

# 2023

# 面向对象程序设计

## 第三讲：类与对象

李际军 lijijun@cs.zju.edu.cn



## 学习目标

1. 理解类的概念，掌握类的定义方法
2. 理解对象与类的关系，掌握对象的创建和使用方法
3. 掌握构造函数、析构函数的概念和使用方法
4. 掌握拷贝构造函数的使用方法
5. 掌握对象数组和对象指针的特点和使用方法
6. 掌握函数调用中参数的传递方式
7. 理解类的组合的特点

# 目录 / Contents



**3.1**

类和对象的概念

**3.2**

类的定义

**3.3**

对象的创建与使用

**3.4**

构造函数

**3.5**

析构函数

**3.6**

构造函数和析构函数的调用顺序

**3.7**

对象数组与对象指针

**3.8**

向函数传递对象

**3.9**

对象的赋值和复制

**3.10**

对象的组合

**3.11**

程序实例



- 在面向对象程序设计中，程序员不需要考虑数据结构和操作函数，只需要考虑对象就行了。
- 比如一个人身高 180 ， 体重 80kg ， 白皮肤，大鼻子等，这些是构成他的主要数据，也是他与别人区别的特征，但现在我们不考虑这些数据和特征，只需要把他看作自然界的一个实体，考虑他是一个什么样的人以及能够做什么即可。
- 类和对象体现了**抽象性**和**封装性**两个面向对象特征。

# 3.0

## 结构与类

- 结构的扩充
  - 1 结构是 C 语言中的一种自定义的数据类型；
  - 2 C++ 对结构进行了扩充：在结构中不仅有不同类型的数据，还可以有函数和访问控制；
  - 3 结构中的数据和函数都是结构的成员，分别称为数据成员和成员函数；
  - 4 扩充后的结构就是类。

- 定义一个复数的结构

```
struct complex
{
    double real;
    double imag;

    public: // 可以省略
        void init(double r,double i)
        { real=r; imag=i; }

        double realcomplex()
        { return real;}
} ;
```

- 结构 (类) 中成员分两类
  - **数据成员**: 状态, 特征, 组成成员,
  - **成员函数**: 修改 / 访问属性, 执行命令等
- C++ 中, 结构成员有两类: 私有 ( `private` ) 成员和公有 ( `public` ) 成员;
- C++ 规定, 在缺省的情况下, 结构中的成员 ( 数据成员、成员函数 ) 都是公有的 ( `public` ), 谁都可以访问这些成员。

- 结构的使用

```
#include <iostream.h>
void main()
{
    complex A;
    A.real=0; // real 是公有的□
    A.init(1.1,2.2);
    cout<<"real of A is"<<A.realcomplex()<<endl;
} ;
```



- 结构中的私有成员只能为该结构的其他成员所访问，其他不行。
- C++ 规定：默认情况下，结构的成员是公有的，谁都可以访问。
- 封装规定：数据成员一般为私有。
- 而上例中 处的语句违反了封装原理。
- C++ 通过引入类解决这一问题：把 struct 变成 class，结构即成为类。

# 3.1

## 类和对象的概念

- 类是对一组具有共同属性特征和行为特征的实体（对象）的抽象，它将相关数据及对这些数据的操作组合在一起。
- 在面向对象程序设计中，程序模块是由类构成的。类是对逻辑上相关的函数与数据的封装，它是对问题的抽象的描述。
- 因此集成程度更高，适合大型程序的开发。

### 3.1.1 类的概念

类（class）是面向对象系统中**最基本的组成元素**，是一种**自定义数据类型**。在C++中，类是一些具有相同属性和行为的对象的抽象。

### 3.1.2 对象的概念

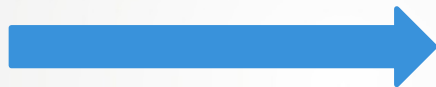
对象是某个特定类所描述的实例。现实世界中的任何一种事物都可以看成一个对象（Object），即万物皆对象。

# 3.2

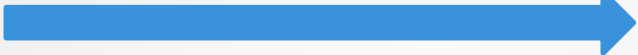


## 类的定义

- 类是实现数据抽象的工具，是对某一类具有相同属性和行为特征对象的抽象，它不仅定义了数据的类型，同时也定义了对数据的操作，是实现面向对象程序设计的基础。
- 面向对象的程序主要由类和对象组成。
- 类的定义一般分为说明和实现部分：
  - ✓ 说明部分包括数据成员的说明和成员函数的说明。数据成员定义该类对象的属性，是不同类型的多个数据。成员函数定义了类对象的行为特征，由多个函数原型声明构成，一般放在 .h 。
  - ✓ 实现部分根据所要完成的具体功能，给出所有成员函数的定义，一般为 .cpp.

```
class 类名 {  
    数据成员  
    .....  
    数据成员  
    成员函数  
    .....  
    成员函数  
};
```



封装了数据和行为

```
class 类名 {  
    private:  私有访问权限  
    数据成员或成员函数  
    protected:  保护访问权限  
    数据成员或成员函数  
    public:  公有访问权限  
    数据成员或成员函数  
};
```

公有成员定义类的外部接口

说明:

- (1) `class` 是声明类的关键字, `class` 后跟类名。类名的首字符通常采用大写字母。
- (2) 类的成员包括数据成员和成员函数两类。
- (3) 类声明中的 `private`、`protected` 和 `public` 关键字称为访问权限符, 它规定了类中成员的访问属性。
- (4) 在 C++ 中, 由于类是一种数据类型, 系统不会为其分配存储空间, 所以不能在类声明中给数据成员赋初值。
- (5) 类声明完成后一定要以 “;” 结束。
- (6) 类是抽象的名词, 而不是具体的某个个体, 因此无法对他进行赋值操作, 正如我们无法对 `int` 这个数据类型进行赋值一样。如 `:int=5`

## ■ 注意事项:

- (1) 访问权限在类体内允许多次出现, 出现的顺序无关紧要。
- (2) 当成员函数的函数体内容较简短时直接在类体内定义。若在类外定义成员函数, 一定要加上作用域限定符 “ :: ” 。
- (3) 在类体中不允许对所定义的数据成员进行初始化。类的定义只是一种设计说明, 不分配存储空间。
- (4) 类中数据成员的类型可以是任意的, 包括基本数据类型、数组、指针和引用等, 还可以是对象。
- (5) 在类体内定义的成员函数属于内联函数。
- (6) 在定义类时, 最后的分号不可省略。



**【例 3-1】** 声明一个学生类

分析：每个学生都有学号、姓名和性别；对于学生的基本操作有输入、输出信息等。

```
class Student                // 声明类
{
private:                    // 访问权限：私有成员
    char studentNo[10];      // 属性，数据成员，表示学号
    char studentName[20];    // 属性，数据成员，表示姓名
    char studentSex[6];      // 属性，数据成员，表示性别
public:                    // 访问权限：公有成员
    Student();              // 行为，成员函数的原型声明，表示构造函数
    void input();
    void print();
};                          // 类声明结束
```

### 3.2.2 类的定义格式

对于 C++，类中共有两类成员：

- 1 ) 描述对象属性的数据成员
- 2 ) 实现对象行为的成员函数

### 3.2.3 类成员访问控制权限

C++ 中类成员访问的控制，是通过设置成员的访问控制权限实现的。

有三种访问属性：`public`（公有类型）、`private`（私有类型）和 `protected`（保护类型）。

## 1、public（公有类型）

public 声明成员为公有成员。定义了类的外部接口，对外是完全公开的，即提供了外部对象与类对象相互之间交互的接口。在类外只能访问该类的公有成员。

公有成员通常都是成员函数。

公有成员的访问权限

【例 3-2】

```
class Human
{
public:    // 声明类的公有成员
    int stature;
    int weight;
    void GetStature()
    { cout<<"Your stature is:"<<stature<<endl; }
    void GetWeight()
    { cout<<"Your weight is:"<<weight<<endl;}
};
```

类内：可以任意访问  
公有成员

### 公有成员的访问权限

```
int main()
{
    Human Tom;           // 定义类的对象
    Tom.stature=185;      // 通过对象访问类的公有数据成员
    Tom.weight=90;        // 通过对象访问类的公有数据成员
    Tom.GetStature();     // 通过对象访问类的公有成员函数
    Tom.GetWeight();      // 通过对象访问类的公有成员函数
    return 0;
}
```

类外：通过对象任意访问公有成员

## 2、private（私有类型）

`private` 声明成员为私有成员。具有这个访问控制级别的成员对类外是完全保密的，只能被它本类中的成员函数或该类的友元函数访问。其他来自类外部任何访问都是非法的。

这样私有成员就完全隐蔽在类中，保护了数据的安全性。



## 私有成员的访问权限

类内：可以任意访问私有成员

```
class Human{  
private:           // 声明类的私有数据成员  
    int stature;  
    int weight;  
public:           // 声明类的公有成员函数  
    void SetStature(int s) {Stature=s; }    // 类的成员函数访问类的私有数据成员  
  
    void GetStature() {cout<<"Your stature is:"<<stature<<endl; }  
    // 类的成员函数访问类的私有数据成员  
    void SetWeight(int w) {Weight=w; } // 类的成员函数访问类的私有数据成员  
    void GetWeight() {cout<<"Your weight is:"<<weight<<endl; }  
};
```

## 私有成员的访问权限

```
int main()
{
    Human Tom;           // 定义类的对象
    //Tom.stature=185; // 错误，不能通过对象访问类的私有数据成员
    // Tom.weight=90; // 错误，不能通过对象访问类的私有数据成员
    Tom.SetStature(185); // 通过对象访问类的公有成员函数给 stature 赋值
    Tom.SetWeight(90);   // 通过对象访问类的公有成员函数给 Weight 赋值
    Tom.GetStature();    // 通过对象访问类的公有成员函数
    Tom.GetWeight();     // 通过对象访问类的公有成员函数
    return 0;
}
```

类外：不能通过对象  
访问类的私有成员

### 3、protected（保护类型）

protected 声明成员为保护成员。具有这个访问控制级别的成员，外界是无法直接访问的。它只能被它所在类及从该类派生的子类的成员函数及友元函数访问。

保护成员和私有成员的性质相似，其差别在继承过程中对产生的新类影响不同。这个问题将在后续章中介绍。

## 保护成员的访问权限

```
class Human
{
protected:          // 声明类的私有数据成员
    int stature;
    int weight;
public:              // 声明类的公有成员函数
    void SetStature(int s) {Stature=s; }    // 类的成员函数访问类的保护数据
    void GetStature() {cout<<"Your stature is:"<<stature<<endl; }
    void SetWeight(int w) {Weight=w; }
    void GetWeight() {cout<<"Your weight is:"<<weight<<endl; }
};
```

