# 第 3 章 怎样使用类和对象

通过第 2 章的学习,已经对类和对象有了初步的了解。在本章中将进一步说明怎样使用类和对象。本章将会遇到一些稍复杂的概念,我们尽量用读者容易理解的方式进行介绍,也请读者细心阅读。

## 3.1 类对象的初始化

### 3.1.1 需要对类对象进行初始化

在程序中常常需要对变量赋初值,即对其初始化。这在面向过程的程序中是很容易实现的,在定义变量时赋以初值。如

```
int a=10;                    //定义整型变量 a,a 的初值为 10
```

在基于对象的程序中,在定义一个对象时,有可能需要作初始化的工作,包括对数据成员赋初值。对象代表一个实体,每一个对象都有它确定的属性。例如有一个 Time 类(时间),用它定义对象 t1,t2,t3。显然,t1,t2,t3 分别代表 3 个不同的时间(时、分、秒)。每一个对象都应当在它建立之时就有确定的内容,否则就失去对象的意义了。在系统为对象分配内存时,应该同时对有关的数据成员赋以初始值。

那么,怎样使它们得到初值呢?有人企图在声明类时对数据成员初始化。如

```
class Time
  { hour=0;                  //不能在类声明中对数据成员初始化
    minute=0;
    sec=0;
  };
```

这是错误的。因为类并不是一个实体,而是一种抽象类型,并不占存储空间,显然无处容纳数据。

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类似的,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 的数据成员,就不能用这种方法初始化。

在第2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2.3中的 set_time 函数)。从例2.3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,这样的程序哪里还有质量和效率?应当找到一种方便的方法对类对象中的数据成员进行初始化。

## 3.1.2 用构造函数实现数据成员的初始化

在C++程序中,对象的初始化是一个重要的问题。不应该使程序员在这个问题上花费过多的精力,C++在类的设计中提供了较好的处理方法。

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**。构造函数的是在对类进行声明的时候由类的设计者定义的,程序用户只须在定义对象的同时指定数据成员的初值即可。

构造函数的名字必须与类名同名,而不能任意命名,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。

先观察下面的例子。

**例3.1** 在例2.3基础上,用构造函数为对象的数据成员赋初值。

**编写程序:**

```
#include<iostream>
using namespace std;
class Time                    //声明 Time 类
  {public:                    //以下为公用函数
    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();          //成员函数声明
    void show_time();         //成员函数声明
   private:                   //以下为私有数据
    int hour;
    int minute;
```

```
    int sec;
};

void Time::set_time()          //定义成员函数,向数据成员赋新值
  {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()                     //主函数
  {
   Time t1;                    //建立对象 t1,同时调用构造函数 t1.Time()
   t1.set_time();              //对 t1 的数据成员赋新值
   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;
  }
```

**运行结果：**

```
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)
```

**程序分析：**

在类中定义了构造函数 Time,它和所在的类同名。在建立对象时自动执行构造函数,根据构造函数 Time 的定义,其作用是对该对象中的全部数据成员赋以初值 0。不要误认为是在声明类时直接对程序数据成员赋初值(那是不允许的),赋值语句是写在构造函数 Time 的函数体中的,只有在调用构造函数 Time 时才执行这些赋值语句,对当前的对象中的数据成员赋值。

执行主函数时,首先建立对象 t1,此时自动执行构造函数 Time。在执行构造函数 Time 过程中对 t1 对象中的数据成员赋初值 0。然后再执行主函数中的 t1.set_time 函数,从键盘输入新值赋给对象 t1 的数据成员,再输出 t1 的数据成员的值。接着建立对象 t2,同时自动执行构造函数 Time,对 t2 中的数据成员赋初值 0。但主函数中没有对 t2 的数据成员再赋新值,直接输出数据成员的初值。

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) 什么时候调用构造函数呢?在建立类对象时会自动调用构造函数。在建立对象时系统为该对象分配存储单元,此时执行构造函数,就把指定的初值送到有关数据成员的存储单元中。每建立一个对象,就调用一次构造函数。在上面的程序中,在主函数中定义了一个对象 t1,此时就会自动调用 t1 对象中的构造函数 Time,使各数据成员的值为 0。

(2) 构造函数没有返回值,它的作用只是对对象进行初始化。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。不能写成

```
int Time()
  {…}
```

或

```
void time()
  {…}
```

(3) 构造函数不需要用户调用,也不能被用户调用。下面的用法是错误的:

```
t1.Time();                      //企图用调用一般成员函数的方法来调用构造函数
```

