第3章 类与对象

类是面向对象技术的核心机制，是面向对象设计中对具有相同或相似性质的对象的抽象，是对数据和操作进行封装的载体，进而完成对数据的安全、高效、合理的访问；对象是类的实例，是类的具体个体，对应现实世界中的实体。

本章主要内容：

- 类的定义与实现。
- 构造函数与析构函数。
- 拷贝构造函数。
- this 指针。
- static 成员。
- const 对象与 const 成员。
- friend 友元。

3.1 理 解 类

软件研发就是要把现实世界中需要人们手工完成的各项工作利用计算机软件和硬件来完成。现实世界是由各式各样的事物组成的，每一项工作都需要若干事物协同工作，如一个教学任务，需要教师、教室、学生、试卷等协同工作。面向过程的软件开发把现实世界的各种工作映射成软件世界中的若干种功能，这种映射看似简单、直接，但是实际上现实世界的各项工作间关系复杂，并且经常变动（由于各工作间关系复杂，一项工作变动将会引起若干个工作的变动），这样映射的软件也需要随着现实的变动而变更，这种变更对软件研发来说，工作量是巨大的。事实上，由于各项工作间的复杂关系，最初建立映射关系也非常困难（多对多、层次、包含等各种关系）。由于现实问题领域中的事物是有限的，为此，可以考虑另外一种映射方式，把现实世界中的每个事物，在软件世界中建立相应的对象。现实世界中每个事物都有自己的数据和功能，那么对应软件世界中的对象也对应相应的属性和操作，这种映射使初始的建立和应对变更都很容易。

现实事物和软件中的对象如何映射呢？具体的映射过程就是抽象与封装的过程，来看看下面的例子。

观察各种各样的钟表，发现它们都是用时针、分针、秒针记载时间，并且时针、分针、秒针是被表的外壳保护（封条）起来的，用户不可以打开表盖直接拨动指针调整时间，为此，钟表需要提供一个可以调整时间的旋钮（用户不需要知道内部时间是如何调整的）和提供可以观

察时间的界面。可以把所有的钟表归为 Clock 一类，Clock 包括时、分、秒属性，并且设置为私有的，不允许外面直接访问；Clock 提供对外可以设置时间、显示时间的方法。

观察各种各样的仓库，发现它们都有库存货品，但是安全起见，一般不允许用户直接进入仓库查询、存取货品，为此需要填写查询单、出库单、入库单交给仓库管理员处理，用户不需要了解仓库管理员具体的查询、出入库操作过程。假如仓库里面优化了流程、改进了服务，用户同样进行查询、出入库请求的方式不变，但是实际上已经享用了升级的服务。可以把所有的仓库归为 Storehouse 一类，Storehouse 包括货物清单属性，并且设置为私有的，不允许外面直接访问；Storehouse 提供对外可以使用的查询、出库、入库方法。

通过上面两个例子发现：**类是具有相同或相似性质的对象的抽象**。抽象是从众多的事物中抽取出共同的、本质性的特征，而舍弃其非本质的特征的过程。

图 3.1 表示了类、对象、实体的相互关系和面向对象的分析思维方式。在用面向对象的软件方法解决现实世界的问题时，首先将物理存在的实体抽象成概念世界的抽象数据类型，这个抽象数据类型里面包括实体中与问题域相关的数据和操作；然后再用面向对象的工具，如 C++语言，将这个抽象数据类型用计算机逻辑表达出来，即构造计算机能够理解和处理的类；最后将类实例化就得到了现实世界实体的面向对象的映射——对象。在程序中对对象进行操作，就可以模拟现实世界中实体上的问题并解决之。

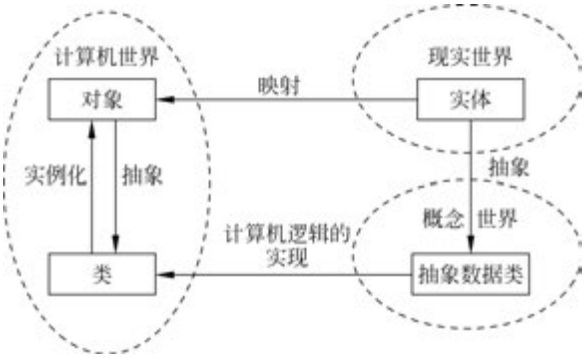

图 3.1　实体对象关系映射图

类还起到封装的作用，封装有两个含义：**一是把对象的全部属性和行为结合在一起，形成一个不可分割的独立单位**。对象的属性值(除了公有的属性值)只能由这个对象的行为来读取和修改；**二是尽可能隐蔽对象的内部属性和实现细节，对外形成一道屏障，与外部的联系只能通过公共接口实现**。如调整时间、显示时间、库存查询、入库、出库等，而隐藏了时、分、秒及库存数据，同时隐藏了公共接口调整时间、显示时间、库存查询、入库、出库等方法的实现细节，也就是类的用户见不到其具体实现。

封装信息的隐藏作用反映了事物的相对独立性，可以只关心它对外所提供的接口，即能做什么，而不注意其内部细节，即如何提供这些服务。

封装的结果使对象以外的部分不能随意存取对象的内部属性，从而有效地避免了外部错误对它的影响，大大减小了查错和排错的难度。另一方面，当对象内部进行修改时，由于它只通过少量的外部接口对外提供服务，因此同样减小了内部的修改对外部的影响。

封装机制将对象的使用者与设计者分开，使用者不必知道对象行为实现的细节，只需要

用设计者提供的外部接口让对象去做。封装的结果实际上屏蔽了复杂性、耦合性，从而降低了软件开发的难度。

3.2 类的定义与实现

在面向对象程序设计中，程序模块是由类构成的，类是对象的属性、功能的抽象描述。C++中的类是数据及相关函数的封装体，类是一种用户自定义数据类型，称为类(class)类型。类的概念从结构体扩展而来，类保留了结构体的成员变量，扩展了表示行为功能的方法，用与数据相关联的函数表示。类的成员由成员变量(数据)和成员函数(方法)共同构成。C++为达到数据封装和信息隐藏的目的，采用访问限定的控制方法，用成员访问限定符private(私有的成员)、public(公有的成员)和 protected(保护的成员)控制访问对象，保障成员数据和成员函数的使用安全。

3.2.1 类的定义

类定义的语法格式为

```
class 类名
{
private:
    私有的成员数据和成员函数声明;
public:
    公有的成员数据和成员函数声明;
protected:
    保护的成员数据和成员函数声明;
};
```

关键字 class 定义数据类型是一个类 class 类型；类名由用户命名，是类的标识；一对花括号表示类的边界，“{”表示类的开始，“}”表示类的结束，注意，“}”后必须以“;”结尾。成员访问限定符 private 声明私有的成员，只能被本类中的成员函数引用，不能被本类以外(除了友元类)的其他函数引用，省略了所有的成员访问限定符，默认为私有的成员；成员访问限定符 public 声明公有的成员，既可以被本类中的函数引用，也可以被本类作用域内的其他函数引用；成员访问限定符 protected 声明受保护的成员，可以被本类中的成员函数引用，可以被派生类的成员函数引用，不能被本类外其他函数引用。

成员数据的声明格式为

```
类型 成员数据名;
```

成员函数的声明格式为

```
类型 成员函数名(形参列表);
```

例如，Clock 类定义如下。

```
class Clock
{
private :
    int hour, minute, second;                    //关于时间的数据
public :
```

```
    void setClock(int h, int m, int s);          //调整时间值
    void showClock();                            //显示时间值
};
```


成员数据的声明方法与一般变量的声明方法相同,数据类型可以是C++中任意合法的数据类型,包括类class类型,如定时炸弹类TimeBomb定义如下。

```
class TimeBomb
{
private :
    Clock timer;                                 //关于定时器的数据
    int explosive;                               //炸药量
    …
public :
    void setTime();                              //设置爆炸时间
    …
};
```

其中,定时器就是一个类Clock类型。

类class定义应注意以下几点。

(1) 类class是定义面向对象程序模块的数据类型,类是生成对象的“模板”,但不是对象,不能接收也不能存储数据,系统不为类分配存储空间。只有给类定义了对象以后,对象才接收并存储具体的值,系统只给对象分配空间。

(2) 类的定义是声明一个数据结构,而不是定义一个函数,定义的最后由分号结束。

(3) 说明类成员访问权限的关键字private、protected和public可以按任意顺序、任意出现多次,但一个成员只能有一种访问权限。为使程序更加清晰,应将私有成员和公有成员归类放在一起。

(4) 不能在类内给数据成员赋初值,只有在类的对象定义以后才能给数据成员赋初值。

注:C++ 11中非静态成员变量(常量)可以直接初始化,或者在初始化表中初始化。

(5) 成员函数可以重载,如可以在Clock中可以定义多个名为set的函数。

```
void setClock(int h, int m, int s);              //用来设置时、分、秒
void setClock(int h, int m);                     //用来设置时、分
void setClock(int h);                            //用来设置时
```

3.2.2 类的实现

成员函数用于描述类的行为,处理数据成员,可以在类内声明并实现定义,例如:

```
class Clock
{
private :
    int hour, minute, second;                    //关于时间的数据
public :
    void setClock(int h, int m, int s){
        hour = (h >= 0 && h <= 23)? h : 0;
        minute = (m >= 0 && m <= 59)? m : 0;
        second = (s >= 0 && s <= 59)? s : 0;
    }
```

```
    void showClock(){
        cout << hour << minute << second;
    }
};
```

然而，人们习惯上在类的定义中只声明其函数原型，在类外定义函数的实现，这样也方便设计与实现的分离。成员函数名前必须加上类名，用作用域运算符::连接类名和函数名，即类名::函数名。在类 class 外定义成员函数的语法格式为

```
返回值类型  类名::函数名(形参表列)
{    函数体    }
```

若在类定义中只声明函数成员 setClock 与 showClock，在类外给出两个成员函数定义如下。

```
void Clock::setClock(int h, int m, int s){
    hour = (h >= 0 && h <= 23)? h : 0;
    minute = (m >= 0 && m <= 59)? m : 0;
    second = (s >= 0 && s <= 59)? s : 0;
}
void Clock::showClock(){
    cout << hour << minute << second;
}
```

当函数体中不包括复杂结构，如循环语句和 switch 语句时，对较简单的成员函数可以定义为内联函数。函数实现放在类的定义中，则默认为内联函数。函数实现放在类外，在前面加上 inline 定义为内联函数。例如：

```
inline void Clock::showClock(){
     cout << hour << minute << second;
}
```

当然，函数体中包含复杂结构时，声明为内联函数，编译系统也不会按照内联函数处理。为了程序结构的清晰和设计与实现的分离(声明写在头文件中，实现写在源文件中)，通常将声明写在类内，实现写在类外。

3.3 对象定义及访问

在面向对象的程序设计中，类 class 是指具有相同性质和行为功能的实体抽象。在 C++中定义一个类，只是定义了一种新的数据类型，可以用这种类型定义变量，用类 class 定义的变量称为对象，对象是类的变量，类的对象也称为类的实例。定义了对象才创建了类这种数据类型的物理实体，类和对象的关系是数据类型和变量的关系。

类是生成对象的“模板”，类不接收也不存储数据的值，系统不为类分配存储空间。只有用类定义了对象以后，对象才接收并存储数据的值，系统给对象分配存储空间。

3.3.1 对象的定义

定义对象的方法有以下三种。

(1) 在定义类的同时直接定义对象，即在类定义的右花括号的后面直接写出对象名表，

例如：

```
class Clock
{
private :
    int hour, minute, second;
public :
    void setClock(int h, int m, int s){
        hour = (h >= 0 && h <= 23)? h : 0;
        minute = (m >= 0 && m <= 59)? m : 0;
        second = (s >= 0 && s <= 59)? s : 0;
    }
    void showClock(){
        cout << hour << minute << second;
    }
}clock1,clock2;                    //在定义类的同时定义对象
```

(2) 在定义类的同时直接定义对象，并且不给类命名，这种类结构只有在定义时使用一次，以后无法使用，因此这种定义方法很少使用，例如：

```
class
{
private :
    int hour, minute, second;
public :
      void setClock(int h, int m, int s){
        hour = (h >= 0 && h <= 23)? h : 0;
        minute = (m >= 0 && m <= 59)? m : 0;
        second = (s >= 0 && s <= 59)? s : 0;
    }
    void showClock(){
        cout << hour << minute << second;
    }
}clock1,clock2;                    //在定义类的同时定义对象
```