类内：可以任意  
访问保护成员



### 保护成员的访问权限

```
int main()
{
    Human Tom; // 定义类的对象
    //Tom.stature=185; // 错误，不能通过对象访问类的保护数据成员
    // Tom.weight=90; // 错误，不能通过对象访问类的保护数据成员
    Tom.SetStature(185);
    Tom.SetWeight(90);
    Tom.GetStature();
    Tom.GetWeight();
    return 0;
}
```

类外：不能通过对象  
访问类的保护成员

public 、 protected 和 private 三种类成员的可访问性

访问限定符	自身的类成员是否可访问	子类的类成员是否可访问	自身的类对象是否可访问
public	✓	✓	✓
protected	✓	✓	×
private	✓	×	×

### 3.2.4 成员函数的实现方式

- 类的成员函数也是函数的一种，它与一般函数的区别是：它属于一个特定的类，并且它必须被指定为 `private`、`public` 或 `protected` 三种访问权限中的一种。
- 在使用类的成员函数时，要注意它的访问权限（它能否被访问），以及它的作用域（类函数能在什么范围内被访问）

类的成员函数的定义方式有两种：

第一种方式是在类中进行函数原型说明，而函数体则在类外进行定义。

采用这种方式定义类函数时，必须用作用域符“::”表明该函数所属的类。

返回类型 类名 :: 函数名（参数列表）

```
{  
    // 函数体  
}
```



【例 3-3】定义时钟类。

```
#include <stdafx.h>
#include <iostream>
using namespace std;
class Clock
{
private:
    int hour, minute, second;
public:
    void setTime(int newH, int newM, int newS);
    // 函数原型说明
    void showTime();
    // 函数原型说明
};
```

```
void Clock::setTime(int newH, int newM, int newS) // 定义成员函数
{
    hour=newH;
    minute=newM;
    second=newS
}
```

说明:

- (1) 在定义成员函数时, 对函数所带的参数, 既要说明其类型, 也要指出参数名;
- (2) 在定义成员函数时, 其返回值必须与函数原型声明中的返回类型相同。

类的成员函数的**定义方式**:

第二种方式是在类内直接进行定义。这种方式一般用在代码比较少的成员函数。被默认为内联函数。

### 3.2.5 成员函数设置为内联函数

一般的函数使用过程中，需要进行函数调用，由于在调用函数时，需要保存现场和返回地址、进行参数传递，再转到子函数的代码起始地址去执行。子函数执行完后，又要取出以前保存的返回地址和现场状态，再继续执行。

内联函数与一般函数的区别在于它不是在调用时发生控制转移，而是在编译时将函数体嵌入到每一个调用处。这样就节省了参数传递、控制转移等开销。



内联函数的定义格式为：

```
inline 返回值类型  函数名（形参列表）  
{  
  
    // 函数体  
  
}
```

说明：

- （1）内联函数体内不能含有循环语句和 switch 语句；
- （2）内联函数的定义必须出现在第一次被调用之前；
- （3）对内联函数不能进行异常接口声明。



内联函数实际上是一种以空间换时间的方案，其缺点是加大了空间方面的开销。它通常用在结构简单、语句少、使用多的情况下。

C++ 默认在类内给出函数体定义的成员函数为内联函数。

【例 3-4】内联函数应用举例，计算正方形的面积及周长。

```
class Square{
private:
    double length;
public:
    Square(double x); // 构造函数
    void area() // 函数体在类内定义，默认为内联函数
    {
        cout<<" 正方形的面积为 : "<<length*length<<endl;
    }
    inline void Perimeter(); // 内联函数声明
};
Square::Square(double x) {
```



```
void Square::Perimeter () // 内联函数定义
{
    cout<<" 正方形的周长为 : "<<4*length<<endl;
}
int main()
{
    Square ss(2.0);
    ss.area ();
    ss.Perimeter ();
}
```





### 说明:

( 1 ) 内联函数代码不宜过长，一般是小于 10 行代码的小程序，并且不能含有复杂的分支 ( switch ) 和循环语句。

( 2 ) 在类内定义的成员函数默认为内联函数。

( 3 ) 在类外给出函数体定义的成员函数，若要定义为内联函数，必须加上关键字 inline 。

( 4 ) 递归调用的函数不能定义为内联函数。

### 3.2.6 成员函数重载

函数重载是指两个以上的函数，具有相同的函数名，可以对应着多个函数的实现。每种实现对应着一个函数体，但是形参的个数或者类型不同，编译器根据实参和形参的类型及个数的最佳匹配，自动确定调用哪一个函数。

【例 3-5】函数重载举例。

创建一个类，在类中定义三个名为 `subtract` 的重载成员函数，分别实现两个整数相减、两个实数相减和两个复数相减的功能

```
struct complex  
{  
    double real;  
    double imag;  
};
```

```
class Overloaded
{
public:
    int subtract(int x, int y);
    double subtract(double x, double y);    // 函数重载
    complex subtract(complex x, complex y); // 函数重载
};
```

```
int Overloaded::subtract (int x, int y)
{
    return x-y;
}
double Overloaded::subtract (double x, double y)
{
    return x-y;
}
complex Overloaded::subtract (complex x, complex y)
{
    complex c;
    c.real=x.real -y.real ;
    c.imag =x.imag -y.imag ;
    return c;
}
```

```
int main()
{
    int m,n;
    double x,y;
    complex a,b,c;
    Overloaded ol;
    m =32; n =23; x =31.1; y =22.2;
    a.real =12.3; a.imag =10.2; b.real =23.5; b.imag =1.2;
    cout<<m<<"-"<<n<<"="<<ol.subtract (m,n)<<endl;
    cout<<x<<"-"<<y<<"="<<ol.subtract (x,y)<<endl;
    c=ol.subtract (a,b);
    cout<<" " <<a.real<<"+"<<a.imag
        <<" " <<"-"<<" " <<b.real<<"+"<<b.imag <<" "
        <<"="<<" " <<c.real<<"+"<<c.imag<<" " <<endl;
    return 0;
}
```



# 3.3

## 对象的创建与使用

在 C++ 中，声明了类，只是定义了一种新的数据类型，只有当定义了类的对象后，才是生成了这种数据类型的特定实体（实例）。对象是类的实际变量，创建一个对象称为实例化一个对象或创建一个对象实例。

类是对象的抽象，而对象是类的实例。



### 3.3.1 对象的定义

1、先声明类类型，然后在使用时再定义对象

定义格式与一般变量  
定义格式相同：

类名 对象名列表；

### 对象的定义

2、在声明类的同时，直接定义对象

```
class Student
{
    .....
} stud1, stud2;
```

3、不出现类名，直接定义对象

```
class
{
    .....
} stud1, stud2;
```





说明:

( 1 ) 必须**先定义类**，然后**再定义类的对象**。多个对象之间用**逗号**分隔。

( 2 ) 声明了一个类就是声明了一种新的**数据类型**，它本身不能接收和存储具体的值，只有定义了类的**对象**后，系统才为其对象分配存储空间。

( 3 ) 在声明类的同时定义类对象是一种**全局对象**，它的生存期一直到整个程序运行结束。



### 3.3.2 对象成员的访问

#### 1. 通过对象名和成员运算符访问对象的成员

使用这种方式访问对象的数据成员的一般形式为：

对象名 . 数据成员

使用这种方式访问对象的成员函数的一般形式为：

对象名 . 成员函数名 ( 实参列表 )

*注意：对象只能访问其的公有 ( public ) 成员*

【例 3-6】建立图书档案类，通过键盘输入每种图书的相关信息，并按价格从低到高的顺序排序输出。

```
class Book                                //Book. h
{
    public:
        char title[20], auther[10], publish[30];
                                   // 书名、作者、出版社
        float price; // 价格
        void input();
        void output();
};
```

//Book. cpp

```
#include <stdafx.h>
#include <iostream>
using namespace std;
#include "Book.h"
```

```
void Book::input ()
{
    cin>>title>>auther>>publish>>price;
}
void Book::output ()
{
    cout<<title<< "    "<<auther<<" "<<publish<<"
"<<price<<endl;
}
```

```
int main()
{
    int i, j;
    Book bk[10], temp;
    cout << " 请输入书名、作者、出版社和价格 " << endl;
    for (i=0; i<10; i++)
        bk[i].input ();
    for (i=0; i<10; i++)
        for (j=i+1; j<10; j++)
        {
            if (bk[i].price > bk[j].price )
            { temp = bk[i] ; bk[i] = bk[j] ;
              bk[j] = temp ;}
        }
}
```



```
    cout << " 输出结果 " << endl;
    cout << " 书名 作者 出版社 价格 " << endl;
    for (i=0; i<10; i++)
        bk[i].output ();
    return 0;
}
```



【例 3-7】定义 Student 类的对象，实现对当前对象信息的输入、设置和输出。

```
#include <iostream>
#include <string>
#include <iomanip>
using namespace std;
// 类声明部分
class Student
{
public:           // 公有成员函数
    void Input();
    void Set(string,string,char);
    void Show();
private:        // 私有数据成员
    string num;      // 学号
    string name;     // 姓名
    char sex;        // 性别
};
```

// 类的实现部分

void Student::Input()

```
{  cout<<"num=";   cin>>num;
    cout<<"name=";  cin>>name;
    cout<<"sex=";   cin>>sex;
}
```

void Student::Set(string num1,string name1,char sex1)

```
{  num=num1;      name=name1;      sex=sex1;  }
```

void Student::Show()

```
{
    cout<<setw(8)<<"num"<<setw(8)<<"name"<<setw(8)<<"sex"<<endl;
    cout<<setw(8)<<num<<setw(8)<<name<<setw(8);
    if(sex=='f' || sex=='F')
        cout<<"female"<<endl;
    else
        cout<<"male"<<endl;
}
```





```
int main()
{
    Student stu1,stu2;
    cout<<"input object stu1:"<<endl;
    stu1.Input();
    cout<<"output object stu1:"<<endl;
    stu1.Show();
    cout<<"set object stu2!"<<endl;
    stu2.Set("s001","Anny",'m');
    cout<<"output object stu2:"<<endl;
    stu2.Show();
    return 0;
}
```

运行结果：

```
input object stu1:
num=x001
name=Marry
sex=f
output object stu1:
num name sex
x001 Marry female
set object stu2!
output object stu2:
num name sex
s001 Anny male
```