构造函数是在定义对象时由系统自动执行的,而且只能执行一次。构造函数一般声明为 public。

(4) 可以用一个类对象初始化另一个类对象,如

```
Time t1;                        //建立对象 t1,同时调用构造函数 t1.Time()
Time t2=t1;                     //建立对象 t2, 并用 t1 初始化 t2
```

此时,把对象 t1 的各数据成员的值复制到 t2 相应各成员,而不调用构造函数 t2.Time()。

(5) 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,例如 cout 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。

(6) 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,只是这个构造函数函数体是空的,也没有参数,不执行初始化操作。

(7)以上介绍的构造函数是最基本的形式。构造函数有不同的形式以便用于不同情况的数据初始化。从 3.1.3 节开始会分别介绍。

### 3.1.3 用带参数的构造函数对不同对象初始化

在例 3.1 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

每一个对象的数据成员都得到同一组初值(例如例 3.1 中各个对象的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,这时就无法使用上面的办法来解决了。

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,以实现对不同对象的初始化。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(如 fun(a,b);)。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,…);**

在建立对象时把实参的值传递给构造函数相应的形参,把它们作为数据成员的初值。

**例 3.2**　有两个长方柱,其高、宽、长分别为 12,25,30;15,30,21。求它们的体积。编写一个基于对象的程序,在类中用带参数的构造函数对数据成员初始化。

**编写程序:**

```
#include<iostream>
using namespace std;
class Box                          //声明 Box 类
  {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;                   //高
     int width;                    //宽
     int length;                   //长
  };
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 的高、宽、长的值
   cout<<"The volume of box1 is "<<box1.volume()<<endl;
   Box box2(15,30,21);             //建立对象 box2,并指定 box2 的高、宽、长的值
   cout<<"The volume of box2 is "<<box2.volume()<<endl;
```

```
    return 0;
}
```

**运行结果：**

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

**程序分析：**

构造函数 Box 有 3 个参数(h,w,l),分别代表高、宽、长。在主函数中定义对象 box1 时,同时给出函数的实参 12,25,30。系统自动调用对象 box1 中的构造函数 Box,通过虚实结合,对 box1 中的 height,width,length 进行赋值。然后由主函数中的 cout 语句调用函数 box1.volume(),并输出 box1 的体积。对 box2 也类似。

**注意**：定义对象的语句形式是

```
Box box1(12,25,30);
```

可以看到：

(1) 带参数的构造函数中的形参,其对应的实参是在建立对象时给定的,即在建立对象时同时指定数据成员的初值。

(2) 定义不同对象时用的实参是不同的,它们反映不同对象的属性。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。

这种初始化对象的方法,使用起来很方便,很直观。从定义语句中直接看到数据成员的初值。

### 3.1.4 在构造函数中用参数初始化表对数据成员初始化

在 3.1.3 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++还提供另一种更简化的方法——**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 3.2 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){ }
```

即在原来函数首部的末尾加一个冒号,然后列出参数的初始化表。上面的初始化表表示：用形参 h 的值初始化数据成员 height,用形参 w 的值初始化数据成员 width,用形参 len 的值初始化数据成员 length。后面的花括号是空的,即函数体是空的,没有任何执行语句。这种形式的构造函数的作用和例 3.2 中在类外定义的 Box 构造函数相同。用参数的初始化表法可以减少函数体的长度,使结构函数显得精练简单。这样就可以直接在**类体**中(而不是在类外)定义构造函数。尤其当需要初始化的数据成员较多时更显其优越性。许多C++程序人员喜欢用这种方法初始化所有数据成员。

带有参数初始化表的构造函数的一般形式如下：

**类名:: 构造函数名([参数表])[:成员初始化表]**

**{**

```
    [构造函数体]
  }
```

其中,方括号内为可选项(可有可无)。

**说明**:如果数据成员是数组,则应当在构造函数的函数体中用语句对其赋值,而不能在参数初始化表中对其初始化。如

```
class Student
  {public:
    Student(int n,char s,nam[]):num(n),sex(s)        //定义构造函数
      {strcpy(name,nam);}                            //函数体
   private:
    int num;.
    char sex;
    char name[20];
};
```

可以这样定义对象 stud1:

```
Student stud1(10101,'m',"Wang_li");
```

利用初始化表,把形参 n 得到的值 10101 赋给私有数据成员 num,把形参 s 得到的值' m' 赋给 sex,把形参数组 nam 的各元素的值通过 strcpy 函数复制到 name 数组中。这样对象 stud1 中所有的数据成员都初始化了,此对象是有确定内容的。

### 3.1.5　可以对构造函数进行重载

在一个类中可以定义多个构造函数,以便为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**。1.3.4 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 3.3**　在例 3.2 的基础上,定义两个构造函数,其中一个无参数,一个有参数。

**编写程序**:

```
#include<iostream>
using namespace std;
class Box
  {public:
    Box();                          //声明一个无参的构造函数 Box
    Box(int h,int w,int len):height(h),width(w),length(len){ }
                                    //定义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();                   //声明成员函数 volume
   private:
    int height;
    int width;
    int length;
  };
```

```
Box::Box()                          //在类外定义无参构造函数 Box
  {height=10;
   width=10;
   length=10;
  }

int Box::volume()                   //在类外定义成员函数 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;
  }
```

**运行结果:**

```
The volume of box1 is 1000
The volume of box2 is 11250
```

**程序分析:**

在类中声明了一个无参数构造函数 Box,在类外定义的函数体中对私有数据成员赋值。第 2 个构造函数 Box 是直接在类体中定义的,用参数初始化表对数据成员初始化,此函数有 3 个参数,需要 3 个实参与之对应。这两个构造函数同名(都是 Box),那么系统怎么辨别调用的是哪一个构造函数呢?是根据函数调用的形式去确定对应哪一个构造函数。

在主函数中,建立对象 box1 时没有给出参数,系统找到与之对应的无参构造函数 Box,执行此构造函数的结果是使 3 个数据成员的值均为 10。然后输出 box1 的体积 1000。建立对象 box2 时给出 3 个实参,系统找到有 3 个形参的构造函数 Box 与之对应,执行此构造函数的结果是使 3 个数据成员的值为 15,30,25。然后输出 box2 的体积 11250。

在本程序中定义了两个同名的构造函数,其实还可以定义更多的重载构造函数。例如还可以有以下的构造函数原型:

```
Box::Box(int h);                    //有一个参数的构造函数
Box::Box(int h,int w);              //有两个参数的构造函数
```

在建立对象时可以给出一个参数和两个参数,系统会分别调用相应的构造函数。

**说明:**

(1) 在建立对象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。如果用户未定义构造函数,则系统会自动提供一个默认构造函数,但它的函数体是空的,

不起初始化作用。如果用户希望在创建对象时就能使数据成员有初值,就必须自己定义构造函数。

(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。如本程序中有以下定义对象的语句:

```
Box box1;                          //建立对象的正确形式
```

注意不要写成

```
Box box1();                        //建立对象的错误形式,不应该有括号
```

上面的语句并不是定义 Box 类的对象 box1,而是声明一个普通函数 box1,此函数的返回值为 Box 类型。在程序中不应出现调用无参构造函数(如 Box())。请记住:构造函数是不能被用户显式调用的。

(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。

### 3.1.6　构造函数可以使用默认参数

**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,编译系统就使形参的值为默认值**。在实际生活中常有一些这样的初始值:计数器的初始值一般默认为 0,战士的性别一般默认为“男”,天气默认为“晴”等,如果实际情况不是这些值,则由用户另行指定。这样可以减少输入量。

在 1.3.6 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。例如,例 3.3 的问题也可以使用包含默认参数的构造函数来处理。

**例 3.4**　将例 3.3 程序中的构造函数改用含默认值的参数,宽、高、长的默认值均为 10。

在例 3.3 程序的基础上改写。

**编写程序:**

```
#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 Box 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 Box 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;                                    //没有给定实参
   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);                             //只给定两个实参
   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;
  }
```

**运行结果:**

```
The volume of box1 is 1000
The volume of box2 is 1500
The volume of box3 is 4500
The volume of box4 is 9000
```

**程序分析:**

由于在定义对象box1时没有给实参,系统就调用默认构造函数,各形参的值均取默认值10。即

```
box1.height=10;box1.width=10;box1.length=10
```

在定义对象box2时只给定一个实参15,它传给形参h(长方柱的高),形参w和len未得到实参过来的值,就取默认值10。即

```
box2.height=15; box2.width=10;box2.length=10;
```

同理:

```
box3.height=15; box3.width=30;box3.length=10;
box4.height=15; box4.width=30;box4.length=20;
```

程序中对构造函数的定义(第12至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){ }
```