(3) 定义类以后，在使用对象之前再定义对象，定义的格式与一般变量的定义格式相同。例如，定义了 Clock 类以后，定义对象如下。

```
Clock clock1,clock2;
```

有了对象，就可以利用对象调用其成员函数，对 clock1 和 clock2 分别按照下面方式调用成员函数 setClock。

```
clock1.setClock(1,2,3);
clock2.setClock(4,5,6);
```

调用后类及对象的存储空间如图 3.2 所示。

由图 3.2 可以看出，每个对象有自己独立的数据空间，但是类的成员函数只存储一份，为所有对象共享。那么当使用成员函数 setClock 时，如何确定到底是处理哪个对象的数据呢？这就要看是哪个对象调用的成员函数 setClock，谁调用就修改谁的数据（函数体中 hour、minute、second 代表调用对象的数据），如“clock1.setClock(1,2,3);”把 clock1 的空间中时、分、秒的值置为 1、2、3，实际上是调用对象向成员函数传递了本对象的地址，3.7 节会有介绍。

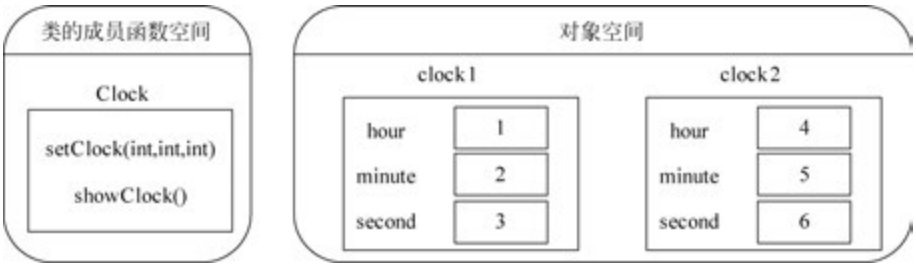

图 3.2　类及对象的存储空间

有了类的对象后，还可以定义指向该类类型的指针和对该类类型的引用，例如：

```
Clock * p;                          //p 为指向 Clock 类型的指针
p = &clock1;                        //p 为指向 clock1 对象
Clock & clock3 = clock2;            //clock3 为对 clock2 的引用
```

3.3.2　对象的访问

对象的访问是指对对象成员的使用，可以通过成员运算符“.”和指向成员运算符“->”访问，针对对象、指针和引用，可以分以下三种情形。

1. 通过对象访问

格式为

```
对象名.数据成员
对象名.成员函数(实参表)
```

例如：

```
clock1.setClock(6,15,24);
clock1.hour = 6;                    //此处不正确，因为 hour 为私有的
```

2. 通过指针访问

格式为

```
指向对象的指针->数据成员
指向对象的指针->成员函数(实参表)
```

或者

```
(*指向对象的指针).数据成员
(*指向对象的指针).成员函数(实参表)
```

例如：

```
Clock * p;
p = &clock1;
```

“p->setClock(6,15,24);”和“(*p).setClock(6,15,24);”都是正确的。

3. 通过引用访问

例如：

```
Clock &clock3 = clock1;
```

“clock3.setClock(6,15,24);”等价于“clock1.setClock(6,15,24);”。

例 3.1 计算两点之间的距离。

```
//ch3_1.cpp
#include<iostream>
#include<cmath>
using namespace std;
class Point
{
public:
     //计算两点之间距离
    double distance(Point & p) {                //引用形参
        return sqrt((p.x-x)*(p.x-x)+(p.y-y)*(p.y-y));
    }
    void setX(double i){x=i;}
    void setY(double j){y=j;}
private:
    double x;
    double y;
};
int main(){
    Point p1,p2;
    p1.setX(2);p1.setY(2);
    p2.setX(5);p2.setY(6);
    cout<<p1.distance(p2);
    return 0;
}
```

例 3.1 中定义了一个点类 Point，其中，数据成员 x、y 为横纵坐标，函数成员 setX、setY、distance 用来设置 x 的值、设置 y 的值、计算两点间距离。在本例中还要注意以下两点。

（1）distance 函数的形参 p 是引用参数，这样 p 直接利用实参对象 p2，而不创建新的形参对象，程序效率更高。在“p1.distance(p2)”执行过程中，成员函数 distance 的函数体内，p 就是 main 函数中的 p2。

（2）在 distance 函数中的“p.x”和“p.y”通过对象名引用了私有成员，这是允许的，因为在 C++中，成员函数中可以访问本类对象的私有成员。distance 是 Point 类的成员函数，p 是 Point 类型对象，所以在 distance 函数体内可以访问“p.x”和“p.y”。

通常类的定义是由设计人员完成的，而类的实现是编码人员的职责。为了分离二者的工作，C++一般把类定义放在头文件中，而实现放在源文件中。对例 3.1 进行分离，如例 3.2 所示。

例 3.2 计算两点之间的距离（代码分离），分为三个文件。

头文件 point.h 中包含类的定义。

```
//point.h
class Point
{
 public:
        double distance(Point & p);
        void setX(double i);
        void setY(double j);
 private:
```

```
    double x;
    double y;
};
```

源文件 point.cpp 中包含类的实现。

```
//point.cpp
#include<iostream>
#include<cmath>
#include "point.h"
using namespace std;
double Point::distance(Point & p){
    return sqrt((p.x-x)*(p.x-x)+(p.y-y)*(p.y-y));
}
void Point::setX(double i){x=i;}
void Point::setY(double j){y=j;}
```

源文件 main.cpp 是对类的使用。

```
//ch3_2.cpp
#include<iostream>
#include<cmath>
#include "point.h"
using namespace std;
int main(){
    Point p1,p2;
    p1.setX(2);p1.setY(2);
    p2.setX(5);p2.setY(6);
    cout<<p1.distance(p2);
    return 0;
}
```

把类的设计、实现和使用完全分离开来，程序结构更加合理，便于项目的团队研发。另外，用同一类定义的对象，各对象的成员数据类型和成员函数类型完全相同，可以相互整体赋值，赋值运算符仍是“=”，当对象 A 赋值给另一个对象 B 时，对象 A 有所有数据成员都会逐位复制，两个对象的数据成员的值相同，彼此是相互独立的，各自都有自己的内存空间。

```
Clock clock1,clock2;
clock1.setClock(6,15,24);
clock2=clock1;
```

赋值后 clock2 的时、分、秒和 clock1 完全一样，为 6 时 15 分 24 秒。

注意：不属于同一个类的对象之间不能相互整体赋值，即使成员完全相同、类名不同的两个对象之间也不能相互赋值。

3.4 构造函数和析构函数

在 C++语言中，构造函数是用来为对象分配内存空间及初始化赋值；析构函数用来释放分配给对象的内存空间，完成用户指定的操作来做好善后工作。

3.4.1 构造函数

构造函数是一种特殊的函数，主要用来在创建对象时初始化对象，即为对象成员数据赋

初始值。共有以下三个作用。

(1) 为对象分配空间并初始化。

(2) 对数据成员赋值。

(3) 请求其他资源。

通常将构造函数声明为公有成员函数(不是必需的),构造函数的名字与类名相同,不能任意命名,构造函数不具有类型,无返回值,因而不能指定包括 void 在内的任何返回值类型。构造函数的定义与其他成员函数的定义一样可以放在类内或类外。

定义构造函数的语法格式为

```
构造函数名(形参):初始化列表
{
   函数体
}
```

其中,构造函数名与类同名;形参、初始化列表、函数体都可以省略。初始化列表是用来对成员进行初始化的,格式为

```
成员名(实参)[, 成员名(实参)…]
```

如对 Clock 类,定义如下构造函数。

```
Clock(int h, int m, int s):hour(h),minute(m){ second = s; }
```

数据成员 hour 和 minute 在初始化列表中用参数 h 和 m 初始化,数据成员 second 用参数 s 在函数体中赋值。

关于初始化列表应注意以下几点。

(1) 数组成员不能在初始化列表中初始化。

(2) static 数据成员不能在初始化列表中初始化。

(3) 非 static 的 const 数据成员必须在初始化列表中初始化。

构造函数是类的成员函数,具有一般成员函数的所有性质:可访问类的所有成员,可以是内联函数,可带有参数表,可带有默认的形参值,还可重载。

构造函数的调用方式是在定义对象时自动执行的特殊成员函数,用户不能显式调用构造函数。在类中定义了构造函数,编译系统自动在建立新对象的地方插入对构造函数调用的代码,在调用构造函数时,执行构造函数中定义的赋值语句,为成员变量赋初值,当然也可以有其他语句。实际上,调用构造函数的格式即为定义对象的语法格式:

```
类名   对象名(实参列表);
```

或

```
类名 *   对象名 = new 类名(实参列表);
```

注意,当构造函数无参数时,对象名后面不能加空的括号。

构造函数按照参数个数分为默认构造函数和有参构造函数两类。

1. 默认构造函数

默认构造函数就是在没有显式提供初始化式(定义的对象名后无实参表)时调用的构造函数。它是一种不带参数的构造函数,或者为所有的形参提供默认参数值的构造函数。

1）系统生成的默认构造函数

前面定义的类都没定义构造函数，当类中没有定义构造函数时，编译系统就会自动生成一个默认的构造函数，这个默认的构造函数不带任何参数，函数体为空，只能给对象开辟一个存储空间，不能为对象中的数据成员赋初值。系统自动生成的构造函数的形式为

类名::构造函数名(){ }

编译系统为前面的 Clock 类自动生成的构造函数是

```
Clock::Clock(){  }
```

完整的 Clock 类定义如下。

```
class Clock
{
private :
    int hour, minute, second;
public :
    Clock(){ }                  //系统自动生成的
    void Clock::setClock(int h, int m, int s);
    void Clock::showClock();
};
```

若有对象定义：

```
Clock clock1;                   //注意 clock1 后面无空括号
```

clock1 此时数据成员的值是由编译系统给定的未初始化的值。

2）自定义的默认构造函数

利用系统默认生成的构造函数，并不能起到给对象赋初值的作用，为此，可以显式定义构造函数，使得对象按照构造函数约定的值给对象赋值。

在 Clock 类内定义以下构造函数。

```
Clock(){hour = 0; minute  = 0; second = 0;}
```

若有对象定义：

```
Clock clock1;                   //注意 clock1 后面无空括号
```

clock1 此时数据成员 hour、minute、second 的值均为 0。

当然构造函数也可以在类内写声明，类外写实现，形式如下。

```
类内声明:
    Clock();
类外实现:
    Clock::Clock(){ hour = 0; minute  = 0; second = 0;}
```

对成员的赋值还可以不写在函数体里面，而写在初始化列表中，例如，在 Clock 类内声明并实现带初始化列表的构造函数：

```
Clock():hour(0), minute(0), second(0){  }
```

带初始化列表的构造函数也可以在类内写声明，类外写实现，形式如下。

```
类内声明:
    Clock();                    //注意:声明部分不包含初始化列表
类外实现:
```

```
Clock::Clock() :hour(0), minute(0), second(0){ }
```

对数据成员的赋值，写在初始化列表与函数体中的区别在于：写在初始化列表中等价于简单变量的定义并且初始化；写在函数体中等价于简单变量的先定义后赋值。因此，写在初始化列表中的效率较高。

3）全部带默认参数值的默认构造函数

全部带默认参数值的默认构造函数在有参构造函数中介绍。

2. 有参构造函数

不带参数的构造函数是用固定的数据对对象初始化，每个新生成的对象都有相同的数据，如果希望每个对象有不同的数据，就得再利用其他成员函数更改或输入数据。其实可以在生成对象时就使得每个对象拥有不同的数据值，这就需要用有参数的构造函数，用参数去给对象初始化或赋值。

将前面的 Clock 类改为定义有参数的构造函数，类定义如下。

```
class Clock
{
private :
    int hour, minute, second;
public :
    Clock(int h, int m, int s){hour = h; minute = m; second = s;}
    void Clock::setClock(int h, int m, int s);
    void Clock::showClock();
};
```

若有对象定义：

```
Clock clock1(1,2,3);
```

带有参数的构造函数在定义对象时指定实际参数，然后由该实际参数传递给构造函数的形式参数，进而对对象的数据成员赋值，因此生成的对象 clock1 的数据成员 hour、minute、second 的值分别为 1、2、3。