## 2. 通过指向对象的指针访问对象中的成员

用这种方式访问对象的数据成员的一般形式为:

数据成员 指向对象的指针 ->

使用这种方式访问对象的成员函数的一般形式为:

## 指向对象的指针 -> 成员函数

【例 3-8】改写【例 3-7】中的主函数，通过指向对象的指针访问对象的数据成员和成员函数。

```
int main()
{
    int i, j;
    Book bk[10], *p1, *p2, temp;
    cout << "请输入书名、作者、出版社和价格 " << endl;
    for (i=0; i<10; i++)
    {
        p1=&bk[i];           //p1 指向 bk[i]
        p1->input ();
        // 通过指向对象的指针访问对象的成员函数
    }
}
```



```
for (i=0; i<10; i++)
{
    p1=&bk[i];
    for (j=i+1; j<10; j++)
    {
        p2=&bk[j];
        if (p1->price > p2->price )
            // 通过指向对象的指针访问对象的数据成员
        {
            temp =*p1 ;
            *p1=*p2 ;
            *p2=temp ;
        }
    }
}
```



```
    cout <<" 输出结果: " <<endl;
    cout <<" 书名 作者 出版社 价格 " <<endl;
    for (i=0; i<10; i++)
    {
        p1=&bk[i];
        p1->output ();
    }
    return 0;
}
```



【例 3-9】定义一个平面上的点类 Point，实现设置、移动、获取坐标和输出坐标功能。

```
#include <iostream>
using namespace std;
// 类的声明部分
class Point
{
public:
    void MoveTo(int,int);           // 设置点的坐标
    void Move(int,int);             // 移动点的坐标
    void MoveH(int);                // 移动点的横坐标
    void MoveV(int);                // 移动点的纵坐标
    int GetX();                     // 获取点的横坐标值
    int GetY();                     // 获取点的纵坐标值
    void Show();                    // 显示点的坐标
private:
    int x,y;                        // 横坐标、纵坐标
};
```



// 类的实现部分

```
void Point::MoveTo(int nx,int ny)// 设置点的坐标
{   x=nx;y=ny;   }
void Point::Move(int hx,int hy)           // 移动点的坐标
{   x+=hx;y+=hy;   }
void Point::MoveH(int hx)// 移动点的横坐标
{   x+=hx;           }
void Point::MoveV(int hy)// 移动点的纵坐标
{   y+=hy;           }
int Point::GetX()// 获取点的横坐标值
{   return x;        }
int Point::GetY()// 获取点的纵坐标值
{   return y;        }
void Point::Show()// 显示点的坐标
{   cout<<"( "<<x<<" , "<<y<<" )"<<endl;   }
```



```
int main()
{
    cout<<"***** Point p1 *****"<<endl;
    Point p1;
    p1.MoveTo(100,200);
    p1.Show();
    cout<<"***** Point *p2 *****"<<endl;
    Point *p2=&p1;
    p2->MoveH(120);
    p2->MoveV(220);
    p2->Show();
    cout<<"***** Point &p3 *****"<<endl;
    Point &p3=p1;
    p3.Move(50,50);
    cout<<"( "<<p3.GetX()<<" , "<<p3.GetY()<<" )"<<endl;
    return 0;
}
```

运行结果:

```
***** Point p1 *****
( 100 , 200 )
***** Point *p2 *****
( 220 , 420 )
***** Point &p3
*****
( 270 , 470 )
```



### 3. 通过对象的引用访问对象中的成员

对象的引用变量与该对象**共占**同一段存储单元，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。

定义一个对象的引用变量的方法为：

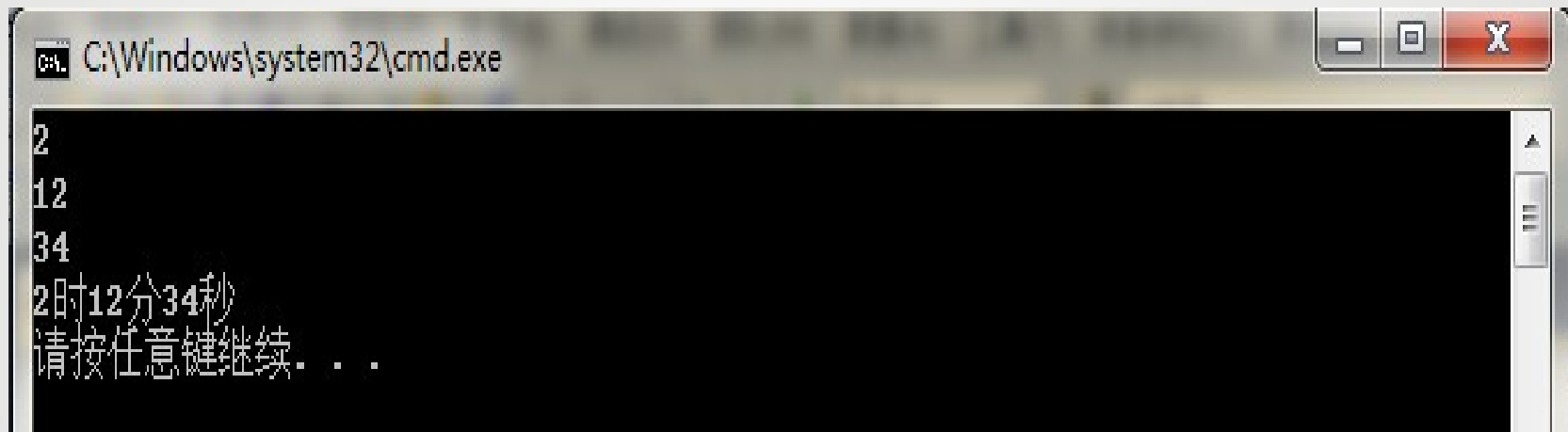
**& 引用变量名 = 对象名 ;**



**【例 3-10】** 通过对象的引用变量来访问对象的数据成员和成员函数。

```
class Time{
public :
    int hour, minute, second;
    void showTime();
};

int main()
{
    Time t1;
    t1.hour=2;
    t1.minute =12;
    t1.second =34;
    Time &t2=t1;           // 定义 t1 对象的引用变量 t2
    cout<<t2.hour<<endl;   // 通过引用变量访问对象的数据成员
    cout<<t2.minute<<endl;
    cout<<t2.second<<endl;
    t2.showTime();         // 通过引用变量访问对象的成员函数
}
```



```
C:\Windows\system32\cmd.exe
2
12
34
2时12分34秒
请按任意键继续...
```



# 3.4

## 构造函数

- 构造函数（Constructor）是一种特殊的成员函数，它是用来完成在声明对象的同时，对对象中的数据成员进行初始化。

### 3.4.1 构造函数的定义和功能

构造函数的定义如下：

类名（形参列表）；

构造函数可以在类内也可在类外定义。在类外定义构造函数的形式如下：

```
类名 :: 类名（形参列表）  
{  
    // 函数体 ;  
}
```

说明:

- (1) 构造函数的名称必须与类名**相同**。
- (2) 构造函数**没有**返回值类型，也不能指定为 void。
- (3) 构造函数可以有**任意**个任意类型的参数。
- (4) 如果没有显式的定义构造函数，系统会自动生成一个**默认**的构造函数。  
这个构造函数不含有参数，也不对数据成员进行初始化，只负责为对象分配存储空间。
- (5) 如果显式的为类定义了构造函数，系统将**不再**为类提供默认构造函数。
- (6) 定义对象时，系统会**自动**调用构造函数。
- (7) 构造函数可以**重载**。
- (8) 构造函数一般被定义为**公有**访问权限。

【例 3-11】举例说明构造函数的使用。

```
class Date
{
private:
    int year;
    int month;
    int day;
public:
    Date(int y, int m, int d);           // 声明构造函数
    void Output();
};
```

```
Date::Date(int y, int m, int d)           // 定义构造函数
{
    year=y;
    month=m;
    day=d;
}
void Date::Output()
{
    cout<<year<<"/"<<month<<"/"<<day<<endl;
}
void main()
{
    Date  today(2012, 10, 10);
    today.Output();
}
```



### 3.4.2 默认构造函数

如果类没有定义构造函数，系统会自动生成一个默认的构造函数。这个构造函数不含有参数，也不会对数据成员进行初始化。默认构造函数的形式如下：

构造函数名 ( ) {}

此时要特别**注意**，数据成员的值是随机的。程序运行时容易出错。

### 3.4.3 无参构造函数

```
class Point
{
private:
    int x;
    int y;
public:
    Point();
};
Point::Point()
{
    x=1;
    y=2;
}
```

### 3.4.4 构造函数的重载

一个类可以定义**多个**构造函数，这些构造函数具有**相同**的名字，但参数的**个数**或参数的**类型**存在差别，这称为构造函数的**重载**。

【例 3-12】构造函数重载举例。

```
class Date
{
private:
    int year;
    int month;
    int day;
public:
    Date(); // 无参的构造函数
    Date(int y, int m, int d); // 含参的构造函数
    void Output();
};
```

```
Date::Date()
{
    year=2012;
    month=10;
    day=11;
}
Date::Date(int y,int m,int d)
{
    year=y;
    month=m;
    day=d;
}
```

```
void Date::Output()
{
    cout<<year<<"/"<<month<<"/"<<day<<endl;
}
void main()
{
    Date today(2012,10,10);
    Date tomorrow;
    today.Output();
    tomorrow.Output();
}
```

### 3.4.5 带默认参数的构造函数

带默认参数的构造函数的原型定义形式如下：

类名（函数名）（参数 1= 默认值，参数 2= 默认值，…）；

所谓的默认参数即为该参数设置一个默认的值，可以为全部或者部分参数设置默认值。

【例 3-13】带默认参数的构造函数应用举例。

```
class Date
{
private:
    int year;
    int month;
    int day;
public:
    Date(int y=2012, int m=1, int d=1);    // 定义带默认参数的构造函数
    void Output();
};
```



```
Date::Date(int y,int m,int d)
{
    year=y; month=m; day=d;
}
void Date::Output()
{
    count<<year<<"/"<<month<<"/"<<day<<endl;
}
void main()
{
    Date today(2012,10,10);    // 使用给定值初始化对象
    Date longago;              // 使用默认值初始化对象
    today.Output();
    longago.Output();
}
```

