到前面来呢？也不行，因为在**Date**类中的第**4**行又用到了**Time**类，也要求先声明**Time**类才能使用它。为了解决这个问题，**C++**允许对类作“提前引用”的声明，即在正式声明一个类之前，先声明一个类名，表示此类将在稍后声明。程序第**3**行就是提前引用声明，它只包含类名，不包括类体。如果没有第**3**行，程序编译就会出错。

在这里简要介绍有关对象提前引用的知识。在一般情况下，对象必须先声明，然后才能使用它。但是在特殊情况下(如上面例子所示的那样)，在正式声明类之前，需要使用该类名。但是应当注意： 类的提前声明的使用范围是有限的。只有在正式声明一个类以后才能用它去定义类对象。如果在上面程序第3行后面增加一行：

```
Date d1;          //企图定义一个对象
```

会在编译时出错。因为在定义对象时是要为这些对象分配存储空间的，在正式声明类之前，编译系统无法确定应为对象分配多大的空间。编译系统只有在“见到”类体后，才能确定应该为对象预留多大的空间。

在对一个类作了提前引用声明后，可以用该类的名字去定义指向该类型对象的指针变量或对象的引用变量(如在本例中，定义了**Date**类对象的引用变量)。这是因为指针变量和引用变量本身的大小是固定的，与它所指向的类对象的大小无关。

请注意程序是在定义**Time**::**display**函数之前正式声明**Date**类的。如果将对**Date**类的声明的位置(程序**13~21**行)改到定义**Time**::**display**函数之后，编译就会出错，因为在**Time**::**display**函数体中要用到**Date**类的成员**month,day,year**。如果不事先声明**Date**类，编译系统无法识别成员**month,day,year**等成员。

在一般情况下，两个不同的类是互不相干的。在本例中，由于在**Date**类中声明了**Time**类中的**display**成员函数是**Date**类的“朋友”，因此该函数可以引用**Date**类中所有的数据。请注意在本程序中调用友元函数访问有关类的私有数据方法：

（1） 在函数名**display**的前面要加**display**所在的对象名**(t1)**；

（2） **display**成员函数的实参是**Date**类对象**d1**，否则就不能访问对象**d1**中的私有数据；

（3） 在**Time**∷**display**函数中引用**Date**类私有数据时必须加上对象名，如**d.month**。

3. 一个函数(包括普通函数和成员函数)可以被多个类声明为“朋友”，这样就可以引用多个类中的私有数据

例如， 可以将例**9.13**程序中的**display**函数不放在**Time**类中，而作为类外的普通函数，然后分别在**Time**和**Date**类中将**display**声明为朋友。在主函数中调用**display**函数，**display**函数分别引用**Time**和**Date**两个类的对象的私有数据，输出年、月、日和时、分、秒。

9.10.2 友元类

不仅可以将一个函数声明为一个类的“朋友”，而且可以将一个类(例如**B**类)声明为另一个类(例如**A**类)的“朋友”。这时**B**类就是**A**类的友元类。友元类**B**中的所有函数都是**A**类的友元函数，可以访问**A**类中的所有成员。

在**A**类的定义体中用以下语句声明**B**类为其友元类：

friend B;

声明友元类的一般形式为

friend 类名;

关于友元，有两点需要说明：

(1) 友元的关系是单向的而不是双向的。

(2) 友元的关系不能传递。

在实际工作中，除非确有必要，一般并不把整个类声明为友元类，而只将确实有需要的成员函数声明为友元函数，这样更安全一些。

关于友元利弊的分析： 面向对象程序设计的一个基本原则是封装性和信息隐蔽，而友元却可以访问其他类中的私有成员，不能不说这是对封装原则的一个小的破坏。但是它能有助于数据共享，能提高程序的效率，在使用友元时，要注意到它的副作用，不要过多地使用友元，只有在使用它能使程序精炼，并能大大提高程序的效率时才用友元。

9.11 类模板

有时，有两个或多个类，其功能是相同的，仅仅是数据类型不同，如下面语句声明了一个类：

```
class Compare_int
{public:
Compare(int a,int b)
{x=a;y=b;}
int max( )
{return(x>y)?x:y;}
int min( )
{return(x<y)?x:y;}
private:
int x,y;
};
```

其作用是对两个整数作比较，可以通过调用成员函数**max**和**min**得到两个整数中的大者和小者。

如果想对两个浮点数**(float**型**)**作比较，需要另外声明一个类：

```
class Compare_float
{public:
Compare(float a,float b)
{x=a;y=b;}
float max( )
{return(x>y)?x:y;}
float min( )
{return(x<y)?x:y;}
private:
float x,y;
}
```

显然这基本上是重复性的工作，应该有办法减少重复的工作。**C++**在发展的后期增加了模板**(template)**的功能，提供了解决这类问题的途径。

可以声明一个通用的类模板，它可以有一个或多个虚拟的类型参数，如对以上两个类可以综合写出以下的类模板：

```
template<class numtype>              //声明一个模板，虚拟类型名为numtype
class Compare                        //类模板名为Compare
{public:
Compare(numtype a,numtype b)
{x=a;y=b;}
numtype max( )
{return (x>y)?x:y;}
numtype min( )
{return (x<y)?x:y;}
 private:
numtype x,y;
};
```