当然，构造函数也可以在类内写声明，类外写实现，形式如下。

```
类内声明:
    Clock(int h, int m, int s);
类外实现:
    Clock::Clock(int h, int m, int s){hour = h; minute = m; second = s;}
```

对成员的赋值还可以不写在函数体里面，而写在初始化列表上。

在 Clock 类内声明并实现带初始化列表的构造函数：

```
Clock(int h, int m, int s):hour(h), minute(m), second(s){  }
```

带初始化列表的构造函数也可以在类内写声明，类外写实现，形式如下。

```
类内声明:
    Clock(int h, int m, int s);       //注意:声明部分不包含初始化列表
类外实现:
    Clock::Clock(int h, int m, int s):hour(h), minute(m), second(s){  }
```

1）构造函数重载

在 Clock 类中只定义一个构造函数：

```
Clock(int h, int m, int s){hour = h; minute = m; second = s;}
```

若有对象定义：

```
Clock  clock1;
```

则编译时会出现错误，因为此时要调用默认构造函数，而系统并没有默认构造函数，因为当显式定义一个构造函数时，系统就不再生成默认构造函数 Clock(){ }。

为了使用 Clock 类定义对象时，既可以给参数，也可以不给参数，即：

```
Clock clock1(1,2,3);          //需要三个参数的构造函数
Clock clock2;                 //需要无参构造函数
```

需要既有三个参数的构造函数，也有无参数的构造函数，因此可以定义两个构造函数：

```
Clock(int h, int m, int s):hour(h), minute(m), second(s){  }
Clock():hour(0), minute(0), second(0){  }
```

这两个参数不同、函数名同为类名的函数即为构造函数的重载。

2）带默认参数的构造函数

在实际应用中，有些构造函数的参数值通常是不变的，只有在特殊情况下才需要改变参数的值。按照带有默认参数的函数思路，可以将构造函数定义成带默认参数值的构造函数，在定义对象时可以不指定实参，用默认参数值来赋值或初始化数据成员，在定义对象时指定实参就可以用指定的实参给对象数据成员赋值或初始化。

对于 Clock 类可以定义一个如下带默认参数的构造函数。

```
Clock(int h = 0, int m = 0, int s = 0){hour = h; minute = m; second = s;}
```

那么使用 Clock 类定义对象时，既可以给参数，也可以不给参数，即：

```
Clock clock0;              //clock0 的数据成员 hour、minute、second 的值为 0、0、0
Clock clock1(1);           //clock1 的数据成员 hour、minute、second 的值为 1、0、0
Clock clock2(1,2);         //clock2 的数据成员 hour、minute、second 的值为 1、2、0
Clock clock3(1,2,3);       //clock3 的数据成员 hour、minute、second 的值为 1、2、3
```

我们还可以看出，当所有参数都提供默认值时，该构造函数也是默认构造函数，如 clock0 的定义。

例 3.3 带有默认参数的构造函数。

```
//ch3_3.cpp
#include <iostream>
using namespace std;
class Point
{
private:
   double x, y;
public:
   Point(double x = 0.0, double y = 0.0);      //定义构造函数,它的名字与类名相同
   void disp();                                //输出私有变量的成员函数
};
Point:: Point(double a, double b)
  {  x = a; y = b;  }                          //初始化私有数据成员 x 和 y
void Point::disp()
   {  cout << x <<","<< y << endl;  }
```

```
int main(){
   Point p1(1.0,2.0),p2(3.0),p3;            //定义对象
   cout <<"p1 = ";
   p1.disp();
   cout <<"p2 = ";
   p2.disp();
   cout <<"p3 = ";
   p3.disp();
   return 0;
}
```

程序运行结果如下。

```
p1 = 1,2
p2 = 3,0
p3 = 0,0
```

3.4.2 析构函数

在类中定义的构造函数,在对象生命周期开始,编译系统会自动地执行构造函数完成对象内存空间的分配和数据的初始化工作。在类中定义的析构函数,在对象生命周期结束自动执行析构函数,完成清理内存工作,并可以执行指定的其他操作。

析构函数是一种在结束对象调用时自动执行的特殊的成员函数,一个类中只能定义一个析构函数。

析构函数声明为公有成员,析构函数名由“～”与类名组合而成,析构函数不接收参数(所以不能重载,只有一个析构函数),没有返回值,定义的语法格式如下。

```
class 类名
{
public:
    …
    ～类名()
    { 指定的操作;}
};
```

例如:

```
class Clock
{
private :
    int hour, minute, second;
public :
    Clock(int h, int m, int s){hour = h; minute = m; second = s;}
    ～Clock(){cout << "destructing";}
    void Clock::showClock(){
        cout << hour << minute << second;
    }
};
```

通常在构造函数中用 new 运算符为对象额外申请了一些空间(额外资源,不属于对象空间),在对象结束生命周期时需要使用析构函数,利用 delete 运算符释放 new 运算符所申请的空间。

当类中没有显式地定义析构函数时，则系统会自动生成一个默认的析构函数，该函数是一个空函数。默认的析构函数的格式如下。

类名 :: ～类名(){　　}

例 3.4　构造函数与析构函数。

```
//ch3_4.cpp
#include <iostream>
#include <cstring>
using namespace std;
class Point
{
private:
   double x, y;                          //x,y 坐标
   char *name;                           //点的名称
public:
//声明构造函数,它的名字与类名相同
   Point(const char *n = NULL, double x = 0.0, double y = 0.0);
   ～Point();
   void disp();                          //输出私有变量的成员函数
};
Point::Point(const char *n, double a, double b){
    x = a; y = b;
    if(n) {                              //如果形参不是默认的空值
        name = new char[strlen(n) + 1];
        strcpy(name,n);
    }
    else{
        name = new char[8];
        strcpy(name,"no name");
    }
    cout << name <<" constructing"<< endl;
}
Point::～Point(){
    cout << name <<" destructing"<< endl;
    delete [] name;
}
void Point::disp(){ cout << name <<":"<< x <<","<< y << endl; }
int main(){
   //定义对象
   Point p1("home",1.0,2.0);
   Point p2("school",3.0);
   Point p3;
   //输出对象
   cout <<"p1 = ";
   p1.disp();
   cout <<"p2 = ";
   p2.disp();
   cout <<"p3 = ";
   p3.disp();
   return 0;
}
```

程序运行结果如下。

```
home constructing
school constructing
no name constructing
p1 = home:1,2
p2 = school:3,0
p3 = no name:0,0
no name destructing
school destructing
home destructing
```

从这个例子可以看出,对象析构的顺序恰好和对象的构造顺序相反。

3.5 拷贝构造函数

一个简单变量可以在定义的同时进行初始化,例如:

```
int i = 100;
```

该语句的含义是开辟一个整型变量空间同时置其值为 100。

对于一个对象,也可以定义用一个已经存在的对象进行初始化。这种初始化的方式需要调用拷贝构造函数来实现。若不定义拷贝构造函数,则系统自动生成默认的拷贝构造函数,把已经存在对象的数据按位复制到新生成对象的空间,但是这种做法有时会出现问题。

例如,对于例 3.4,画出对应的存储状态如图 3.3 所示。

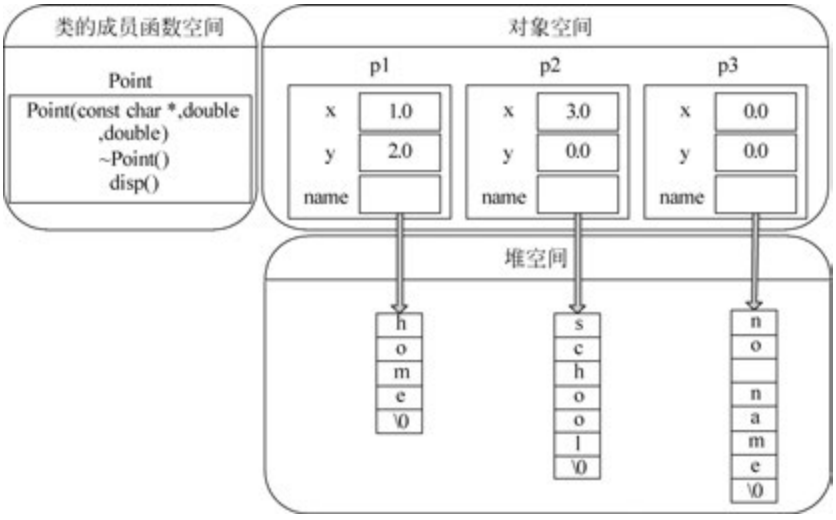

图 3.3 申请资源的对象存储

需要注意的是,堆空间并不属于对象空间,利用 sizeof 计算对象所占空间大小时只包含 x、y、name 的空间。另外,若对对象进行复制或赋值操作,也只是处理对象空间。例如,若在例 3.4 的 main 函数中 p3 对象定义之后增加一行代码:

```
Point p4 = p1;
```

则对应存储状态如图 3.4 所示。

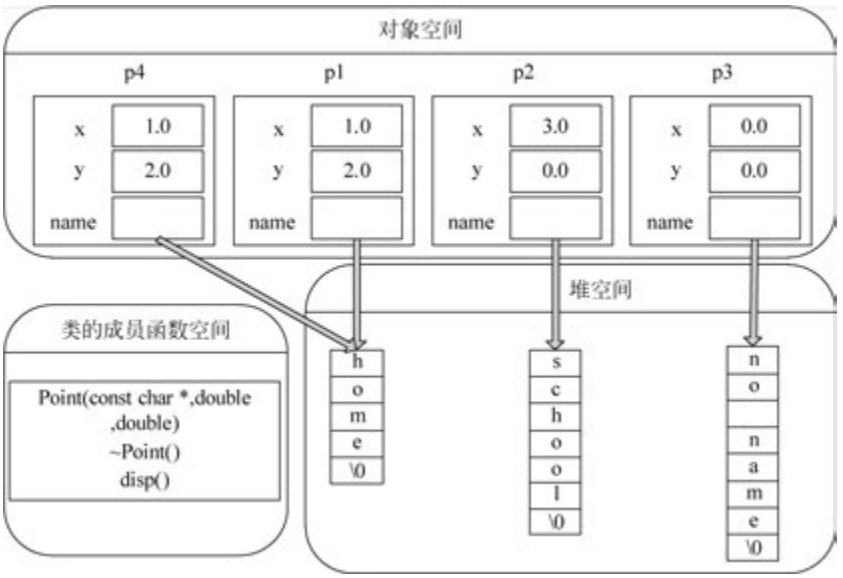

图 3.4　申请资源的对象存储

由 p1 复制 p4 时，只复制对象空间，二者有相同的 x、y、name。这样 p1 和 p4 的 name 都指向同一块堆空间。当析构时，p4 先析构，会释放掉 home 的空间，当 p1 析构时，再释放空间就错误了，运行时会出现内存访问错误。因此，此时不能够完全采用默认的按位复制方法，需要自定义拷贝构造函数，按照自己的定义的规则进行复制。

拷贝构造函数名与类名相同，参数是本类对象的引用，拷贝构造函数没有返回值。定义拷贝构造函数的语法格式为

```
class 类名
{
public:
  类名(const 类名 & 形参)
  {  拷贝构造函数的函数体     }
        …
};
```

当然，函数的实现也可以写在类定义的外面。

其中，形参是用来初始化新对象的对象的常引用(引用是必需的，否则会产生递归的拷贝构造过程)。

当用一个已经存在的对象初始化本类的新对象时，如果没有定义拷贝构造函数，则系统会自动生成一个默认的拷贝构造函数来完成初始化的工作，默认的拷贝构造函数完全按照已存在对象复制一个完全相同的新对象。默认的拷贝构造函数格式为

```
类名(const  类名 & 形参){     }
```

构造函数是当定义一个新对象时自动调用的函数，是一个从无到有的过程，但是拷贝构造函数需要利用一个已经存在的对象再生成一个新的对象，也就是由原有的对象复制生成一个新的对象。例如：

```
Point p1("home",1.0,2.0);
Point p2 = p1;                          //也可以写作 Point p2 (p1);
```