说明：

- （1）默认参数只能在原型声明中指定，**不能**在构造函数的定义中指定。
- （2）在构造函数原型声明中，所有给默认值的参数都必须在不给默认值的参数的**右面**。
- （3）在对象定义时，若**省略**构造函数的某个参数的值，则其右面所有参数的值都必须省略，而采用默认值。
- （4）构造函数带有默认参数时，在定义对象时要**注意避免二义性**。例如：  
`Date(int y=2012,int m=1,int d=1);`  
`Date();`

### 3.4.6 构造函数与初始化列表

构造函数也可以采用**构造初始化列表**的方式对**数据成员**进行**初始化**。例如，可以把【例 3-12】中的构造函数 `Date(int y, int m, int d)` 的定义改写为：

```
Date::Date(int y, int m, int d):year(y), month(m), day(d)
{
}
```

它与【例 3-12】的定义等价

```
#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



在例 3-15 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。

将【例 3-16】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。

```
#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



# 3.5

## 析构函数

析构函数 (Destructor) 与构造函数相反，当对象的生命期结束（删除对象）时，就会自动调用析构函数清除它所占用的内存空间。

系统执行析构函数的**四种**情况：

（1）在一个函数中定义了一个对象，当这个函数被调用结束时，该对象应该释放，在对象释放前会**自动**执行析构函数。

（2）具有 static 属性的对象（静态对象，将在第四章介绍）在函数调用结束时该对象并不释放，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，其生命期将结束，这时才调用析构函数。

（3）**全局对象**，在 main 函数结束时，其生命期将结束，这时才调用其的析构函数。

（4）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，调用该对象的析构函数。

析构函数的定义格式为：

~ 类名 ();

说明：

- ( 1 ) 析构函数名是由 “ ~ ” 加类名组成，区别于构造函数。
- ( 2 ) 析构函数没有参数、没有返回值，而且不能重载。
- ( 3 ) 一个类有且仅有一个析构函数，且应为 public 。
- ( 4 ) 在对象的生命期结束前，由系统自动调用析构函数。
- ( 5 ) 如果没有定义析构函数，系统会自动生成一个默认的析构函数，这个析构函数不做任何事情。

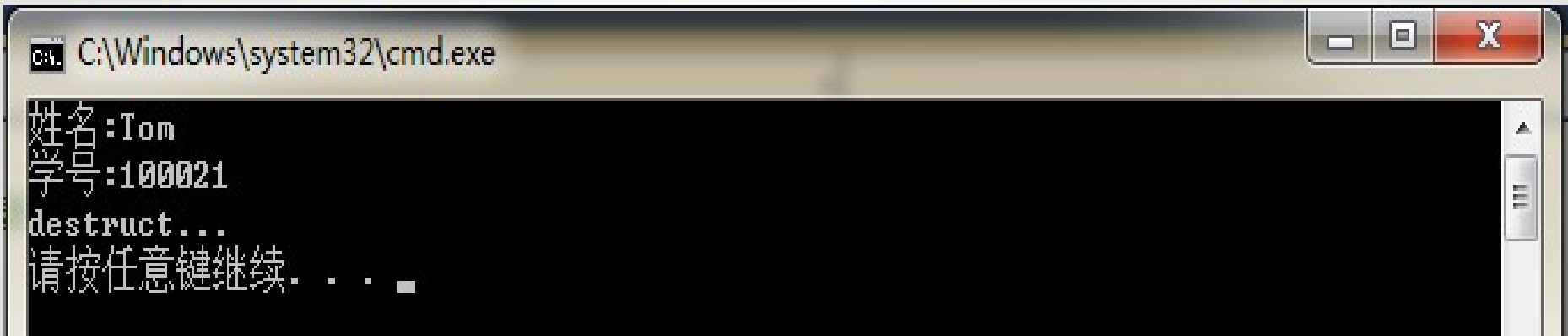


**【例 3-18】析构函数应用举例**

```
#include<string>
#include<iostream>
using namespace std;
class Student
{
private:
    string name;
    int number;
public:
    Student(string na, int nu);
    ~Student();           // 析构函数原型声明
    void Output();
};
```

```
Student::Student(string na, int nu)
{
    name=na;
    number=nu;
}
Student::~~Student()    // 析构函数定义
{
    cout<<"destruct..."<<endl;
}
```

```
void Student::Output()
{
    cout<<" 姓名 "<<":"<<name<<endl;
    cout<<" 学号 "<<":"<<number<<endl;
}
int main()
{
    Student S1("Tom",100021);
    S1.Output();
    return 0;
}
```

A screenshot of a Windows command prompt window. The title bar shows the path 'C:\Windows\system32\cmd.exe'. The command prompt has a black background with white text. The output of a program is displayed: '姓名:Tom', '学号:100021', 'destruct...', and '请按任意键继续. . .'.

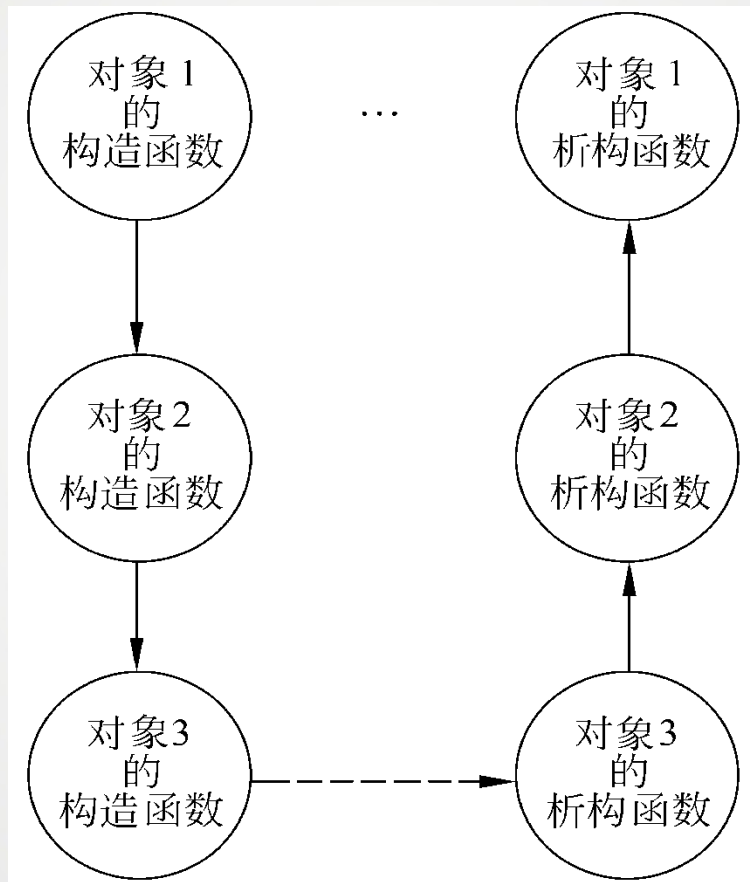
```
C:\Windows\system32\cmd.exe  
姓名:Tom  
学号:100021  
destruct...  
请按任意键继续. . .
```



# 3.6

## 构造函数和析构函数的调用顺序

- 当创建一个对象，其生命周期开始时调用构造函数，当删除一个对象，其生命周期结束时调用析构函数。即二者何时调用与对象的生命周期有关。
- 如果程序中定义多个对象，那么创建和删除这些对象时，调用构造函数和析构函数有一定的顺序。
- 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，也就是最先得调用的构造函数，其对应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。



根据对象的生存期，不同对象调用构造函数和析构函数的时间：

（1）**全局对象**（在函数之外定义）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），调用其析构函数。

（2）**局部对象**（在函数中定义的对象）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、对象释放前先调用析构函数。

（3）如果在函数中定义了**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被释放，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。

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。



### [例 3-19] 构造函数与析构函数的执行顺序——Point 类的多个对象的创建

```
class Point{                                //point.h
private:
```

```
    int x,y;
```

```
public:
```

```
    Point(int a,int b);
```

```
    ~Point();
```

```
};
```

```
#include <stdafx.h>
```

```
#include <iostream>
```

```
//point.cpp
```

```
using namespace std;
```

```
#include "point.h "
```

```
int main()
```

```
{
```

```
    Point p1(1,2),p2(3,5);
```

```
    return 0;
```

```
}
```

```
#include <stdafx.h>
#include <iostream>
using namespace std;
#include "point.h "
int main()
{
    Point p1(1,2),p2(3,5);
    return 0;
}
```

//point.cpp

```
Point::Point(int a,int b) // 定义构造函数
{
    cout<<"constructor....."<<endl;
    x=a;
    y=b;
    cout<<"("<<x<<" "<<y<<"")"<<endl;
}
Point::~~Point()           // 定义析构函数
{
    cout<<"destructor....."<<endl;
    cout (<<x<<" "<<y<<"") <<endl;
}
```

【例 3-20】构造函数与析构函数执行顺序

```
class Time
{
private:
    int hour;
    int minute;
    int second;
public:
    Time(int h, int m, int s);
    ~Time();
};
```