只需要一行就够了,简单方便。

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,无须输入数据,对象全按事先指定的值进行初始化。

**说明:**

(1) 应该在什么地方指定构造函数的默认参数?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。因为类定义是放在头文件中的,它是类的对外接口,用户是可以看到的,而函数的定义是类的实现细节,用户往往是看不到的。在声明构造函数时指定默认参数值,使用户知道在建立对象时怎样使用默认参数。

(2) 程序第 5 行在声明构造函数时,形参名可以省略,即写成

```
Box(int=10,int=10,int=10);
```

(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。由于不需要实参也可以调用构造函数,因此全部参数都指定了默认值的构造函数也属于默认构造函数。前面曾提到过:一个类只能有一个默认构造函数,也就是说,可以不使用参数而调用的构造函数,一个类只能有一个。其道理是显然的,是为了避免调用时的歧义性。如果同时定义了下面两个构造函数,是错误的。

```
Box();                                  //声明一个无参的构造函数
Box(int=10,int=10,int=10);              //声明一个全部参数都指定了默认值的构造函数
```

在建立对象时,如果写成

```
Box box1;
```

编译系统无法识别应该调用哪个构造函数,出现了歧义性,编译时报错。应该避免这种情况。

(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。例如在一个类中有以下构造函数的声明:

```
Box(int=10,int=10,int=10);              //指定全部为默认参数
Box();                                  //声明无参的构造函数
Box(int,int);                           //声明有两个参数的构造函数
```

若有以下定义语句:

```
Box box1;                   //是调用上面第 1 个构造函数,还是调用第 2 个构造函数
Box box2(15,30)             //是调用上面第 1 个构造函数,还是调用第 3 个构造函数
```

应该执行哪一个构造函数呢?出现了歧义性。但如果构造函数中的参数并非全部为默认值时,就要分析具体情况。如有以下 3 个原型声明:

```
Box();                      //无参的构造函数
Box(int,int=10,int=10);     //有一个参数不是默认参数
Box(int,int);               //有两个参数的构造函数
```

若有以下定义对象的语句:

```
Box box1;                    //正确,不出现歧义性,调用第1个构造函数
Box box2(15);                //调用第2个构造函数
Box box3(15,30);             //错误,出现歧义性
```

很容易出错,要十分仔细。因此,一般不应同时使用构造函数的重载和有默认参数的构造函数。

### 3.1.7 用构造函数实现初始化方法的归纳

(1) 在类中定义构造函数的函数体中对数据进行赋初值,如例3.1中:

```
public:
Time()
  {hour=0;.
   minute=0;
   sec=0;
  }
```

在建立对象时执行构造函数,给数据赋初值。如果定义了多个对象,每个对象中的数据的初值都是相同的(今为0)。

(2) 用带参数的构造函数,可以使同类的不同对象中的数据具有不同的初值,如例3.2,在类中定义构造函数:

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

在定义对象时指定实参。

```
Box box1(12,25,30);
```

把12,25,30传递给构造函数的形参,再赋给对象中各数据。不同的对象可以有不同的初值。

(3) 在构造函数中用参数初始化表实现对数据赋初值。如:

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

其作用与(2)相同,但免去了(2)中定义的函数体,使构造函数简单精练,使用方便。定义对象的形式与(2)相同:

```
Box box1(12,25,30);
```

(4) 在定义构造函数时可以使用默认参数。如例3.3中:

```
Box(int h=10,int w=10,int len=10)
  {height=h;
   width=w;
```

```
    length=len;
  }
```

上面的构造函数可以改用参数初始化表如下：

```
Box(int h=10,int w=10,int len=10):height(h),width(w),length(len) { };
```

这样更为简洁方便。

在定义对象时，如果不指定实参，则以默认参数作为初值。如：

```
Box box1;                    //此时 h=10,w=10,len=10
Box box1(12);                //此时 h=12,w=10,len=10
Box box1(12,25);             //此时 h=12,w=25,len=10
Box box1(12,25,30);          //此时 h=12,w=25,len=30
```

(5) 构造函数可以重载，即在一个类中定义多个同名的构造函数。如例 3.4 中定义了两个同名的构造函数 Box：

```
Box()                        //定义无参构造函数 Box
  {height=10;
   width=10;
   length=10;
  }
Box(int h,int w,int len): height(h),width(w),length(len){ };
    //定义有参构造函数 Box,用初始化表对数据初始化
```

定义对象：

```
Box box1;                    //不指定实参,调用无参构造函数 Box
Box box2(15,30,25);;         //指定 3 个实参,调用有参构造函数 Box
```