对象 p1 生成时没有参照任何现有对象，而对象 p2 生成是参照 p1 生成的，也称作用 p1 初始化 p2。

注意，如下语句未调用拷贝构造函数。

```
Point p1("home",1.0,2.0);
Point p2;
p2 = p1;
```

p2 并不是由 p1 参照生成的，因为 p2 是在第 2 行生成的，生成时没有参照任何对象。第 3 行是由 p1 给已经存在的 p2 进行赋值，并不是拷贝构造。

3.5.1 浅拷贝与深拷贝

根据拷贝构造的规则，拷贝构造函数分为浅拷贝和深拷贝。浅拷贝就是把对象的成员数据一一赋值(编译系统一般采用按位复制)。但是可能会有这样的情况：对象还使用一些资源，这里的资源可以是堆资源(如例 3.4 中的 name 所指向的 new 申请来的空间)，或者一个文件。当浅拷贝的时候，两个对象就有用共同的资源，同时对资源可以访问，这样可能会出问题。如图 3.4 所示的就是一个浅拷贝的情形，p1 到 p4 复制只是浅层次的复制对象空间(x、y、name)，并未将 p1 的 name 所指向的堆空间复制一份给 p4，因此称为浅拷贝。当 p4 析构后，p1 再析构就出问题了，原因是拷贝构造时 p4 和 p1 的 name 共享同一个资源空间。

深拷贝就是用来解决共用资源的问题的，它把资源也复制一份，使生成对象拥有和原对象不同的资源，但资源的内容是一样的。对于堆资源来说，就是再开辟一片堆内存，把原来的堆内容也复制到新开辟的堆内存，这是深层次的拷贝，因此称为深拷贝。深拷贝需要自定义拷贝构造函数，来指定复制的规则，见例 3.5。

例 3.5 深拷贝构造。

```
//ch3_5.cpp
#include <iostream>
#include <cstring>
using namespace std;
class Point
{
 private:
   double x, y;                    //x,y 坐标
   char *name;                     //点的名称
 public:
//声明构造函数,它的名字与类名相同
   Point(const char *n = NULL, double x = 0.0, double y = 0.0);
   Point(const Point &p);
   ~Point();
   void disp();                    //输出私有变量的成员函数
};
Point::Point(const char *n, double a, double b){
    x = a; y = b;
    if(n) {                        //如果形参不是默认的空值
          name = new char[strlen(n) + 1];
          strcpy(name,n);
    }
    else{
```

```
        name = new char[8];
        strcpy(name,"no name");
    }
    cout << name <<" constructing"<< endl;
}
Point::Point(const Point &p){    //拷贝构造函数
    x = p.x; y = p.y;
    if(p.name) {                 //如果形参不是默认的空值
        name = new char[strlen(p.name) + 1];
        strcpy(name,p.name);
    }
    else{
        name = new char[8];
        strcpy(name,"no name");
    }
    cout << name <<" copy constructing"<< endl;
}
Point::~Point(){
    cout << name <<" destructing"<< endl;
    delete [] name;
}
void Point::disp(){ cout << name <<":"<< x <<","<< y << endl; }
int main(){
   //定义对象
   Point p1("home",1.0,2.0);
   Point p2("school",3.0);
   Point p3;
   Point p4 = p1;
   //输出对象
   cout <<"p1 = ";
   p1.disp();
   cout <<"p2 = ";
   p2.disp();
   cout <<"p3 = ";
   p3.disp();
   return 0;
}
```

例 3.5 的拷贝构造函数中对 x、y 进行赋值，所以 p4 的 x、y 与 p1 的 x、y 完全相同，但是指针 name 的值是不同的，p4 的 name 指向一个新申请的空间，这样当析构时，p1 和 p4 会各自释放掉自己的堆空间。例 3.5 的存储状态如图 3.5 所示。

浅拷贝只是拷贝对象空间，深拷贝可以复制对象空间之外申请的资源，到底是利用深拷贝还是利用浅拷贝，并不是只取决于时间效率、空间效率等，而是取决于哪一个逻辑上是正确的。

3.5.2 标记拷贝构造

对于深拷贝，需要把资源复制一份，而有时候资源很大，复制需要大量的时间和空间，甚至有些资源是不可复制的，这时深拷贝是不可取的，为此，可以采用带标记的拷贝构造。

带标记的拷贝构造需要区分哪些对象是原始生成的，哪些对象是拷贝构造函数生成的。原始生成的对象申请了额外的资源，拷贝构造函数生成的对象不再申请额外资源，去共享原

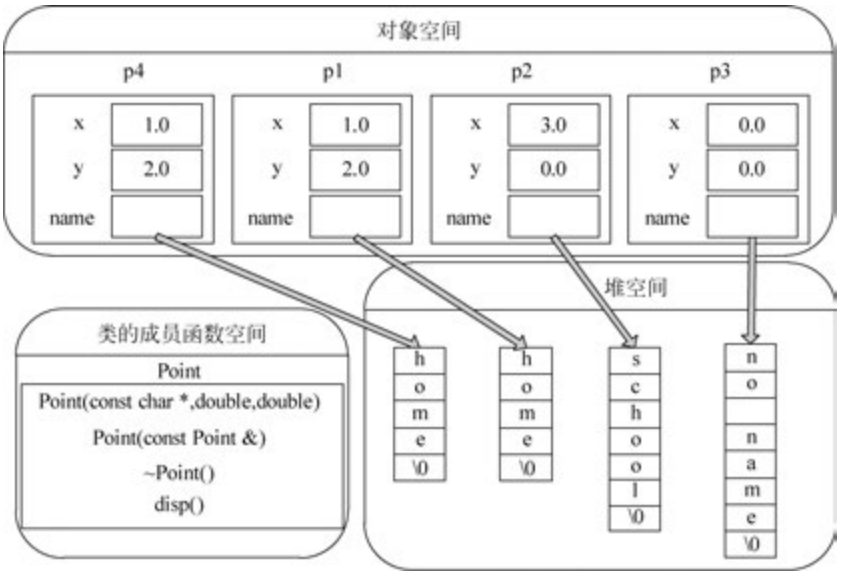

图 3.5　深拷贝构造

有对象申请的资源，但是析构时拷贝构造函数生成的对象不可以释放额外资源，原始生成的对象析构时才可以释放额外资源，因此析构函数中需要区别原始生成的对象和拷贝构造对象。为此，可以给类增加一个标记成员，用以区别原始生成的对象和拷贝构造对象，见例 3.6。

例 3.6　标记拷贝构造。

```
//ch3_6.cpp
#include <iostream>
#include <cstring>
using namespace std;
class Point
{
private:
   double x, y;                    //x,y 坐标
   char *name;                     //点的名称
   int flag;                       //区分构造对象和拷贝构造对象
public:
//声明构造函数,它的名字与类名相同
   Point(const char *n = NULL, double x = 0.0, double y = 0.0);
   Point(const Point &p);
   ~Point();
   void disp();                    //输出私有变量的成员函数
};
Point::Point(const char *n, double a, double b){
    x = a; y = b;
    flag = 1;                      //flag 为 1 时代表构造生成对象
    if(n) {                        //如果形参不是默认的空值
        name = new char[strlen(n) + 1];
        strcpy(name,n);
    }
    else{
        name = new char[8];
        strcpy(name,"no name");
```

```
    }
    cout << name <<" constructing"<< endl;
}
Point::Point(const Point &p){
    x = p.x; y = p.y;
    name = p.name;
    flag = 0;                          //flag 为 0 时代表拷贝构造生成对象
    cout << name <<" copy constructing"<< endl;
}
Point::~Point(){
    cout << name <<" destructing"<< endl;
    if(flag)
        delete [] name;
}
void Point::disp(){ cout << name <<":"<< x <<","<< y << endl; }
int main(){
   //定义对象
   Point p1("home",1.0,2.0);
   Point p2("school",3.0);
   Point p3;
   Point p4 = p1;
   //输出对象
   cout <<"p1 = ";
   p1.disp();
   cout <<"p2 = ";
   p2.disp();
   cout <<"p3 = ";
   p3.disp();
   return 0;
}
```

例 3.6 的内存状态如图 3.6 所示，p4 和 p1 的 name 共享同一段堆资源空间，但是类增加了一个 flag 成员，构造时对其赋值为 1，拷贝构造时对其赋值为 0，然后析构时根据 flag 的值确定是否利用 delete 释放堆资源空间，保证了堆资源空间只释放一次。

3.5.3 函数参数与返回值

拷贝构造函数是当利用现有的对象生成新的对象时调用，除了前面介绍的对象初始化时会调用拷贝构造函数，还有实参对象给形参对象传值时和函数返回对象时也是利用现有对象生成新的对象，也要调用拷贝构造函数。

1. 函数对象参数

当函数的参数为对象时，要利用现有的实参对象生成形参对象，这时候要调用拷贝构造函数。

例 3.7 函数对象参数拷贝构造。

```
//ch3_7.cpp
#include <iostream>
#include <cstring>
using namespace std;
//Point 类定义和实现
class Point
```

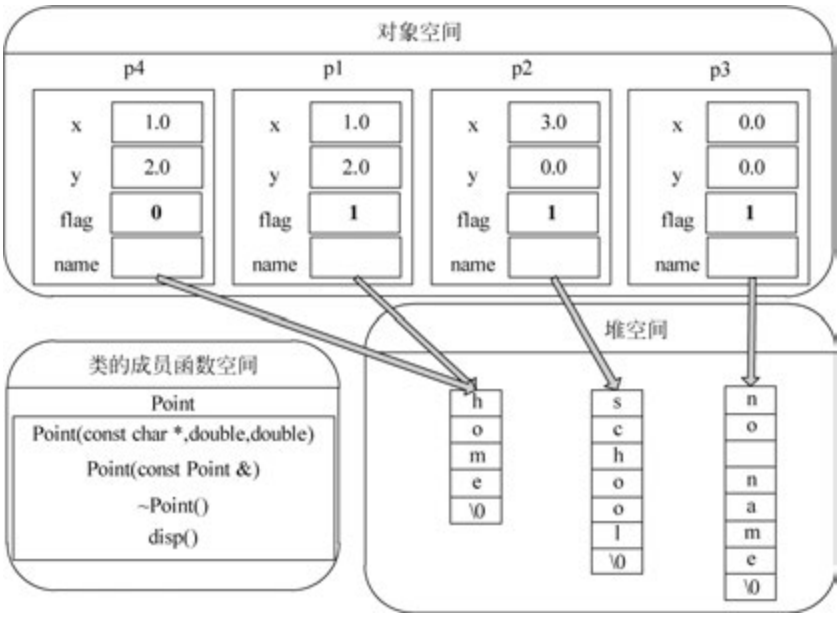

图 3.6　标记拷贝构造

```
{
 private:
   double x, y;                    //x,y 坐标
   char * name;                    //点的名称
 public:
//声明构造函数,它的名字与类名相同
   Point(const char * n = NULL, double x = 0.0, double y = 0.0);
   ~Point();
   void disp();                    //输出私有变量的成员函数
};
Point::Point(const char * n, double a, double b){
    x = a; y = b;
    if(n) {                        //如果形参不是默认的空值
          name = new char[strlen(n) + 1];
          strcpy(name,n);
    }
    else{
        name = new char[8];
        strcpy(name,"no name");
    }
    cout << name <<" constructing"<< endl;
}
Point::~Point(){
    cout << name <<" destructing"<< endl;
    delete [] name;
}
void Point::disp(){ cout << name <<":"<< x <<","<< y << endl; }
//test 函数
void test(Point p){
    p.disp();
}
```

```
//main 函数
int main(){
    Point p1("home",1.0,2.0);
    test(p1);
    return 0;
}
```

例 3.7 的存储状态如图 3.7 所示。具体的执行过程是这样的：首先执行 main 函数，开辟一个 p1 的空间，以 p1 为实参调用 test 函数；然后开始执行 test 函数，用实参 p1 拷贝构造生成形参 p(因为没有定义拷贝构造函数，所以按位复制)，执行 p 的 disp()方法，test 函数结束，释放局部变量形参，即执行 p 的析构函数释放堆空间；最后返回到 main 函数，main 函数结束时，释放 main 函数中的局部变量对象 p1，即执行 p1 的析构函数去释放堆空间，然而对应堆空间已经被 test 函数的 p 释放了，因此程序出现了错误。为此，可以自定义深拷贝构造函数或标记拷贝构造函数解决。