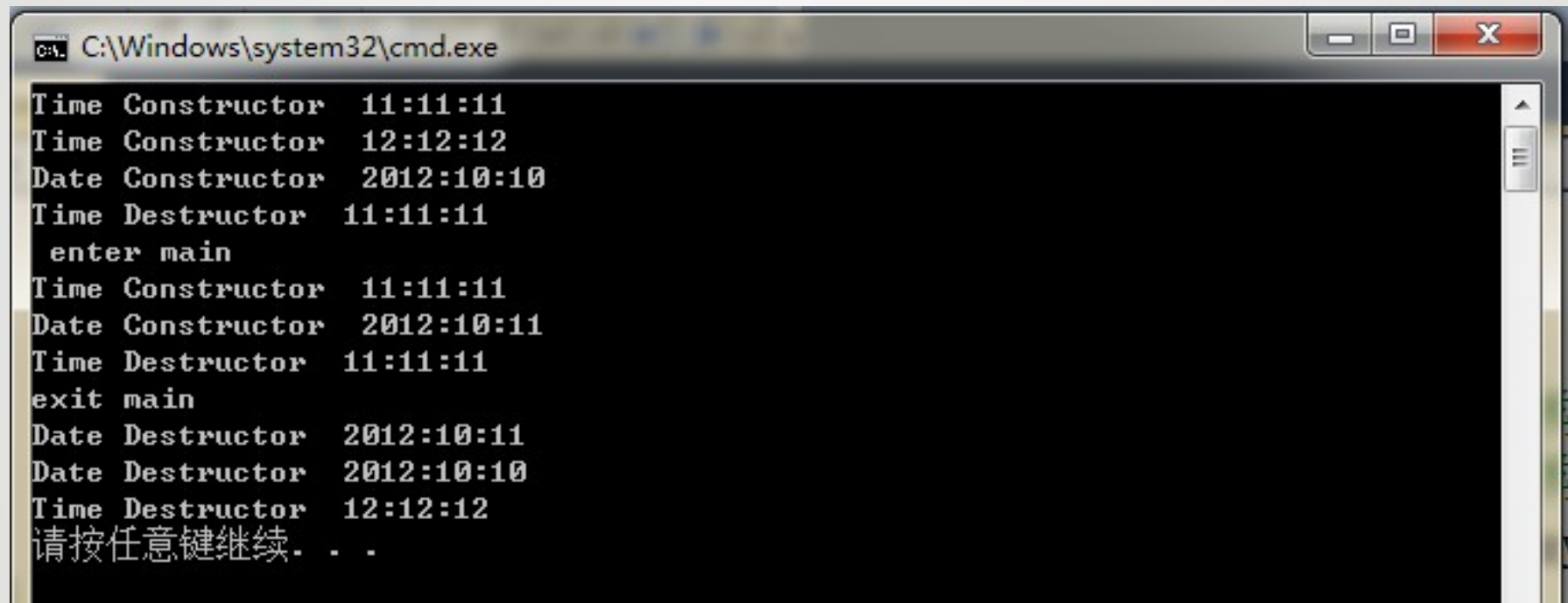
```
Time::Time(int h, int m, int s)
{
    hour=h;
    minute=m;
    second=s;
    cout<<"Time Constructor "<<hour<<": "<<minute<<": "<<second<<endl;
}
Time::~~Time ()
{
    cout<<"Time Destructor"<<hour<<": "<<minute<<": "<<second<<endl;
}
```

```
class Date{
private:
    int year;
    int month;
    int day;
public:
    Date(int y,int m,int d); // 声明构造函数
    ~Date();                // 声明析构函数
}yesterday(2012,10,10);    // 定义全局对象
```

```
Date::Date(int y,int m,int d)    // 定义构造函数
{
    year=y;
    month=m;
    day=d;
    (1) Time time(11,11,11);
    // 在类 Date 定义的构造函数中定义类 Time 的对象 (局部)
    (2) static Time time1(12,12,12);
    // 在类 Date 定义的构造函数中定义类 Time 的静态对象 (局部)
    (3) cout<<"Date Constructor
    "<<year<<":"<<month<<":"<<day<<endl;
}
```

```
Date::~~Date ()
{
    cout<<"Date Destructor
        "<<year<<":"<<month<<":"<<day<<endl;
}
void main()
{
    ( 4 )    cout<<" enter main"<<endl;
    ( 5 )    Date  today(2012,10,11);
    ( 6 )    cout<<"exit main"<<endl;
} ( 7 )
```





```
C:\Windows\system32\cmd.exe

Time Constructor  11:11:11
Time Constructor  12:12:12
Date Constructor  2012:10:10
Time Destructor  11:11:11
enter main
Time Constructor  11:11:11
Date Constructor  2012:10:11
Time Destructor  11:11:11
exit main
Date Destructor  2012:10:11
Date Destructor  2012:10:10
Time Destructor  12:12:12
请按任意键继续. . .
```



# 3.7

## 对象数组与对象指针

### 3.7.1 对象数组

对象数组的元素是**对象**，它不仅具有数据成员，而且也具有成员函数。

定义对象数组、使用对象数组的方法与基本数据类型相似。

在执行对象数组说明语句时，系统不仅为对象数组分配**内存空间**，以存放数组中的每个对象，而且还会**自动**调用匹配的构造函数完成数组内**每个**对象的初始化工作。

声明对象数组的格式为：

类名 数组名 [ 下标表达式 ]；

在使用对象数组时，也只能引用单个数组元素，并且通过对象数组元素只能访问其的公有成员。访问对象数组元素的数据成员

数组名 [ 下标 ]. 数据成员 ；

访问对象数组元素的成员函数

数组名 [ 下标 ]. 成员函数 ( 实参列表 )；

注意：当定义一个对象数组时，系统会为数组的每一个数组元素调用一次构造函数，来初始化每一个数组元素。

【例 3-21】对象数组使用举例。

```
class Box
{
public :
    Box(int h=10, int w=12, int len=15); // 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);
}
```

```
int main( )
{
    Box a[3]={                // 定义对象数组
        Box(),                // 调用构造函数 Box , 用默认参数初始化第 1 个元素的数据成员
        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;
    cout<<"volume of a[1] is "<<a[1].volume( )<<endl;
    cout<<"volume of a[2] is "<<a[2].volume( )<<endl;
}
```

A screenshot of a Windows command prompt window. The title bar shows the path "C:\Windows\system32\cmd.exe". The command prompt has a black background with white text. It displays the following output:

```
volume of a[0] is 1800  
volume of a[1] is 5400  
volume of a[2] is 8320  
请按任意键继续. . .
```

The text "请按任意键继续. . ." is a common prompt in Chinese command prompts, meaning "Press any key to continue...".



■ 对象数组也可以像普通数组一样在定义的同时进行初始化。

■ 例如：

```
– int main(){  
–     A a[3]={A(100),A(200),A(300)};  
–     for(int i=0;i<3;i++)  
–         a[i].Display();  
–     return 0;  
– }
```

■ **注意：**将每个元素的初始值用类名加括号括起来。

### 3.7.2 对象指针

声明对象指针的格式为：

类名 \* 对象指针名 ；

用对象指针访问对象数据成员的格式为：

对象指针名 -> 数据成员 ；

用对象指针访问对象成员函数的格式为：

对象指针名 -> 成员函数（实参列表）；

同一般变量的指针一样，对象指针在使用之前**必须**先进行**初始化**。可以让它指向一个已定义的对象，也可以用 **new 运算符** 动态建立堆对象。

### 3.7.2 对象指针

### ■通过对象指针间接访问对象成员的格式:

```
( * 对象指针名) . 数据成员名;           // 访问数据成员
```

( \* 对象指针名) . 成员函数名 (参数表) ;     // 访问成员函数

对象的指针名 -> 数据成员名; // 访问数据成员

【例 3-22】对象指针应用举例。

```
#include <stdafx.h>
#include <iostream>
using namespace std;
class Square{
private:
    double length;
public:
    Square(double len);
    void Outpout();
};
Square::Square (double len):length(len)
{
}
```

```
void Square::Output()
{
    cout<<"Square Area:"<<length * length<<endl;
}

int main()
{
    Square s(2.5), *s1;
    s1=&s;
    s1->Output();
    Square *s2=new Square(3.5);
    s2->Output();
    delete s2;
    return 0;
}
```

也可以通过对象指针来访问对象数组，这时对象指针指向对象数组的首地址。

【例 3-23】改写【例 3-21】的主函数，通过对象指针引用 Box 类的对象数组。

```
int main( )
{
    Box a[3]={           // 定义对象数组！
        Box(),           // 调用构造函数 Box，用默认参数初始化第 1 个元素的数据成员
        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
    };
    Box *p=a;
    for(int i=0;i<3;i++, p++)
    {
        cout<<"volume of a["<<i<<"] is "<<p->volume()<<endl;
    }
}
```

### 3.7.3 this 指针

this 指针是一个隐含于每一个成员函数中的特殊指针。它是一个指向正操作该成员函数的对象。当对一个对象调用成员函数时，编译程序先将对象的地址赋给 this 指针，然后调用成员函数。每次成员函数存取数据成员时，C++ 编译器将根据 this 指针所指向的对象来确定应该引用哪一个对象的数据成员。

通常 this 指针在系统中是隐含存在的，也可以把它显式表示出来。

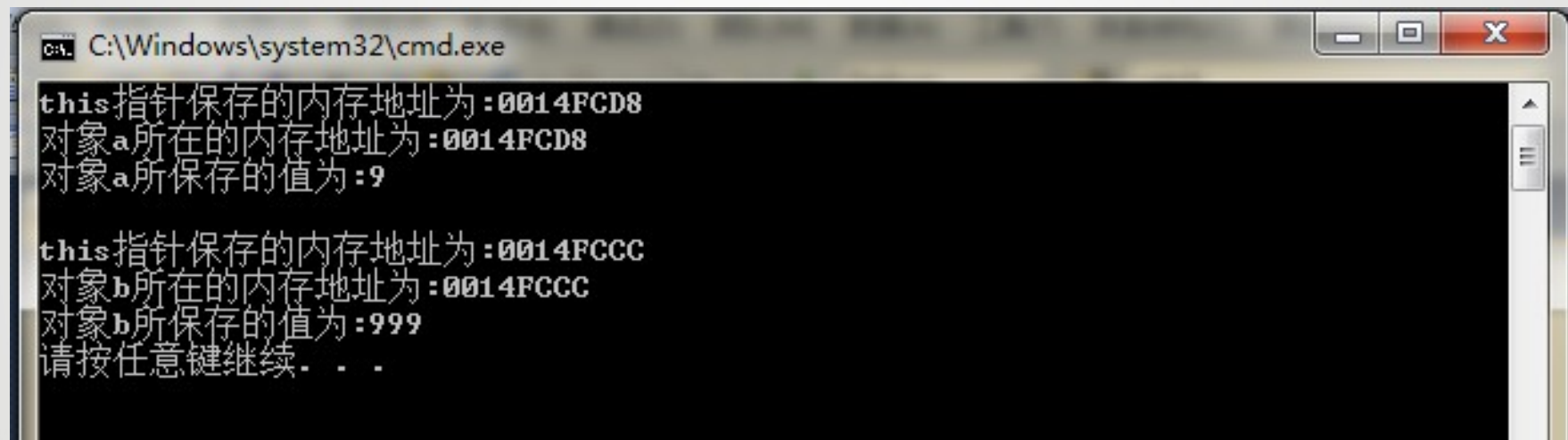
**注意：**静态成员函数没有 this 指针。

【例 3-24】 this 指针应用举例。

```
#include <iostream>
using namespace std;
class A
{
public:
    int get() {return i;}
    void set(int x)
    {
        this->i=x;
        cout<<"this 指针保存的内存地址为 : "<<this<<endl;
    }
private:
    int i;
};
```



```
int main()
{
    A a;
    a.set(9);
    cout<<" 对象 a 所在的内存地址为 :"<<&a<<endl;
    cout<<" 对象 a 所保存的值为 :"<<a.get()<<endl;
    cout<<endl;
    A b;
    b.set(999);
    cout<<" 对象 b 所在的内存地址为 :"<<&b<<endl;
    cout<<" 对象 b 所保存的值为 :"<<b.get()<<endl;
    return 0;
}
```



```
C:\Windows\system32\cmd.exe  
  
this 指针保存的内存地址为:0014FCD8  
对象a所在的内存地址为:0014FCD8  
对象a所保存的值为:9  
  
this 指针保存的内存地址为:0014FCCC  
对象b所在的内存地址为:0014FCCC  
对象b所保存的值为:999  
请按任意键继续. . .
```