**注意**：一般不应同时使用有默认参数的构造函数和构造函数的重载，容易出现歧义。

### 3.1.8　利用析构函数进行清理工作

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(假设是自动局部对象)，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 静态(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的对象，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，调用该全局的对象的析构函数。

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。程序设计者要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。由于没有函数参数,因此它不能被重载。一个类可以有多个构造函数,但是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 3.5** 包含构造函数和析构函数的C++程序。

**编写程序:**

```
#include<string>
#include<iostream>
using namespace std;
class Student                                    //声明 Student 类
  {public:
     Student(int n,string nam,char s)            //定义有参数的构造函数
     {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; }      //输出指定的信息

     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()                                       //主函数
  {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_fan",'m');    //定义对象 stud2
    stud2.display();                         //输出学生 2 的数据
    return 0;
}
```

**运行结果：**

```
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_fan
sex: m

Destructor called.                           (执行 stud2 的析构函数)
Destructor called.                           (执行 stud1 的析构函数)
```

**程序分析：**

在 main 函数的前面声明类，它的作用域是全局的。这样做可以使 main 函数更简练一些。在 Student 类中定义了构造函数和析构函数。在执行 main 函数时先建立对象 stud1，在建立对象时调用对象的构造函数，给该对象中数据成员赋初值。然后调用 stud1 的 display 函数，输出 stud1 的数据成员的值。接着建立对象 stud2，在建立对象时调用 stud2 的构造函数，然后调用 stud2 的 display 函数，输出 stud2 的数据成员的值。

至此，主函数中的语句已执行完毕，对主函数的调用结束了，在主函数中建立的对象是局部的，它的生命期随着主函数的结束而结束，在撤销对象之前的最后一项工作是调用析构函数。在本例中，析构函数并无任何实质上的作用，只是输出一个信息。我们在这里使用它，只是为了说明析构函数的使用方法。

最后两行是哪一个对象的析构函数所输出的呢？在行右侧的括号内做了说明，请读者先考虑一下，在 3.1.9 节中将作进一步的说明。

## 3.1.9　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。

有的读者在看到 3.1.8 节例 3.5 程序输出结果的最后两行时，还以为该两行右侧括号内的说明是印错了。许多人会自然地认为：应该是先执行 stud1 的析构函数，然后再执行 stud2 的析构函数啊。是不是书上把 stud1 和 stud2 印反了？实际上，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 3.1 所示。可简记为：**先构造的后析构，后构造的先析构**。它相当于一个栈，**先进后出**。

读者可能还不放心：你怎么知道先执行的是 stud2 的析构函数呢？请读者自己先想

出一种方法来验证。

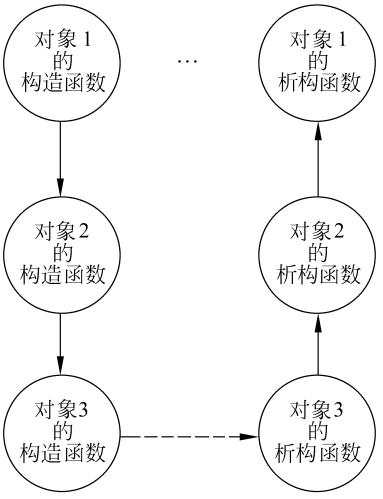


图 3.1

这里提供一个简单的方法：将 Student 类中定义的析构函数的函数体改为

```
cout<<"Destructor called."<<num<<endl;
```

即在输出时增加一项 num，输出本对象中数据成员 num 的值(学号)。这样就可以从输出结果中分析出输出的是哪个对象中学生的学号，从而确定执行的是哪个对象的析构函数。

修改后的运行结果的最后两行为

```
Destructor called.10011
                    (可见执行的是 stud2 的析构函数)
Destructor called.10010
                    (可见执行的是 stud1 的析构函数)
```

10011 是对象 stud2 中的成员 num 的值，10010 是对象 stud1 中的成员 num 的值。这就清楚地表明了：**先构造的后析构，后构造的先析构**。

上面曾提到：在一般情况下，调用析构函数的次序与调用构造函数的次序相反。这是对同一类存储类别的对象而言的。例如例 3.5 程序中的 stud1 和 stud2 是在同一个函数中定义的局部函数，它们的特性相同，按照“先构造的后析构，后构造的先析构”的原则处理。

但是，并不是在任何情况下都是按这一原则处理的。在学习 C 语言时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。