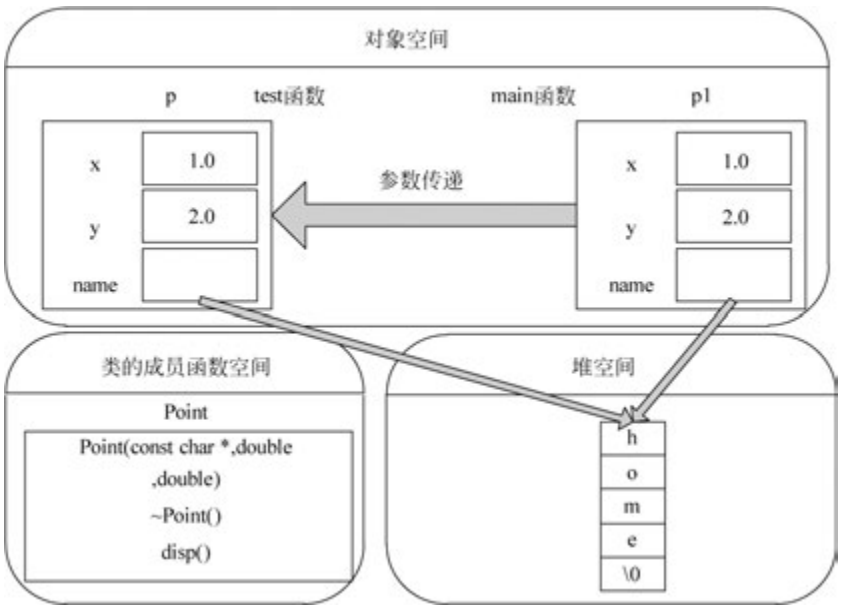

图 3.7　函数对象参数——浅拷贝

例如，对 Point 类增加一个深拷贝构造函数：

```
Point::Point(const Point &p)
{
    x = p.x; y = p.y;
    if(p.name) {                    //如果形参不是默认的空值
          name = new char[strlen(p.name) + 1];
          strcpy(name,p.name);
    }
    else{
        name = new char[8];
        strcpy(name,"no name");
    }
    cout << name <<" copy constructing"<< endl;
}
```

则对应存储状态如图 3.8 所示，test 函数中形参 p 的 name 和 main 函数中实参 p1 的 name 各自指向不同堆空间，析构时就不会出现问题。

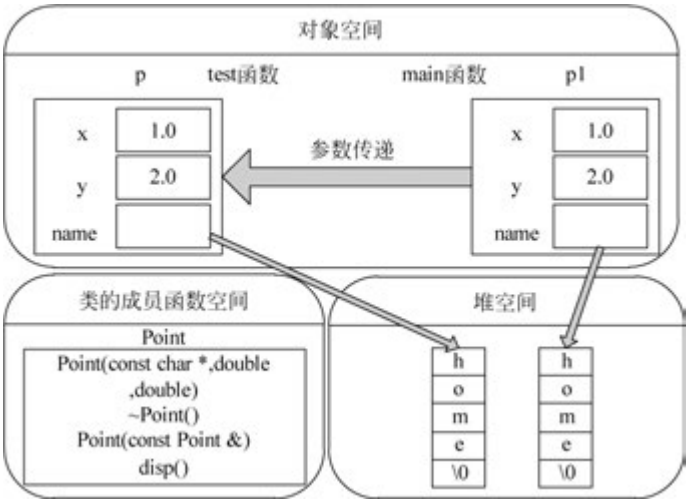

图 3.8　函数对象参数——深拷贝

2. 函数返回对象

当函数的返回值为对象值时，如函数内定义对象 Point p；当函数返回时 return p;。

p 对象超出作用区域(函数)，这时会调用拷贝构造函数创建该 p 对象的一个临时复制对象，并把它赋给(调用语句不同、编译器不同赋给方式会不同)主调函数中需要的对象；然后 p 的析构函数释放对象占用的内存资源，接着再调用临时复制对象的析构函数(优化的编译器有可能此时不释放)释放这个对象占用的内存资源。

把例 3.7 中的 test 函数和 main 函数修改为如下。

```
//test 函数
Point test()
{
    Point p("home",1.0,2.0);
    return p;
}
//main 函数
int main()
{
    Point p1;
    p1 = test();
    return 0;
}
```

main 函数的 p1 先定义后，用 test 函数返回的临时对象赋值(按位复制对象空间)，如图 3.9 所示。对象的析构顺序是 p、临时对象、p1，p 析构时释放掉堆空间，然后临时对象析构再释放堆空间时就错了。

把 main 函数中的 p1 改为由临时对象进行初始化(如下面的代码所示)，此时会有两次拷贝构造过程：第一次是 test 函数的“return p;”语句，由 p 拷贝构造生成临时对象；第二次是“Point p1＝test();”语句，由临时对象拷贝构造生成 p1。然而，很多编译器都会优化，

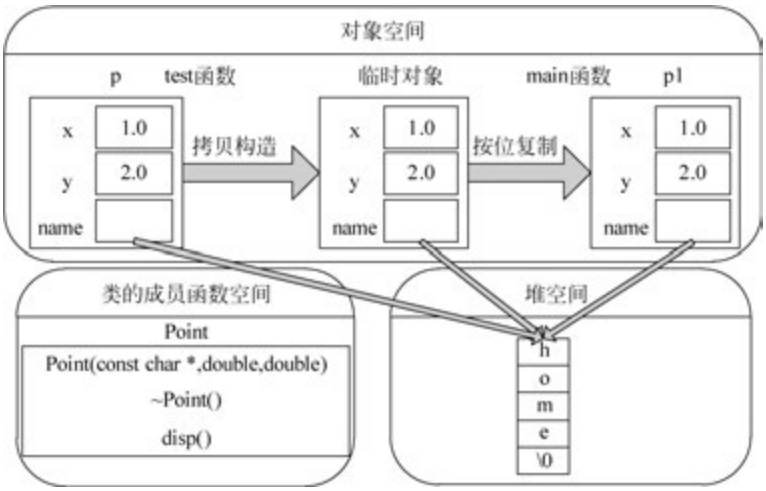

图 3.9　函数返回对象——浅拷贝(复制)

第二次拷贝构造不会执行,而是 p1 直接利用临时对象(既不构造,也不拷贝构造),如图 3.10 所示。对象的析构顺序是 p、p1(即临时对象),p 析构时释放掉堆空间,然后,p1(即临时对象)析构再释放堆空间时就错了。

```
//main 函数
int main()
{
    Point p1 = test();
    return 0;
}
```

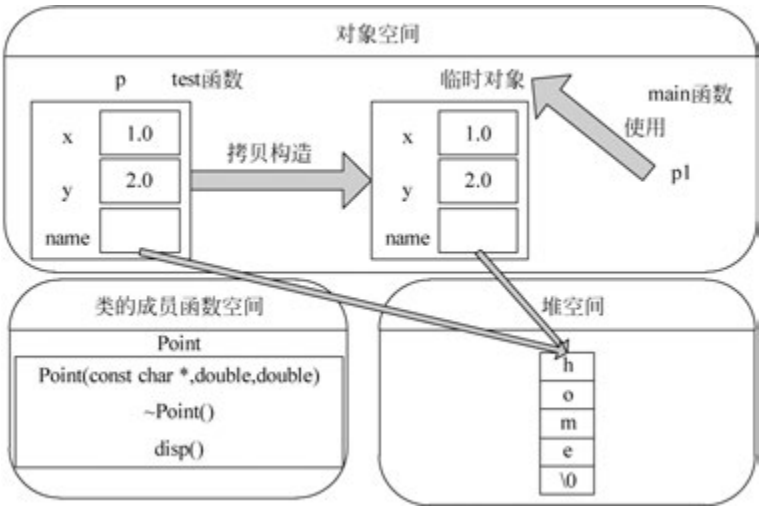

图 3.10　函数返回对象——浅拷贝(初始化)

要解决上述问题,只需要给 Point 类定义深拷贝构造函数即可。

通过上面的代码可以看出,如果一个对象利用了额外的资源,并且在析构函数中定义了释放额外资源的功能,若未定义深拷贝构造函数或标记的拷贝构造函数,当对象作函数参数或者函数返回对象时都会出现问题。

3.6 对象数组

数组元素不仅可以由简单类型组成(例如,整型数组的每一个元素都相当于一个整型变量),也可以由对象组成(对象数组的每一个元素都是同类型的对象)。

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30个学生,每个学生的属性包括姓名、年龄、成绩等。如果为每一个学生建立一个对象,不采用数组,需要分别取30个对象名,不便于程序用循环处理。这时可以定义一个“学生类”类型对象数组,每一个数组元素是一个“学生类”对象。

例如,有学生类定义:

```
class Student{
private:
    char name[20];
    int age;
    float score;
};
```

定义stud数组,有30个元素,每个元素都是学生对象。

```
Student stud[30];
```

在建立数组时,同样要调用构造函数。例如,stud有30个元素,需要调用30次构造函数,调用的都是系统自动生成的默认构造函数。当定义数组不初始化时一定要保证有默认构造函数。

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。

例如,有学生类定义:

```
class Student{
private:
    char name[20];
    int age;
    float score;
public:
    Student(int s){
        strcpy(name,"no name");
        age = 20;
        score = s;
    }
};
```

数组定义:

```
Student stud[3] = {60,70,78};
```

这种定义是合法的,三个实参分别传递给三个数组元素的构造函数。stud[0]、stud[1]、stud[2]的score分别为60、70、78。

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。因此，如果构造函数有多个参数，在定义对象数组时应当在花括号中分别写出构造函数并指定实参。如果构造函数有三个参数，分别代表姓名、年龄、成绩。例如，有学生类定义：

```
class Student{
private:
    char name[20];
    int age;
    float score;
public:
    Student(const char * n, int a, int s){
        strcpy(name,n);
        age = a;
        score = s;
    }
};
```

则可以这样定义对象数组：

```
Student stud[3] = { Student("zhang san",20,60), Student("Li si",19,70), Student("Wang wu",18,
78) };
```

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括起来，对应构造函数的一组形参，不会混淆。

例 3.8 计算一组学生的总成绩和平均成绩。

```
//ch3_8.cpp
#include<iostream>
#include<cstring>
using namespace std;
class Student{
private:
    char name[20];
    int age;
    float score;
public:
    Student(const char *n, int a, int s){
        strcpy(name,n);
        age = a;
        score = s;
    }
    float getScore(){            //返回成绩值
        return score;
    }
};
int main(){
    Student stud[3] = { Student("zhang san",20,60), Student("Li si",19,70),
    Student("Wang wu",18,78) };
    float sum = 0,average;
    int i;
    for(i = 0;i<3;i++)
        sum += stud[i].getScore();
    average = sum/3;
    cout <<"sum = "<< sum <<"   average = "<< average << endl;
```

```
    return 0;
}
```

程序运行结果如下。


```
sum = 208   average = 69.3333
```

3.7 this 关 键 字

每个对象有自己独立的数据空间，但是类的成员函数只存储一份，为所有对象共享，当通过对象调用非静态成员函数（静态情况在 3.8 节介绍）时，需要把调用对象的地址也传递给成员函数，以确定成员函数要处理的数据是哪一个对象的数据，成员函数通过 this 指针接收调用对象的地址，所以每个成员函数（非静态的）都有一个隐含的指针变量 this。例如，Clock 类定义及实现：

```
class Clock
{
private :
   int hour, minute, second;              //关于时间的数据
public :
   void setClock(int h, int m, int s);    //调整时间值
};
void Clock::setClock(int h, int m, int s){
  hour = h;
  minute = m;
  second = s;
}
```

相当于在函数形式参数中有一个 this 指针，即：

```
void Clock::setClock(Clock *this, int h, int m, int s)   //形式参数中 this 不能够显式写出来
{
    this->hour = h;
    this->minute = m;
    this->second = s;
}
```

对于对象定义：

```
Clock c1,c2;
c1.setClock(1, 2, 3);
```

c1 对 setClock 函数的调用等价于：

```
setClock(&c1,1,2,3);          //注意程序中不能够这样写
```

这样 this 获得 c1 的地址，函数体中的“this-> hour＝h;”即给 c1 的 hour 赋值。

this 指针的用途之一是，很多程序员习惯上把形式参数与类数据成员命名相同的名字，这样每个形式参数的含义将一目了然。例如：

```
void Clock::setClock(int hour, int minute, int second) {
    this->hour = hour;
    this->minute = minute;
    this->second = second;
}
```

在 setClock 函数中 this-> hour、this-> minute、this-> second 代表调用对象的数据成员，hour、minute、second 代表形式参数。

可以看出，在类的非静态成员函数中访问类的非静态成员数据的时候，编译器会自动将对象本身的地址作为一个隐含参数传递给函数。也就是说，即使没有写上 this 指针，编译器在编译的时候也会加上 this 的，它作为非静态成员函数的隐含形参，对各成员的访问均通过 this 进行。