- 在 C++ 中，多个对象的数据成员存储各自存储，但成员函数只存储一个副本，如何区分当前调用成员函数的对象呢？解决的方法就是 this 指针。
- this 指针：一个指向对象的指针，它隐含在类的成员函数中，用来指向成员函数所属类当前正在被操作的对象。
- 问题：类中数据成员的名字与成员函数形参的名字相同时，如何区分？

- 【例 3-25】阅读程序，分析执行结果。

- `#include<iostream>`
- `using namespace std;`
- `class A`
- `{`
- `public:`
- `A(int =0,int =0);`
- `void Show();`
- `void Set(int,int);`
- `~A();`
- `private:`
- `int x,y;`
- `};`
- `A::A(int x1,int y1):x(x1),y(y1)`
- `{`
- `cout<<" 调用构造函数! "<<endl;`
- `}`

```
A::~A()
{   cout<<" 调用析构函数! "<<endl;   }
void A::Set(int x,int y)
{
    this->x=x; this->y=y;    // 数据成员加 this 指针
}
void A::Show()
{
    cout<<"x="<<x<<" "<<"y="<<y<<endl;
}
int main()
{
    A a(10,20);
    a.Show();
    a.Set(100,200);
    a.Show();
    return 0;
}
```

运行结果：  
调用构造函数！  
x=10 y=20  
x=100 y=200  
调用析构函数！



# 3.8

## 向函数传递对象

- C++ 语言中，函数的参数和返回值的传递方式有三种：值传递、 指针传递和引用传递。其方法与传递其它类型的数据一样。

### 3.8.1 使用对象作为函数参数

把作为实参的对象的值复制给形参创建的局部对象，这种传递是**单向**的，只从实参到形参。因此，函数对形参值做的改变**不会**影响到实参。



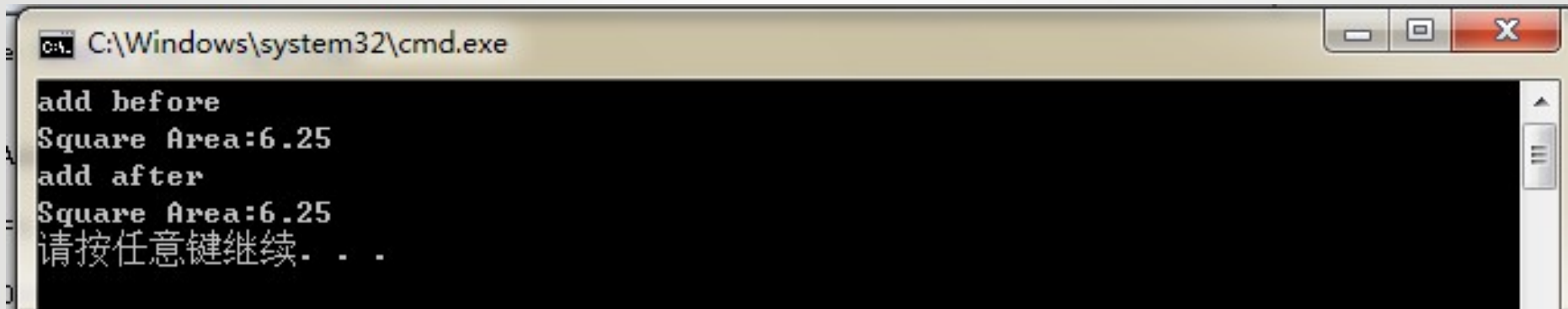
**【例 3-26】** 对象作为函数参数应用举例。

```
#include <stdafx.h>
#include <iostream>
using namespace std;
class Square
{
private:
    double length;
public:
    Square(double len);
    void Add(Square s);
    void Outpout();
};
```



```
Square::Square (double len):length(len)
{
}
void Square::Add (Square s)
{
    s.length =s.length +1.0;
}
void Square::Outpout()
{
    cout<<"Square Area:"<<length * length<<endl;
}
```

```
int main()
{
    Square s (2.5);
    cout<<"add before"<<endl;
    s.Output ();
    s.Add (s);
    cout<<"add after"<<endl;
    s.Output ();
    return 0;
}
```



```
C:\Windows\system32\cmd.exe  
add before  
Square Area:6.25  
add after  
Square Area:6.25  
请按任意键继续. . .
```

### 3.8.2 使用对象指针作为函数参数

对象指针作为参数传递的是**地址**。也就是说实参向形参传递的是实参所指向对象的地址。也就是实参对象指针变量和形参对象指针变量指向同一内存地址，因而作为形参的对象，其值的改变，也就是改变了**实参对象的值**，所以指针传递是一种**双向传递**。

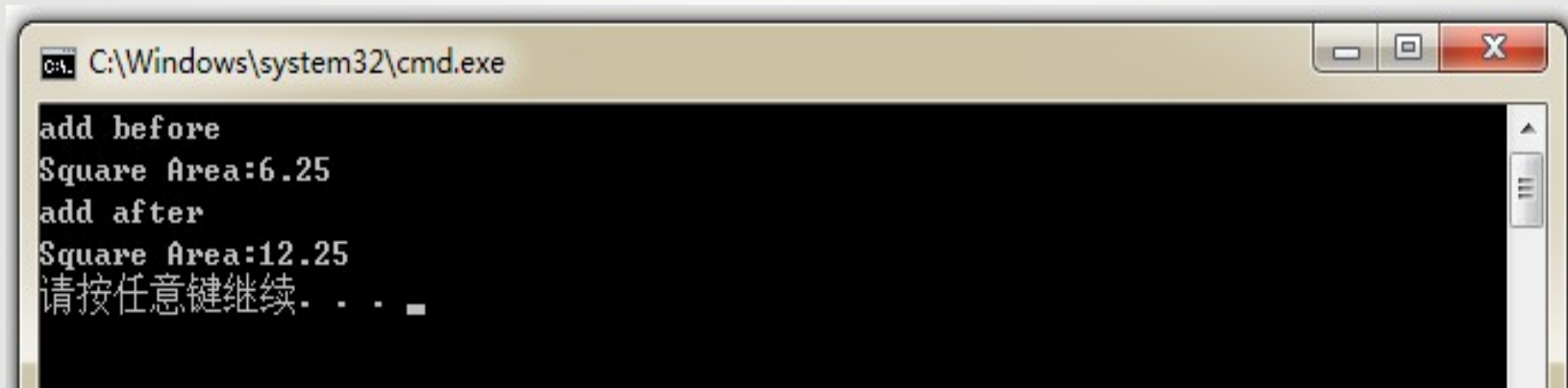
【例 3-27】，验证对象指针作为函数参数是属于双向传递。

```
class Square
{
private:
    double length;
public:
    Square(double len);
    void Add(Square *s);
    void Outpout();
};
```

```
Square::Square (double len):length(len)
{
}
void Square::Add (Square *s)
{
    s->length =s->length +1.0;
}
void Square::Outpout ()
{
    cout<<"Square Area:"<<length * length<<endl;
}
```



```
int main()
{
    Square s(2.5);
    cout<<"add before"<<endl;
    s.Output ();
    s.Add (&s);
    cout<<"add after"<<endl;
    s.Output ();
    return 0;
}
```



```
C:\Windows\system32\cmd.exe

add before
Square Area:6.25
add after
Square Area:12.25
请按任意键继续. . .
```



### 3.8.3 使用对象引用作为函数参数

采用了引用方式进行参数传递，形参对象就相当于实参对象的“别名”，对形参的操作其实就是对实参的操作。

使用对象引用作为函数参数不但有指针作为参数的优点，而且比指针作为参数更简单、更直接。

【例 3-28】用对象引用进行参数传递。

```
#include <stdafx.h>
#include <iostream>
using namespace std;
class Square{
private:
    double length;
public:
    Square(double len);
    void Add(Square &s);
    void Outpout();
};
```

```
Square::Square (double len):length(len)
{
}
void Square::Add (Square &s)
{
    s.length =s.length+1.0;
}
void Square::Output ()
{
    cout<<"Square Area:"<<length * length<<endl;
}
```



```
int main()
{
    Square s(2.5);
    cout<<"add before"<<endl;
    s.Output ();
    s.Add (s);
    cout<<"add after"<<endl;
    s.Output ();
    return 0;
}
```

使用引用时，应该注意遵循如下规则：

- （ 1 ） 引用被创建的同时**必须被初始化**（指针则可以在任何时候被初始化）。
- （ 2 ） **不能有 NULL 引用**，引用必须与合法的存储单元关联（指针则可以是 NULL ）。
- （ 3 ） 一旦引用被初始化，就**不能改变引用的关系**（指针则可以随时改变所指的对象）。

### 3.8.4 三种传递方式比较

(1) 值传递是单向的，形参的改变并不引起实参的改变。指针和引用传递是双向的，可以将改变由形参“传给”实参。

(2) 引用是 C++ 中的概念。 `int m; int &n = m;` `n` 相当 `m` 的别名或者绰号，对 `n` 的任何操作就是对 `m` 的操作。所以 `n` 既不是 `m` 的拷贝，也不是指向 `m` 的指针，其实 `n` 就是 `m` 它自己。实际上“引用”可以做的任何事情“指针”也都能够做。

### 3.8.4 三种传递方式比较

（3）指针能够毫无约束地操作内存中的任何东西。指针虽然功能强大，但是用起来十分危险，所以如果的确只需要借用一下某个对象的“别名”，那么就用“引用”，而不要用“指针”，以免发生意外。

（4）使用引用作为函数参数与使用指针作为函数参数相比较，前者更容易使用、更清晰，而且当参数传递的数据较大时，引用传递参数的效率高和所占存储空间更小。

【例 3-29】三种传递方式比较举例。

```
#include <stdafx.h>
#include <iostream>
using namespace std;
    // 值传递
void change1(int n)
{
    cout<<"\n"<<" 值传递 -- 函数操作地址 "<<&n;
        // 显示的是拷贝的地址而不是源地址
    n++;
}
```





// 引用传递

```
void change2(int &n)
```

```
{
```

```
    cout<<"\n"<<" 引用传递 -- 函数操作地址 "<<&n;
```

```
    n++;
```

```
}
```