下面归纳一下系统在什么时候调用构造函数和析构函数：

(1) 如果在全局范围中定义对象(即在所有函数之外定义的对象)，那么它的构造函数在本文件模块中的所有函数(包括 main 函数)执行之前调用。但如果一个程序包含多个文件，而在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。如果对象所在的函数被多次调用，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 1 次调用此函数定义对象时调用构造函数一次，在调用函数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。

例如，在一个函数中定义了两个对象：

```
void fn()
  {Student stud1;                          //定义自动局部对象
   static Student stud2;                   //定义静态局部对象
    ⋮
```

```
}
```

在调用 fn 函数时,先建立 stud1 对象、调用 stud1 的构造函数,再建立 stud2 对象、调用 stud2 的构造函数。在 fn 调用结束时,对象 stud1 是要释放的(因为它是自动局部对象),因此调用 stud1 的析构函数。而 stud2 是静态局部对象,在 fn 调用结束时并不释放,因此不调用 stud2 的析构函数。直到程序结束释放 stud2 时,才调用 stud2 的析构函数。可以看到 stud2 是后调用构造函数的,但并不先调用其析构函数。原因是两个对象的存储类别不同、生命周期不同。

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,是类的设计中的一个重要部分。以上介绍了最基本的、使用最多的普通构造函数,在 3.6 节中将会介绍**复制构造函数**,4.7 节中还要介绍**转换构造函数**。在以后深入的学习和编程实践中将会进一步掌握它们的应用。

**说明**:在上面几节中介绍了通过构造函数给类对象中的数据成员赋初值的方法(在第 5 章还要介绍对派生类对象的初始化,更复杂一些)。有的读者可能觉得太复杂了,难以掌握。应当说明,构造函数是由类的设计者定义的,或者说,是声明一个类的一部分。在一般情况下,类的设计(声明) 者和类的使用者不是同一个人。在头文件中声明了一个类(包括定义构造函数) 后,用户只要用#include 指令包含了此文件,就可以用这个类来定义对象。在定义对象时进行初始化是比较简单的,如

```
Box box(15,30,20);                                    //定义对象 box 时给出 3 个初值
```

作为用户可以不关心构造函数的具体写法和在构造函数中如何赋初值的细节,只要知道构造函数的原型(知道有几个形参、形参的类型以及顺序) 以及怎样使用即可,并不需要 C++的编程人员都自己写构造函数。

## 3.2　对象数组

学过 C 语言的读者对数组的概念应当比较熟悉了。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),也可以由类对象组成(**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,其属性包括姓名、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};
```

这 3 个实参与形参的对应关系是怎样的?是为每个对象各提供第一个实参呢?还是全部作为第一个对象的 3 个实参呢?编译系统是这样处理的:这 3 个实参分别作为 3 个元素的第 1 个实参。读者可以自己上机验证一下。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名并在括号内指定实参。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,向它提供 3 个实参
  Student(1002,19,76),       //调用第 2 个元素的构造函数,向它提供 3 个实参
  Student(1003,18,72)        //调用第 3 个元素的构造函数,向它提供 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 3.6** 计算和输出 3 个立方体的体积。

本例使用对象数组。

**编写程序:**

```
#include<iostream>
using namespace std;
class Box
  {public:
     Box(int h=10,int w=12,int len=15) :hetght(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
      Return 0;
    };
    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：**

```
volume of a[0] is 1800
volume of a[1] is 5400
volume of a[2] is 8320
```

请读者自己分析程序，了解对象数组中各元素的数据成员的值以及程序执行过程。

## 3.3　对象指针

在 C 语言中已学习过变量的指针，也学习过结构体指针。在此基础上再理解有关对象的指针就很容易了。指针不仅可以指向普通变量，也可以指向对象。

### 3.3.1　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，以存放其数据成员的值。一个**对象存储空间的起始地址就是对象的指针。可以定义一个指针变量，用来存放对象的地址，这就是指向对象的指针变量**。如果有一个类：

```
class Time
  {public:
     int hour;
     int minute;
     int sec;
     void get_time();     //在类中声明成员函数
  };
void Time::get_time()    //在类外定义成员函数
  {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;**

在上面的基础上,可以通过对象指针 pt 来访问对象和对象的公有成员。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
```

上面第2,3两行的作用是等价的,第4,5两行也是等价的。

## 3.3.2 指向对象成员的指针

对象有地址,存放对象的起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,使 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**。重温指向**普通函数**的指针变量的定义方法:

**类型名(*指针变量名)(参数表列);**

如