还有一种情况就是，在类的非静态成员函数中返回类调用对象本身的时候，直接使用“return * this”，在第 4 章运算符重载部分会有应用。

3.8 static 成 员

static 是 C++中很常用的修饰符，它可以修饰函数、局部变量和全局变量，也可以修饰类的成员，用来控制成员的存储方式和可见性。

3.8.1 static 数据成员

在类内数据成员的声明前加上关键字 static，该数据成员就是类内的静态数据成员，静态数据成员也被称作类的成员。无论这个类的对象被定义了多少个，静态数据成员在程序中也只有一份拷贝，由该类型的所有对象共享访问。也就是说，静态数据成员是该类的所有对象所共有的。对该类的多个对象来说，静态数据成员只分配一次内存，供所有对象共用。所以，静态数据成员的值对每个对象都是一样的，而非静态数据成员，每个类对象都有自己的拷贝。

对于类 X 的定义及对象 a、b、c、d 的定义：

```
class X {
public:
   char ch;
   static int s;
 };
 int X::s = 0;
 void f()
 { X  a, b, c, d ; }
```

存储方式如图 3.11 所示。

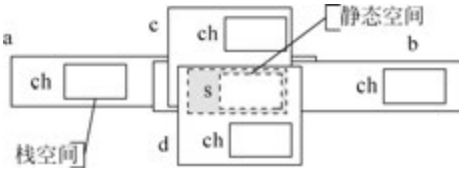

图 3.11 静态数据成员存储示意图

由图 3.11 可以看出，对象 a、b、c、d 各有自己的非静态数据成员空间 ch，但是它们共享一个静态数据成员 s。静态数据成员存储在全局数据区，静态数据成员要在程序一开始运行时就必须存在，静态数据成员定义时要分配空间，因此，需要类外初始化静态数据成员，即使静态数据成员是私有的也要类外初始化。静态数据成员初始化与一般数据成员初始化不

同,静态数据成员初始化的格式为

<数据类型><类名>::<静态数据成员名>=<值>

例如:

```
int X::s = 0;
```

初始化在类体外进行,而前面不加 static,以免与一般静态变量或对象相混淆;初始化时不加该成员的访问权限控制符 public、protected、private 等。静态数据成员的类外初始化等同于成员函数的类外实现,若未在类外初始化,程序连接时会出现错误。由于静态数据成员存储在全局数据区,所以 sizeof 运算符不包含静态数据成员的空间。

静态数据成员的存储空间实际上不依赖于对象而只依赖于类,换句话说,即使不定义任何对象,静态数据成员空间也照样存在,因此可以通过对象访问静态数据成员,也可以通过类访问静态数据成员。因此,类的静态数据成员有以下两种访问形式。

<类对象名>.<静态数据成员名>或<类类型名>::<静态数据成员名>

例如,对图 3.11 中静态数据成员 s 可以有 5 种访问办法:X::s(没有定义对象 a、b、c、d,X::s 也是正确的)、a.s、b.s、c.s、d.s,它们都代表访问同一个变量空间,通过一个改变值,其他方式获取的值全部改变。但是,静态数据成员和普通数据成员一样遵从 public、protected、private 访问规则,若上述 X 的静态数据成员是私有的,X::s、a.s、b.s、c.s、d.s 访问 s 都是错误的,因为不可见。把例 3.8 改用静态数据成员实现,见例 3.9。

例 3.9 利用静态数据成员计算一组学生的总成绩和平均成绩。

```
//ch3_9.cpp
#include<iostream>
#include<cstring>
using namespace std;
class Student{
private:
    char name[20];
    int age;
    float score;
public:
    Student(const char *n, int a, int s){
        strcpy(name,n);
        age=a;
        score=s;
        sum+=score;                    //构造同时把 score 累加到静态成员 sum
    }
    static float sum;                  //声明静态数据成员
};
float Student::sum=0;                  //初始化静态数据成员
int main(){
    Student stud[3]={ Student("zhang san",20,60), Student("Li si",19,70), Student("Wang
wu",18,78) };
    float average;
    average=Student::sum/3;            //类名访问静态成员,用对象名 stud[1].sum 是一样的
    cout<<"sum="<<Student::sum<<"  average="<<average<<endl;
    return 0;
}
```

程序运行结果如下。

```
sum = 208   average = 69.3333
```

静态数据成员通常用在各个对象都有相同的某项属性的时候。例如，对于一个存款类，每个实例的利率都是相同的，所以，应该把利率设为存款类的静态数据成员。这样做有两个好处：第一，不管定义多少个存款类对象，利率数据成员都共享分配在全局数据区的内存，所以节省存储空间；第二，一旦利率需要改变时，只要改变一次，则所有存款类对象的利率全改变过来了。

同全局变量相比，使用静态数据成员有两个优点：一是静态数据成员没有进入程序的全局名字空间，因此不存在与程序中其他全局名字冲突的可能性；二是可以实现信息隐藏，静态数据成员可以是 private 成员，而全局变量不能。

3.8.2 static 函数成员

与静态数据成员一样，还可以创建一个静态成员函数，它为类的全部服务而不是只为某个类的具体对象服务。静态成员函数与静态数据成员一样，都属于类定义的一部分。静态成员函数声明时前面加 static 关键字，在类外写函数实现时不需要关键字 static，调用静态成员函数，可以用成员访问操作符(.)和(->)为一个类的对象或指向类对象的指针调用静态成员函数，也可以直接使用作用域运算符，格式如下。

<类名>::<静态成员函数名>(<参数表>)

把例 3.9 改用静态成员函数实现见例 3.10。

例 3.10 利用静态函数成员计算一组学生的总成绩和平均成绩。

```
//ch3_10.cpp
#include<iostream>
#include<cstring>

using namespace std;
class Student{
private:
    char name[20];
    int age;
    float score;
public:
    Student(const char *n, int a, int s){
        strcpy(name,n);
        age = a;
        score = s;
        sum += score;                          //构造同时把 score 累加到静态成员 sum
    }
    static float sum;                          //声明静态数据成员
    static float getAverage();                 //声明静态函数成员
    void display();                            //输出总和与平均值
};
float Student::getAverage(){return sum/3;}     //实现时函数不需要用 static 修饰
float Student::sum = 0;                        //初始化静态数据成员
void Student::display(){                       //成员函数引用静态成员
    cout <<"sum = "<< sum <<"   average = "<< getAverage()<< endl;
```

```
}
int main(){
    //静态成员函数、静态成员数据不定义对象用类可以访问
    cout <<"sum = "<< Student::sum <<"  average = "<< Student::getAverage()<< endl;
    Studentstud[3] = { Student("zhang san",20,60), Student("Li si",19,70), Student("Wang
wu",18,78) };
    //静态成员函数、静态成员数据定义对象后,通过类访问
    cout <<"sum = "<< Student::sum <<"  average = "<< Student::getAverage()<< endl;
    //静态成员函数、静态成员数据定义对象后,通过对象访问
    cout <<"sum = "<< stud[0].sum <<"  average = "<< stud[0].getAverage()<< endl;
    //成员函数
    stud[0].display();
    return 0;
}
```

程序运行结果如下。

```
sum = 0   average = 0
sum = 208   average = 69.3333
sum = 208   average = 69.3333
sum = 208   average = 69.3333
```

在例 3.10 中可以看出,没有定义 Student 对象时,也可以通过类引用静态成员;成员函数 display 可以引用静态成员函数 average 和静态数据成员 sum,反之则不然。

普通的成员函数一般都隐含一个 this 指针,因为普通成员函数总是被某个具体对象调用的,this 即为该对象的地址。但是与普通函数相比,静态成员函数由于不是与任何的对象相联系,因此它不具有 this 指针。从这个意义上讲,它无法访问属于类对象的非静态数据成员,也无法访问非静态成员函数,它只能调用其他的静态成员函数。也就是说,静态成员之间可以相互访问,包括静态成员函数访问静态数据成员和访问静态成员函数,但是静态成员函数不能访问非静态成员函数和非静态数据成员,而非静态成员函数可以任意地访问静态成员函数和静态数据成员。

3.9 const 成员和 const 对象

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,把有关的数据定义为常量。

3.9.1 const 数据成员

类中某些数据成员的值是不需要改变的,如数学类的 π、物理类的 g 等都是常量,可以用关键字 const 来声明常数据成员。因为常数据成员的值是不能改变的,所以,定义对象时必须初始化,需要通过构造函数的初始化列表对常数据成员进行初始化。例如,在 Clock 类体中定义了常数据成员 hour:

```
class Clock
{
```

```
private :
    const int hour;                          //常量数据
    int minute, second;                      //非常量数据
public :
    Clock(int h, int m, int s):hour(h),minute(m),second(s){ }
};
```

必须采用在构造函数中对常数据成员 hour 初始化的方法，不能在构造函数的函数体中对常数据成员赋值。

```
Clock(int h, int m, int s):hour(h) {minute = m; second = s;}
```

是可以的，minute、second 是非常量成员，可以在构造函数体中赋值。

```
Clock(int h, int m, int s) {hour = h; minute = m; second = s;}
```

是不可以的，hour 必须在构造函数初始化列表中初始化。

```
Clock(int m, int s) {minute = m; second = s;}
```

hour 不初始化也是不可以的，hour 必须在构造函数初始化列表中初始化。

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；成员函数可以引用本类中的 const 数据成员，但不可以修改它们。

3.9.2 const 函数成员

定义的类的成员函数中，常常有一些成员函数不改变类的数据成员（如打印函数），也就是说，这些函数是“只读”函数，把不改变数据成员的函数加上 const 关键字进行标识，标识为常成员函数。声明常成员函数的格式如下。

类型 成员函数名(参数表) const;

例如：

```
class Clock
{
private :
    int hour, minute, second;
public :
    void setClock(int h, int m, int s){
        hour = h;
        minute = m;
        second = s;
    }
    void showClock() const{                  //常成员函数
        cout << hour << minute << second;
    }
};
```

常成员函数提高了程序的可读性，更重要的是它还能提高程序的可靠性，即已定义成 const 的成员函数，一旦试图修改数据成员的值，则编译器按错误处理。例如：

```
void Clock::showClock() const{               //常成员函数
    minute++;                                //改变数据成员，编译出错
    cout << hour << minute << second;
}
```

在常成员函数中语句“minute++;”改变数据成员,编译出错。

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,只要在常成员函数中不改变数据成员即可。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,只要 const 数据成员不被修改即可。

常成员函数不能调用另一个非常成员函数,因为非常成员函数是可以改变数据成员的,这样常成员函数就间接改变了数据,违背了常成员函数的定义规则。

例 3.11 圆类。

```
//ch3_11.cpp
#include<iostream>
using namespace std;
class Circle{
private:
    double x,y;                          //圆心坐标
    double radius;                       //半径
    const double pi;                     //π
public:
    //常成员数据必须在构造函数初始化列表中初始化
    Circle(double x,double y,double radius,double pi):pi(pi){
        this->x=x; this->y=y; this->radius=radius;
    }
    double getRadius(){
        return radius;
    }
    double area() const{                 //常成员函数可以访问常成员数据和非常成员数据
        return pi*radius*radius;
    }
};
int main(){
    Circle c1(1.2,3.4,3,3.14);
    cout<<c1.area();
    return 0;
}
```

从例 3.11 中可以看出,常成员函数 area 可以访问常成员数据 pi 和非常成员数据 radius。但是若把 area 函数改为

```
double area() const{
    return pi*getRadius()*getRadius();
}
```

这是错误的,因为 area 函数调用了非常成员函数 getRadius(),这是不允许的。另外,常成员函数可以用来定义重载函数,也就是说,在已经定义常成员函数 area 的基础上,还可以重载定义一个非常成员函数 area 如下。

```
double area(){                           //成员函数可以访问常成员数据和非常成员数据
    return pi*radius*radius;
}
```

这两个 area 函数是合法的重载。对于利用 const 重载的成员函数,非常对象默认调用非 const 成员函数,若没有非 const 成员函数则调用 const 成员函数;常对象(3.9.3 节介绍)需要调用 const 成员函数,若不调用 const 成员函数则编译出错。

3.9.3 const 对象

在定义对象时也可以指定对象为常对象，常对象一经定义不能修改，例如：

```
const Clock c1(12,34,46);              //c1 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 3.11 中已定义的 Circle 类，定义常对象：