// 指针传递

```
void change3(int *n)
```

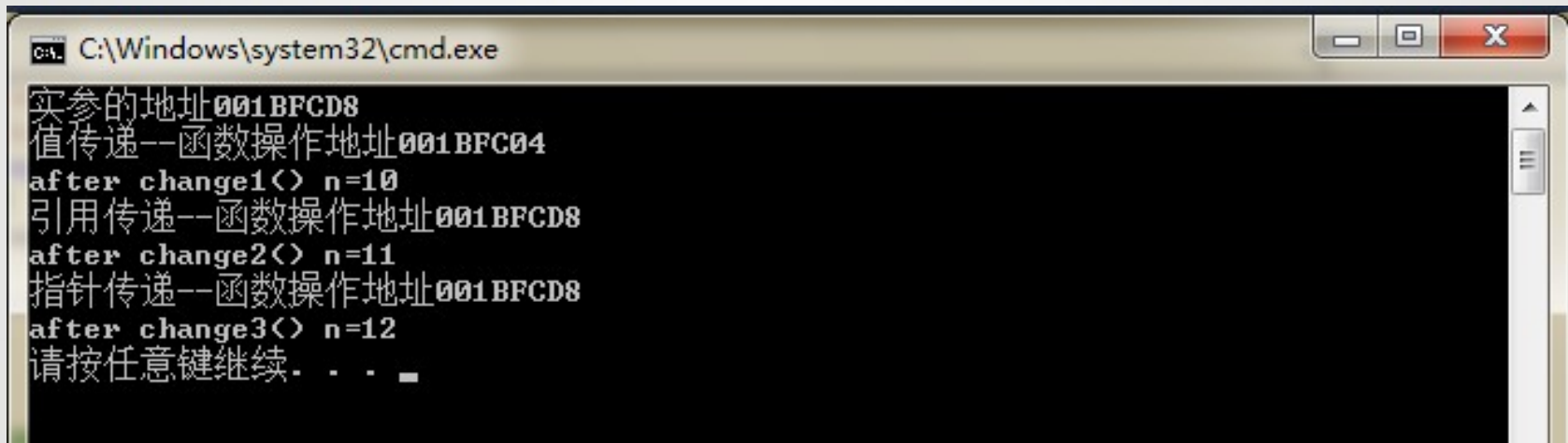
```
{
```

```
    cout<<"\n"<<" 指针传递 -- 函数操作地址 "<<&*n ;
```

```
    *n=*n+1;
```

```
}
```

```
int main()
{
    int n=10;
    cout<<" 实参的地址 "<<&n;
    change1(n);
    cout<<"\n"<<"after change1() n="<<n;
    change2(n);
    cout<<"\n"<<"after change2() n="<<n;
    change3(&n);
    cout<<"\n"<<"after change3() n="<<n<<"\n";
    return 0;
}
```



```

C:\Windows\system32\cmd.exe
实参的地址001BFC08
值传递--函数操作地址001BFC04
after change1() n=10
引用传递--函数操作地址001BFC08
after change2() n=11
指针传递--函数操作地址001BFC08
after change3() n=12
请按任意键继续. . .
    
```

代码效率上看。以对象传递方式的效率相对低一些。它需要创建新的对象来接受实参传来的值。用对象指针传递的方式效率会略高一些。而当为对象引用形式时效率就更高，因为它就是实参本身。



# 3.9

## 对象的赋值和复制

同类的对象之间可以互相赋值。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。这是通过对赋值运算符的重载实现的。实际上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 必须属于同一个类的两个对象。

**【例 3-30】对象赋值举例。**

```
class Cube
{
public:
    Cube(int=10,int=10,int=10 )
    int volume();
private:
    int height;
    int width;
    int length;
};
```



```
Cube::Cube(int h,int w,int len)
{
    height=h;
    width=w;
    length=len;
}
int Cube::volume()
{
    return(height * width * length); // 返回体积的值
}
```



```
int main()
{
    Cube Cube1(20,20,20),Cube2;
    cout<<"The volume of Cube1 is "<<Cube1.volume()<<endl;
    cout<<"The volume of Cube2 is "<<Cube2.volume()<<endl;
    Cube2=Cube1;    // 将Cube1 的值赋给Cube2
    cout<<"Cube2=Cube1 "<<endl;
    cout<<"The volume of Cube2 is "<<Cube2.volume()<<endl;
    return 0;
}
```



```
C:\Windows\system32\cmd.exe

The volume of Cube1 is 8000
The volume of Cube2 is 1000
Cube2=Cube1
The volume of Cube2 is 8000
请按任意键继续. . .
```

说明：

（1）对象的赋值只对其中的数据成员赋值，不对成员函数赋值。数据成员是占存储空间的，不同对象的数据成员占有不同的存储空间，赋值的过程是将一个对象的数据成员在存储空间的状态复制给另一对象的数据成员的存储空间。而不同对象的成员函数是同一个函数代码段，不需要、也无法对它们赋值。

（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意想不到的严重后果。

对象初始化有两种方法：

- ( 1 ) 创建对象时由构造函数初始化；
- ( 2 ) 用已有的同类的对象通过赋值方式进行初始化。

通过赋值方式进行初始化的过程，实际上是通过类的拷贝构造函数来完成的。

拷贝构造函数是一种特殊的构造函数，它具有一般构造函数的所有特性，但其形参是本类对象的引用。

作用：使用一个已经存在的对象去初始化同类的另一个对象。



拷贝构造函数定义格式如下：

构造函数名（类名 &）；

拷贝构造函数的参数采用引用方式。

■ 拷贝构造函数特点：

- （1）函数名与类名相同，并且该函数没有返回值。
- （2）该函数只有一个参数，并且是同类对象的引用。
- （3）每个类都必须有一个拷贝构造函数。如果类中没有定义，则系统会自动生成一个默认拷贝构造函数

```
class B{  
public:  
    B();  
    B(B &);  
}  
int main(){  
    B b1;  
    B b2=b1;  
}
```

与赋值语句的区别:

( 1 )	B b1;	( 2 )	B b1 , b2;
	B b2=b1;		b2=b1;
	[		{

使用拷贝构造函数时应**注意**以下问题：

- （1）并不是所有的类声明都需要拷贝构造函数，仅当准备用传值的方式传递类对象时，才要拷贝构造函数。
- （2）拷贝构造函数的名字必须与类名相同，并且没有返回值。
- （3）拷贝构造函数只有一个参数，必须是本类对象的引用。
- （4）每一个类必须至少有一个拷贝构造函数。如果用户在定义类时没有给出拷贝构造函数，系统会自动产生一个**缺省的拷贝构造函数**。

## 什么时候调用拷贝构造函数？

调用拷贝构造函数的情况有三种：

- （1）明确表示由一个对象初始化另一个对象。
- （2）当对象作为函数实参传递给函数形参时。
- （3）当对象作为函数的返回值，创建一个临时对象。



【例 3-31】复制构造函数 3 种调用机制的实例

```
#include<iostream>
using namespace std;
class Date{
public:
    Date(int y=2000, int m=2, int d=2);    // 带默认参数的构造函数
    Date(const Date& t);                  // 拷贝构造函数
    void Show();
private:
    int year, month, day;
};
Date:: Date(int y, int m, int d){
    year = y;  month = m;  day = d;
    cout<<" 带默认参数的拷贝函数已被调用。 \n";
}
```

```
Date::Date(const Date &t)
{
    year = t.year ; month = t.month ; day = t.day ;
    cout<<" 拷贝构造函数已被调用。 \n";
}

void Date::Show()
{
    cout<<year<<". "<<month<<". "<<day<<endl; }

void fun1(Date t) // 第 2 种情况，函数参数为类的对象
{
    cout<<" 开始执行 fun1 函数： "<<endl;
    t.Show();
}

Date fun2() // 第 3 种情况，函数的返回值为类的对象
{
    cout<<" 开始执行 fun2 函数： "<<endl;
    Date t(2010,4,4);
    return t;
}
```

```
int main(){
    Date t1(2005,10);
    t1.Show();
    cout<<"***** 第一种情况 *****"<<endl;
    Date t2(t1);
    t2.Show();
    cout<<"***** 第二种情况 *****"<<endl;
    cout<<" 调用 fun1 函数: "<<endl;
    fun1(t1);
    cout<<"***** 第三种情况 *****"<<endl;
    cout<<" 调用 fun2 函数之前: "<<endl;
    t1.Show();
    cout<<" 调用 fun2 函数: "<<endl;
    t1=fun2();
    t1.Show();
    return 0;
}
```



- 运行结果：
- 带默认参数的构造函数已被调用。
- 2005.10.2
- \*\*\*\*\* 第一种情况 \*\*\*\*\*
- 拷贝构造函数已被调用。
- 2005.10.2
- \*\*\*\*\* 第二种情况 \*\*\*\*\*
- 调用 fun1 函数：
- 拷贝构造函数已被调用。
- 开始执行 fun1 函数：
- 2005.10.2
- \*\*\*\*\* 第三种情况 \*\*\*\*\*
- 调用 fun2 函数之前：
- 2005.10.2
- 调用 fun2 函数：
- 开始执行 fun2 函数：
- 带默认参数的构造函数已被调用。
- 拷贝构造函数已被调用。
- 2010.4.4

## 构造函数和析构函数

1. 当用类的一个对象去初始化该类的另一个对象时系统自动调用拷贝构造函数实现拷贝赋值。

```
int main()
{
    Point A(1,2);
    Point B(A); // 拷贝构造函数被调用
    cout<<B.GetX()<<endl;
}
```

## 构造函数和析构函数

2. 若函数的形参为类对象，调用函数时，实参赋值给形参，系统自动调用拷贝构造函数。例如：

```
void fun1(Point p)
{
    cout<<p.GetX()<<endl;
}

int main()
{
    Point A(1, 2);
    fun1(A); // 调用拷贝构造函数
}
```

## 构造函数和析构函数

3. 当函数的返回值是类对象时，系统自动调用拷贝构造函数。

例如：

```
Point fun2()  
{  
    Point A(1, 2);  
    return A; // 调用拷贝构造函数  
}  
  
int main()  
{  
    Point B;  
    B=fun2();  
}
```