请将此类模板和前面第一个**Compare_int**类作一比较，可以看到有两处不同：

(1) 声明类模板时要增加一行

template <class 类型参数名**>**

(2) 原有的类型名**int**换成虚拟类型参数名**numtype**。在建立类对象时，如果将实际类型指定为**int**型，编译系统就会用**int**取代所有的**numtype**，如果指定为**float**型，就用**float**取代所有的**numtype**。这样就能实现“一类多用”。

由于类模板包含类型参数，因此又称为参数化的类。如果说类是对象的抽象，对象是类的实例，则类模板是类的抽象，类是类模板的实例。利用类模板可以建立含各种数据类型的类。

在声明了一个类模板后，怎样使用它？怎样使它变成一个实际的类？

先回顾一下用类来定义对象的方法：

Compare_int cmp1(4,7);　　// Compare_int是已声明的类

用类模板定义对象的方法与此相似，但是不能直接写成

Compare cmp(4,7);　　// Compare是类模板名

Compare是类模板名，而不是一个具体的类，类模板体中的类型**numtype**并不是一个实际的类型，只是一个虚拟的类型，无法用它去定义对象。必须用实际类型名去取代虚拟的类型，具体的做法是：

Compare <int> cmp(4,7);

即在类模板名之后在尖括号内指定实际的类型名，在进行编译时，编译系统就用**int**取代类模板中的类型参数**numtype**，这样就把类模板具体化了，或者说实例化了。这时**Compare<int>**就相当于前面介绍的**Compare_int**类。

例**9.14**是一个完整的例子。

例**9.14** 声明一个类模板，利用它分别实现两个整数、浮点数和字符的比较，求出大数和小数。

```
#include <iostream>
using namespace std;
template<class numtype>              //定义类模板
class Compare
{public:
Compare(numtype a,numtype b)
{x=a;y=b;}
```

```
numtype max( )
{return (x>y)?x:y;}
numtype min( )
{return (x<y)?x:y;}
private:
numtype x,y;
};

int main( )
{Compare<int> cmp1(3,7);          //定义对象cmp1，用于两个整数的比较
cout<<cmp1.max( )<<" is the Maximum of two integer numbers."<<endl;
cout<<cmp1.min( )<<" is the Minimum of two integer numbers."<<endl<<endl;
Compare<float> cmp2(45.78,93.6);  //定义对象cmp2，用于两个浮点数的比较
cout<<cmp2.max( )<<" is the Maximum of two float numbers."<<endl;
cout<<cmp2.min( )<<" is the Minimum of two float numbers."<<endl<<endl;
Compare<char> cmp3('a','A');      //定义对象cmp3，用于两个字符的比较
cout<<cmp3.max( )<<" is the Maximum of two characters."<<endl;
cout<<cmp3.min( )<<" is the Minimum of two characters."<<endl;
return 0;
}
```

运行结果如下：

7 is the Maximum of two integers.

3 is the Minimum of two integers.

93.6 is the Maximum of two float numbers.

45.78 is the Minimum of two float numbers.

a is the Maximum of two characters.

A is the Minimum of two characters.

还有一个问题要说明： 上面列出的类模板中的成员函数是在类模板内定义的。如果改为在类模板外定义，不能用一般定义类成员函数的形式：

```
numtype Compare::max( ) {...}          //不能这样定义类模板中的成员函数
```

而应当写成类模板的形式：

```
template<class numtype>
```

```
numtype Compare<numtype>::max( )
{{return (x>y)?x:y;}
```

归纳以上的介绍，可以这样声明和使用类模板：

(1) 先写出一个实际的类。由于其语义明确，含义清楚，一般不会出错。

(2) 将此类中准备改变的类型名(如**int**要改变为**float**或**char**)改用一个自己指定的虚拟类型名(如上例中的**numtype**)。

(3) 在类声明前面加入一行，格式为

template<class 虚拟类型参数**>**，如

```
template<class numtype>              //注意本行末尾无分号
class Compare
{…};                                 //类体
```

(4) 用类模板定义对象时用以下形式：

类模板名<实际类型名> 对象名;

类模板名<实际类型名> 对象名(实参表列);

如

```
Compare<int> cmp;
Compare<int> cmp(3,7);
```

(5) 如果在类模板外定义成员函数，应写成类模板形式：

template<class 虚拟类型参数>

函数类型 类模板名<虚拟类型参数>::成员函数名(函数形参表列) {…}

说明：

（1） 类模板的类型参数可以有一个或多个，每个类型前面都必须加**class**，如

```
template<class T1,class T2>
class someclass
{…};
```

在定义对象时分别代入实际的类型名，如

```
someclass<int,double> obj;
```

(2) 和使用类一样，使用类模板时要注意其作用域，只能在其有效作用域内用它定义对象。

(3) 模板可以有层次，一个类模板可以作为基类，派生出派生模板类。