```
const Circle c1(1.2,3.4,3,3.14);       //定义常对象 c1
c1.getRadius();                        //企图调用常对象 c1 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。

1. 对象与指针间的关系

对象分为 const 和非 const 的，指针也分为 const 和非 const 的，那么对象与指针的关系分为以下几种情形。

对于圆类定义(Circle)：

```
class Circle{
private:
    double x,y;                        //圆心坐标
    double radius;                     //半径
    const double pi;                   //π
public:
    //常成员数据必须在构造函数初始化列表中初始化
    Circle(double x,double y,double radius,double pi):pi(pi){
        this->x=x; this->y=y; this->radius=radius;
    }
    double getRadius(){
        return radius;
    }
    void setRadius(double radius){
        this->radius=radius;
    }
    double area() const{               //常成员函数可以访问常成员数据和非常成员数据
        return pi*radius*radius;
    }
};
```

1）指向对象的指针

对象与指针都不定义为 const 的，例如：

```
Circle c1(1.2, 2.3, 4, 3.14);
Circle * p = &c1;
```

通过对象名 c1 可以访问任何公有成员，如 c1. setRadius(10)、c1. area()。

通过指针 p 可以访问任何公有成员，如 p-> setRadius(10)、p-> area()。

p 的值是可以改变的。

2）指向常对象的指针

指向常对象的指针指向的对象可以是 const 的，也可以是非 const 的，但是通过指向常对象的指针只能访问对象的 const 成员，例如：

```
Circle c1(1.2, 2.3, 4, 3.14);
const Circle * p = &c1;
```

在指针定义中，const 修饰 Circle，所以通过 p 访问对象，对象数据不允许更改（即只能够访问 const 成员）；const 不修饰 p，所以 p 可以更改，也就是说，p 还可以用其他的常对象地址赋值。

通过对象名 c1 可以访问任何公有成员，如 c1. area()、c1. setRadius(10)。

通过指针 p 可以访问 const 公有成员，如 p-> area()是可以的，p-> setRadius(10)是不可以的。

p 的值是可以改变的。

3）指向对象的常指针

对象定义非 const 的，指针定义是 const 的，例如：

```
Circle c1(1.2, 2.3, 4, 3.14);
Circle * const p = &c1;
```

c1 为非 const 对象，p 为 const 的；在指针定义中，const 不修饰 Circle，所以对象数据允许更改；const 修饰 p，所以 p 不可以更改，也就是说，p 不可以再用其他的对象地址赋值。

通过对象名 c1 可以访问任何公有成员，如 c1. setRadius(10)、c1. area()。

通过指针 p 可以访问任何公有成员，如 p-> setRadius(10)、p-> area()。

p 的值是不可以改变的。

4）指向常对象的常指针

对象定义 const 的，指针指向的对象必须是 const 的，并且指针也是 const 的，例如：

```
const Circle c1(1.2, 2.3, 4, 3.14);
const Circle * const p = &c1;
```

c1 为常对象，p 指向对象必须是 const 的；在指针定义中，const 修饰 Circle，所以对象数据不允许更改；const 修饰 p，所以 p 不可以更改，也就是说，p 不可以用其他的常对象地址赋值。

通过对象名 c1 可以访问 const 公有成员，如 c1. area()是可以的，c1. setRadius(10)是不可以的。

通过指针 p 可以访问 const 公有成员，如 p-> area()是可以的，p-> setRadius(10)是不可以的。

p 的值是不可以改变的。

2. 函数参数为对象指针

(1) 如果函数的形参是指向 const 对象的指针,在执行函数过程中不能改变指针变量所指向的对象的值,允许实参是指向 const 对象的指针,或指向非 const 对象的指针。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 的或非 const 的)。

函数 f 定义如下。

```
void f(const Circle * p){
  //cout << p -> setRadius(10);      非法
  cout << p -> area();
}
```

函数调用:

```
Circle c1(1.2, 2.3, 4, 3.14);
f(&c1);
```

因为 p 是指向 const 对象的指针,所以通过形参指针 p 访问实参对象的非 const 函数是不允许的,不管实参对象是否是 const 的。

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

3. 函数参数为对象引用

一个对象的引用就是对象的别名。实质上,对象名和引用名都指向同一段内存单元。如果形参为对象的引用名,实参为对象名,则在调用函数进行实参形参结合时,并不是为形参另外开辟一个存储空间,而是把实参对象的地址传给形参(引用名),这样引用名也指向实参对象,这样,被调用函数中形参引用变量即为实参对象。

函数 f 定义如下。

```
void f(Circle & r){
  cout << r.setRadius(10);
  cout << r.area();
}
```

函数调用:

```
Circle c1(1.2, 2.3, 4, 3.14);
f(c1);
```

在函数 f 中对 r 的访问等价于对实参 c1 的访问,若 f 中改变了 r 的数据,则实参对象 c1 也会随之改变。然而,很多时候,在函数中只允许读取形参(即实参对象)数据,但不能改变它,为此,需要在引用定义时加上 const 约束,例如:

```
void f(const Circle & r){
  //cout << r.setRadius(10);      非法
  cout << r.area();
}
```

对于 const 引用,则不能够通过引用对象访问非 const 成员,也就是不能修改其数据成

员。另外,若实参对象为 const 对象,则形参引用必须为 const 的。

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,可以提高程序运行效率。

3.10 友元函数和友元类

类的私有成员只能在类的定义范围内使用,也就是说,类的私有成员只能通过本类的成员函数来访问,但有时候又需要在类的外部访问类的私有成员,甚至需要同时访问多个类的私有成员。为了解决这个问题,C++引入了友元来实现在类的外部访问类的私有成员的问题。这样,既可不放弃私有数据的安全性,又可在类的外部访问类的私有成员。

友元提供了不同类或对象的成员函数之间、类的成员函数与一般函数之间进行数据共享的一种手段。通过友元这种方式,一个普通函数或类的成员函数可以访问封装在类内部的数据,外部通过友元可以看见类内部的一些属性。但这样做,会使数据的封装性受到削弱,使程序的可维护性变差,使用时一定要慎重。

在一个类中,声明为友元的外界对象可以是不属于任何类的一般函数,也可以是另一个类的成员函数,还可以是一个完整的类。

3.10.1 友元函数

如果友元是普通函数,则称为友元函数。友元函数是在类声明中用关键字 friend 说明的非成员函数。它不是当前类的成员函数,而是独立于当前类的外部函数,可以访问该类的所有对象的私有或公有成员,位置可以放在私有部分,也可放在公有部分(因为它不是成员,不受访问控制权限的约束)。

普通函数声明为友元函数的一般形式为

```
friend <数据类型><友元函数名>(参数表);
```

例 3.12 普通函数作为类的友元函数。

```
//ch3_12.cpp
//平面上点到点的距离为 sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2))
#include <iostream>
#include <cmath>
using namespace std;
//Point 类定义和实现
class Point
{
private:
   double x, y;                          //x,y 坐标
public:
   Point(double x = 0.0, double y = 0.0);
   void disp();                          //输出私有变量的成员函数
   friend double distance1(Point p1,Point p2);        //声明为 Point 的友元

};
Point::Point(double x, double y){
```

```
    this->x = x; this->y = y;
}
void Point::disp(){ cout<<"点("<<x<<","<<y<<")"; }
//距离函数
double distance1(Point p1,Point p2){
   return sqrt((p1.x-p2.x)*(p1.x-p2.x)+(p1.y-p2.y)*(p1.y-p2.y));
}
//main 函数
int main(){
    Point p1(1.0,2.0);
    Point p2(3.0,4.0);
    p1.disp();
    p2.disp();
    cout<<"距离为:"<<distance1(p1,p2)<<endl;
    return 0;
}
```

由于友元函数不是成员函数,因此,在类外定义友元函数时,不必像成员函数那样,在函数名前加“类名::”;也不能通过 this 指针引用对象的成员,必须通过入口参数传递进来的对象名或对象指针来引用该对象的成员。

当一个函数需要访问多个类时,应该把这个函数同时声明为这些类的友元函数,这样,这个函数才能访问这些类的数据,如例 3.13。

例 3.13 普通函数作为多个类的友元函数。

```
//ch3_13.cpp
//平面上点到直线的距离
//设点的坐标为(x,y),直线为 ax+by+c=0
//点到直线的距离为 d=abs(a*x+b*y+c)/sqrt(a*a+b*b)
#include<iostream>
#include<cmath>
using namespace std;
class Line;                                    //Line 类声明
//Point 类定义和实现
class Point
{
 private:
   double x, y;                                //x,y 坐标
 public:
   Point(double x=0.0, double y=0.0);
   void disp();                                //输出私有变量的成员函数
   friend double distance1(Point p,Line l);    //声明为 Point 的友元
};
Point::Point (double x, double y){
    this->x = x; this->y = y;
}
void Point::disp(){ cout<<"点("<<x<<","<<y<<")"; }
//直线类定义和实现
class Line
{
 private:
   double a, b, c;                             //直线为 ax+by+c=0 的系数
 public:
   Line(double a=0.0, double b=0.0, double c=0.0);
```

```
    void disp();                                    //输出私有变量的成员函数
    friend double distance1(Point p,Line l);        //声明为 Line 的友元
};
Line::Line(double a, double b, double c){
    this->a = a; this->b = b; this->c = c;
}
void Line::disp(){ cout<<"线("<<a<<"x+"<<b<<"y+"<<c<<"=0)";}
//距离函数
double distance1(Point p,Line l){
    return fabs(l.a*p.x+l.b*p.y+l.c)/sqrt(l.a*l.a+l.b*l.b);
}
//main 函数
int main(){
    Point p1(1.0,2.0);
    Line l1(3.0,4.0,5.0);
    p1.disp();
    l1.disp();
    cout<<"距离为:"<<distance1(p1,l1)<<endl;
    return 0;
}
```

在例 3.13 中,distance1 函数即为 Point 的友元,又为 Line 的友元,这样两个类的互相不可见的私有数据在 distance1 函数中都可以访问。

3.10.2 友元成员函数

如果一个类的成员函数是另一个类的友元函数,则称这个成员函数为友元成员。通过友元成员函数,不仅可以访问自己所在类对象中的私有和公有成员,还可以访问由关键字 friend 声明语句所在的类对象中的私有成员和公有成员,从而可以使两个类相互合作,完成某个任务。点和直线距离可以利用友元成员函数实现,如例 3.14。

例 3.14 成员函数作为类的友元函数。

```
//ch3_14.cpp
//平面上点到直线的距离
//设点的坐标为(x,y),直线为 ax+by+c=0
//点到直线的距离为 d=abs(a*x+b*y+c)/sqrt(a*a+b*b)
#include<iostream>
#include<cmath>
using namespace std;
class Line;
//Point 类的定义和实现
class Point
{
 private:
    double x, y;                                    //x,y 坐标
 public:
    Point(double x=0.0, double y=0.0);
    void disp();                                    //输出私有变量的成员函数
    double distance1(Line l);                       //声明为 Point 的成员
};
Point::Point(double x, double y){
    this->x = x; this->y = y;
}
```

```
void Point::disp(){ cout <<"点("<< x <<","<< y <<")"; }
//直线类的定义和实现
class Line
{
 private:
   double a, b, c;                                  //直线为 ax + by + c = 0 的系数
 public:
   Line(double a = 0.0, double b = 0.0, double c = 0.0);
   void disp();                                     //输出私有变量的成员函数
   friend double Point::distance1(Line l);          //声明为 Line 的友元
};
Line::Line(double a, double b, double c){
    this->a = a; this->b = b; this->c = c;
}
void Line::disp(){ cout <<"线("<< a <<"x + "<< b <<"y + "<< c <<" = 0)";}
//距离函数
double Point::distance1(Line l){
   return fabs(l.a * x + l.b * y + l.c)/sqrt(l.a * l.a + l.b * l.b);
}
//main 函数
int main(){
    Point p1(1.0,2.0);
    Line l1(3.0,4.0,5.0);
    p1.disp();
    l1.disp();
    cout <<"距离为:"<< p1.distance1(l1)<< endl;
    return 0;
}
```

当一个类的成员函数作为另一个类的友元函数时,必须先定义成员函数所在的类,如类 Point 的成员函数 distance1()为类 Line 的友元函数,就必须先定义类 Point。并且在声明友元函数时,要加上成员函数所在类的类名和运算符“::”,如例 3.14 中在 Line 类中声明

```
friend double Point::distance1(Line l);
```