**【例 3-32】** 拷贝构造函数应用举例。

设计一个复数类，两个数据成员分别表示复数的实部和虚部。定义两个构造函数，一个是具有两个参数双精度实型的普通构造函数，另一个是拷贝构造函数，

两个构造函数分别在不同的情况下初始化对象。定义 add 函数完成两个虚数的加法。





```
class Complex
{
public:
    Complex(double r,double i);
    Complex(Complex &c);
    Complex add(Complex c);
    void Output();
private:
    double real,image;
};
```

```
Complex::Complex(double r,double i):real(r),image(i)
{
    cout<<" 调用两个参数的构造函数 "<<endl;
}
Complex::Complex(Complex &c)
{
    real = c.real;
    image = c.image;
    cout<<" 调用拷贝构造函数 "<<endl;
}
```



```
void Complex::Output()
{
    cout<<"("<<real<<","<<image<<")"<<endl;
}
Complex Complex::add(Complex c)
{
    Complex y(real + c.real,image + c.image);
    return y;
}
void f(Complex n)
{
    cout<<"n=";
    n.Output();
}
```



```
int main()
{
    ( 1 )    Complex a(3.0,4.0),b(5.6,7.9);
    ( 2 )    Complex c(a);
              cout<<"a=";
              a.Output();
              cout<<"c=";
              c.Output();
    ( 3 )    f(b);
    ( 4 )    c = a.add(b);
              c.Output();
}
```

```

C:\Windows\system32\cmd.exe
调用两个参数的构造函数
调用两个参数的构造函数
调用拷贝构造函数
a=(3,4)
c=(3,4)
调用拷贝构造函数
n=(5.6,7.9)
调用拷贝构造函数
调用两个参数的构造函数
调用拷贝构造函数
(8.6,11.9)
请按任意键继续. . . _
    
```

3.10



## 对象的组合

类的数据成员不但可以是基本类型，而且也可以是自定义类型，当然也可以是类的对象。

所谓类的组合是指一个类内嵌其它类的对象作为本类的成员。两者之间是包含与被包含的关系。

```
class B  
{ .....  
}
```

```
class A  
{ B b;  
}
```

创建类的对象时，如果这个类具有内嵌对象成员，那么各个内嵌对象应首先被自动创建。因此，在创建类的对象时，既要对本类的基本类型数据成员进行初始化，同时也要对内嵌对象成员进行初始化。



组合类构造函数的**定义格式**为：

类名 :: 类名（形参表）：内嵌对象 1（形参表），内嵌对象  
2（形参表），.....

{

    // 类的初始化

}

注意:

( 1 ) 类的构造函数的形参表中的形参, 不但要考虑对本类基本类型数据成员的初始化工作, 而且也要考虑内嵌对象的初始化工作。也就是说, 类的形参列表应该由对象成员所需形参和本类基本类型数据成员所需形参两部分组成。

（ 2 ）在创建一个组合类的对象时，不仅它自身的构造函数将被调用，而且其内嵌对象的构造函数也将被调用。这时构造函数调用的顺序为：

- ① 调用内嵌对象的构造函数，调用顺序按照内嵌对象在组合类的声明中出现的先后顺序依次调用。
- ② 执行本类构造函数的函数体。
- ③ 析构函数的调用顺序与构造函数刚好相反。

（ 3 ）若调用缺省构造函数（即无形参的），则内嵌对象的初始化也将调用相应的缺省构造函数。

（ 4 ）组合类同样有拷贝构造函数。若无则调用默认的拷贝构造函数。

【例 3-33】类的组合应用举例。定义点类 Point 和求两点间距离的类 Distance, 观察两个类的构造函数和析构函数被调用的顺序。

```
class Point
{
    private:
        float x, y;
public:
    Point(float xx, float yy)
    {
        cout<<"point 构造函数被调用 "<<endl;
        x=xx;
        y=yy;
    }
}
```

```
Point(Point &p)
{
    x=p.x;
    y=p.y;
    cout<<"pont 拷贝构造函数被调用 "<<endl;
}
~Point()
{
    cout<<"Point 析构造函数被调用 "<<endl;
}
float GetX()
{
    return x;
}
```

```
float GetY()  
{  
    return y;  
}  
};
```

```
class Distance
{
    private:
        Point p1,p2;
        double dist;
    public:
        Distance(Point a,Point b); // 构造函数
        Distance(Distance & d); // 拷贝构造函数
        ~Distance();
        double GetDis();
};
```



```
Distance::Distance(Point a,Point b):p1(a),p2(b)
{
    double x=double(p1.GetX()-p2.GetX());
    double y=double(p1.GetY()-p2.GetY());
    dist=sqrt(x*x+y*y);
    cout<<"Distance 构造函数被调用 "<<endl;
}
Distance::Distance(Distance & d):p1(d.p1 ),p2(d.p2 )
{
    cout<<"Distance 拷贝构造函数被调用 "<<endl;
    dist=d.dist ;
}
```

```
Distance::~~Distance ()
{
    cout<<"Distance 析构函数被调用 "<<endl;
}
double Distance::GetDis()
{
    return dist;
}
```

```
int main()
{
    (1) Point pa(2,2),pb(5,5); // 创建 pa,pb
    (2) Distance da(pa,pb); // 创建 da
    (3) Distance db(da); // 创建 db
    cout<<" 通过 da 得到点 (2,2) 到点 (5,5) 的距离为: " << da.GetDis () << endl;
    cout<<" 通过 db 得到点 (2,2) 到点 (5,5) 的距离为: " << db.GetDis () << endl;
} (4) 删除 db
    删除 da
    删除 pb
    删除 pa
```

分析:

( 1 ) `main()` 的第一条语句 “`Point pa(2, 2), pb(5, 5);`” 声明了两个 `Point` 对象 `pa` 和 `pb`, `Point` 的构造函数被调用 2 次。

( 2 ) 第二条语句 “`Distance da(pa, pb);`” 把 `Point` 类的对象 `pa` 和 `pb` 传递给形参, 并且在 `Distance` 的构造函数中用 `Point` 的对象 `a`, `b` 对 `da` 对象的 `p1` 和 `p2` 对象成员初始化, `Point` 的拷贝构造函数被调用了 4 次。在对内嵌类对象 `p1`, `p2` 初始化完成后, 开始对本类对象进行初始化, 调用 `Distance` 的构造函数 1 次。当 `Distance` 构造函数执行结束前, `Point` 类的对象 `a`, `b` 被删除, 调用 `Point` 的析构函数 2 次。

( 3 ) 第三条语句 “ Distance db(da); ” 调用 Distance 的拷贝构造函数完成对 db 对象的初始化。按照内嵌类对象先构造的原则, 先对 db 对象的 Point 类对象成员 p1 , p2 进行初始化, 所以先调用 Point 类的拷贝构造函数 2 次, 再调用 Distance 类的拷贝构造函数 1 次。

( 4 ) 第四、五条语句 “ cout<< ” 通过 da 得到点 (2, 2) 到点 (5, 5) 的距离为: “<<da.GetDis ()<<endl;        cout<< ” 通过 db 得到点 (2, 2) 到点 (5, 5) 的距离为: “<<db.GetDis ()<<endl; ” 分别输出点 ( 2, 2 ) 和点 ( 5, 5 ) 之间的距离。

( 5 ) 程序运行结束前, 按照与构造相反的次序析构各对象

# 3.11



## 程序实例

【例 3-34】实现一个简单的学生成绩管理系统。

通过该系统，可以进行学生信息的插入、删除和输出。

分析：为了方便对学生信息的操作，应定义一个结构体 `Student_s`，包括学号、姓名和成绩；设计一个学生类 `Student_c`，其中数据成员 `Student_struct[MAXSIZE]` 表示最多存放 `MAXSIZE` 个学生，每个元素代表一个学生；`total` 当前线性表中元素的个数，也就是学生的人数。

```
struct Student_s
{
    long no;
    char name[10];
    float score;};
class Student_c
{
private:
    Student_s Student_struct[MAXSIZE];
    int total;
public:
    Student_c();
    int Insert_seq(int i, Student_s x);           // 插入第 i 个学生的信息
    int Delete_seq(int i);                       // 删除第 i 个学生
    void Print_seq();                             // 打印所有学生信息
};
```



```
void menu();  
int main()  
{  
    Student_c Student_Object;  
    int n;  
    bool m=true;  
    while(m)  
    {  
        menu();  
        cin>>n;  
        switch(n)  
        {
```

case 1:

```
{  
    int i;  
    Student_s x;  
    cout<<" 请输入插入位置: ";  
    cin>>i;  
    cout<<" 请输入学生的学号、姓名和成绩: "<<endl;  
    cin>>x.no >>x.name >>x.score ;  
    Student_ Object.Insert_seq(i,x);  
    cout<<" 插入后的情况: "<<endl;  
    Student_ Object.Print_seq();  
    break;  
}
```

```
    case 2:
    {
        int i;
        cout<<" 请输入删除位置: ";
        cin>>i;
        Student_ Object.Delete_seq(i);
        cout<<" 删除后的情况: "<<endl;
        Student_ Object.Print_seq();
        break;
    }
    case 0:m=false;
    }
}
return 0;
}
```

```
Student_c::Student_c ()
{
    total=0;
}
int Student_c::Insert_seq (int i,Student_s x)
{
    int j;
    if(total==MAXSIZE)
    {
        cout<<"table is full"<<endl;
        return -1;
    }
    if(i<1 || i>(total+1))
    {
        cout<<"place is wrong!"<<endl;
        return 0;
    }
}
```

```
    for(j=total-1;j>=i-1;j--)
    {
        Student_struct[j+1]=Student_struct[j];
    }
    Student_struct[i-1]=x;
    ++total;
    return 1;
}
```

```
int Student_c::Delete_seq (int i)
{
    int j;
    if(i<1||i>total)
    {
        cout<<"this element don't exist!"<<endl;
        return -1;
    }
    for(j=i;j<=total-1;j++)
    {
        Student_struct[j-1]=Student_struct[j];
    }
    --total;
    return 1;
}
```

```
void Student_c::Print_seq ()
{
    int i;
    for (i=0;i<=total-1;i++)
    {
        cout<<Student_struct[i].no<<setw(10)<<Student_struct[i].name
        <<setw(10)<<Student_struct[i].score <<endl;
    }
    cout<<endl<<endl;
}
```

```

C:\Windows\system32\cmd.exe

1. 插入
2. 删除
0. 退出

请选择: 1
请输入插入位置: 3
请输入学生的学号、姓名和成绩:
3 王武 82
插入后的情况:
2 李斯 76
1 张三 93
3 王武 82

1. 插入
2. 删除
0. 退出

请选择: 2
请输入删除位置: 1
删除后的情况:
1 张三 93
3 王武 82

1. 插入
2. 删除
0. 退出
    
```