与例 3.13 的主要区别为 distance1()是成员函数,实现时要有类名约束,调用时也要通过对象名调用。

3.10.3 友元类

当一个类作为另一个类的友元时,称这个类为友元类。当一个类成为另一个类的友元类时,这个类的所有成员函数都成为另一个类的友元函数,因此,友元类中的所有成员函数都可以通过对象名直接访问另一个类中的私有成员,从而实现了不同类之间的数据共享。

友元类声明的形式如下。

```
friend class <友元类名>;
```

友元类的声明可以放在类声明中的任何位置。

计算点和点的距离、点和直线的距离可以利用如例 3.15 所示的友元类方法实现。

例 3.15 友元类。

```
//ch3_15.cpp
//平面上点到点的距离为 sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2))
```

```
//平面上点到直线的距离
//设点的坐标为(x,y),直线为 ax + by + c = 0
//点到直线的距离为 d = abs(a * x + b * y + c)/sqrt(a * a + b * b)
#include <iostream>
#include <cmath>
using namespace std;
//Point 类的定义和实现
class Point
{
 private:
   double x, y;                                  //x,y 坐标
 public:
   Point(double x = 0.0, double y = 0.0);
   void disp();                                  //输出私有变量的成员函数
   friend class ComputeTools;                    //声明为 Point 的友元类
};
Point::Point(double x, double y){
    this->x = x; this->y = y;
}
void Point::disp(){ cout <<"点("<< x <<","<< y <<")"; }
//直线类的定义和实现
class Line
{
 private:
   double a, b, c;                               //直线为 ax + by + c = 0 的系数
 public:
   Line(double a = 0.0, double b = 0.0, double c = 0.0);
   void disp();                                  //输出私有变量的成员函数
   friend class ComputeTools;                    //声明为 Line 的友元类
};
Line::Line(double a, double b, double c){
    this->a = a; this->b = b; this->c = c;
}
void Line::disp(){ cout <<"线("<< a <<"x + "<< b <<"y + "<< c <<" = 0)";}
 //ComputeTools 类的定义和实现
class ComputeTools{
public:
    //重载点与点距离函数
    static double distance(Point p1,Point p2);
    //重载点与直线距离函数
    static double distance(Point p,Line l);
};
double ComputeTools::distance(Point p1,Point p2){
  return sqrt((p1.x - p2.x) * (p1.x - p2.x) + (p1.y - p2.y) * (p1.y - p2.y));
}
double ComputeTools::distance(Point p,Line l){
   return fabs(l.a * p.x + l.b * p.y + l.c)/sqrt(l.a * l.a + l.b * l.b);
}
//main 函数
int main(){
    Point p1(1.0,2.0),p2(3.0,4.0);
    Line l1(3.0,4.0,5.0);
    p1.disp();
    p2.disp();
```

```
    cout <<"距离为:"<< ComputeTools::distance(p1,p2)<< endl;
    p1.disp();
    l1.disp();
    cout <<"距离为:"<< ComputeTools::distance(p1,l1)<< endl;
    return 0;
}
```

程序运行结果如下。

```
点(1,2)点(3,4)距离为:2.82843
点(1,2)线(3x + 4y + 5 = 0)距离为:3.2
```

例 3.15 中定义了一个 ComputeTools 类作为 Point 点类、Line 线类(称为数据对象类)的友元类,在 ComputeTools 类定义了访问 Point 点类、Line 线类的一些静态函数,来处理 Point 点类、Line 线类对象的相关联的数据。这样设计,当某种数据对象类改变、增减一种数据对象类时,不会影响其他数据对象类,只需要改变 ComputeTools 类即可。例如,增加一种曲线类,只需要在 ComputeTools 类中增加一些点到曲线的距离函数等,这是一种常用的设计方式。

友元关系是不能传递的。类 B 是类 A 的友元,类 C 是类 B 的友元,并不表示类 C 是类 A 的友元。

友元关系是单向的。类 A 是类 B 的友元,类 A 的成员函数可以访问类 B 的私有成员和保护成员;反之,类 B 不是类 A 的友元,类 B 的成员函数却不可以访问类 A 的私有成员和保护成员。

3.11 类组合关系

类中的成员,除了可以为基本数据类型外,还可以是一个已定义类类型,称此做法为类的组合或类的聚合,第 5 章会有关于类的关系的详细讲解。

类可以将其他类对象作为自己的成员,形成类的嵌套。当一个类的对象作为另一个类的成员时,称此对象为对象成员。声明对象成员时,其数据类型为该对象所对应类的类名。例如:

```
class Date                                    //先定义一个日期类
{
 private:
   int year,month,day;                        //声明私有成员变量
 public:
   Date(int yr,int mn,int dy)                 //定义构造函数,它的名字与类名相同
   { year = yr; month = mn;day = dy;}
};
class Student                                 //定义 Student 类
{
private:
  int num;                                    //声明私有成员变量
  char name[20];
  Date birthday;                              //声明私有对象成员
public:
    Student(int n,const char * nam,Date birth);        //声明构造函数
```

```
};
Student:: Student(int n,const char * nam, Date birth):birthday(birth)
{ num = n; strcpy(name,nam); }                    //定义组合类的构造函数
```


对包含对象成员的类对象初始化时，既要初始化类的对象，还要初始化对象成员，显然一般类的构造函数不能承担这项初始化工作，需要定义组合类的构造函数。定义组合类的构造函数的语法格式为

<类名>(参数总表):对象成员 1(形参表),对象成员 2(形参表),…{ 函数体 }

例如：

```
Student:: Student(int n,const char * nam, Date birth):birthday(birth)
{ num = n; strcpy(name,nam); }                    //定义组合类的构造函数
```

构造函数的冒号后面的部分称为成员初始化列表，用于完成对组合类中对象成员的初始化。birthday(birth)调用对象 birthday 所属类 Date 的拷贝构造函数，创建对象 birthday，参数 birth 取自参数总表，即从 Student 的构造函数传递给 Date 的构造函数。

还可以重载一个构造函数：

```
Student:: Student(int n,const char * nam, int y,int m,int d):birthday(y,m,d)
{  num = n; strcpy(name,nam);  }
```

比较两个构造函数见例 3.16。

例 3.16 使用对象成员。

```
//ch3_16.cpp
#include <iostream>
#include <cstring>
using namespace std;
class Date                                        //先定义一个日期类
{
private:
   int year,month,day;                            //声明私有成员变量
public:
   Date(int yr,int mn,int dy)                     //定义构造函数,它的名字与类名相同
   { year = yr; month = mn;day = dy;
   cout <<"constructor Date"<< endl;}
   Date(const Date& d){cout <<"copy constructor Date"<< endl;}
};
class Student                                     //定义 Student 类
{
private:
  int num;                                        //声明私有成员变量
  char name[20];
  Date birthday;                                  //声明私有对象成员
public:
    Student(int n,const char * nam,Date birth);   //声明构造函数
    Student(int n,const char * nam, int y,int m,int d);
};
Student:: Student(int n,const char * nam, Date birth):birthday(birth)
{ num = n; strcpy(name,nam); }                    //定义组合类的构造函数(拷贝构造成员对象)
Student:: Student(int n,const char * nam, int y,int m,int d):birthday(y,m,d)
{ num = n; strcpy(name,nam); }                    //定义组合类的构造函数(构造成员对象)
int main(){
```

```
    Date d1(1,2,3);
    Student s1(1,"abc",d1);
    Student s2(2,"abcd",4,5,6);
    return 0;
}
```

程序运行结果如下。

```
constructor Date
copy constructor Date
copy constructor Date
constructor Date
```

运行结果中第一行的构造函数用来构造对象 d1；第二行拷贝构造函数是对象 s1 的实参 d1 拷贝构造生成形参对象 birth；第三行拷贝构造函数是由对象 birth 拷贝构造生成对象 s1 的成员对象 birthday；第四行构造函数是构造生成 s2 的成员对象 birthday。可以看出，以对象为参数的构造函数由拷贝构造函数生成成员对象。

如果类的成员是一个指向其他类类型的指针，则不涉及成员对象的生成问题，因为此时没有成员对象，只有一个指针。

```
class Student                                          //定义 Student 类
{
private:
    int num;                                           //声明私有成员变量
    char name[20];
    Date * birthday;                                   //声明指向 Date 类型的指针
public:
    Student(int n,const char * nam,Date * birth);      //参数传递进来一个地址即可
};
Student:: Student(int n,const char * nam, Date * birth){
    num = n; strcpy(name,nam);
    birthday = birth;                                  //日期对象已经建立,传递地址进来
 }
int main(){
    Date d1(1,2,3);
    Student s1(1,"abc",&d1);                           //传递地址
    return 0;
}
```

在 main 函数中建立日期对象 d1(构造)，并将地址传递给 s1 对象的成员 birthday(既不构造，也不拷贝构造)，这种成员构成方式称为聚合，Date 和 Student 之间的关系更为松散，第 5 章中会有详细介绍。

3.12 案例分析：算盘

珠算(珠算文化)是以算盘为工具进行数字计算的一种方法，珠算始于汉代，至宋走向成熟，元明达于兴盛，清代以来在中国全国范围内普遍流传。珠算是以算盘为工具，以算理、算法为基础，运用口诀通过手指拨动算珠进行加、减、乘、除和开方等数学运算的计算技术。珠算文化涵盖了与珠算相关的数学科学、数学教育、应用技术及智能开发等内容，在文学、历史、音乐、美术等相关文化领域也有一定的作用。

算盘的新形状为长方形，周为木框，内贯直柱，俗称“档”。一般从九档至十五档，档中横以梁，梁上两珠，每珠作数五，梁下五珠，每珠作数一，运算时定位后拨珠计算，可以做加减乘除等算法，如图 3.12 所示。

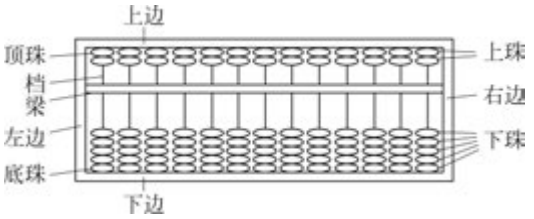

图 3.12　算盘

下面以加法为例，讨论算盘类的设计，即设计类的数据成员和函数成员。

数据成员设计：假定算盘为 13 档(0～12 档)，每档是一个 0～9 的整型数字(由上珠和下珠组成，此处不再设计成两个整数)。

函数成员设计：函数成员包含加、减、乘、除和开方等。此处以加为例，加法需要提供一个整型参数，与算盘上已有数据进行加法运算，运算结果改变了算盘上原有的数据，并在加的过程中输出算盘的拨珠口诀。

算盘类的定义如下。

```
class Abacus{
    int data[13];                          //算盘的 13 档数据,初值均为 0,下标 0 为个位档
    public:
        Abacus();                          //初始化算盘的 13 档均为 0
        void add(int num);
        //… 其他函数略
        void show();                       //输出算盘当前状态
    private:
        void carry(int location);          //进位
};
```

构造函数进行算盘的归 0 操作：

```
Abacus::Abacus(){
    for(int i = 0;i < 13;i++)
        data[i] = 0;
}
```

按照加法口诀，加分不进位的加(加的结果小于 10)和进位的加(加的结果大于或等于 10)两种情形：不进位的加在加数为 1～4 时分为直加和满五加，不进位的加在加数为 5～9 时直加；进位的加在加数为 6～9 时分为进十加和破五进十加，进位的加在加数为 1～5 时进十加。在进位的加情形，需要前进一挡继续加 1，并且这个情形可能持续下去，因此设计一个递归函数 carrt。完整代码如下。

```
#include<iostream>
using namespace std;
class Abacus{
      int data[13];                        //算盘的 13 档数据,初值均为 0,下标 0 为个位档
      public:
        Abacus();                          //初始化算盘的 13 档均为 0
```

```
        void add(int num);
        //... 其他函数略
        void show();                          //输出算盘当前状态
    private:
        void carry(int location);             //进位
};
Abacus::Abacus(){
    for(int i = 0;i < 13;i++)
        data[i] = 0;
}
void Abacus::add(int num){
    //分解 num 的每一位数,从个位开始,然后把该数加到算盘对应的档上
    int currentLocation = 0;                  //需要加到算盘当前档位
    int lowOrderNumber;                       //低位取数
    while(num){
        lowOrderNumber = num % 10;
        cout << currentLocation <<"档:";
        if(lowOrderNumber){                   //非 0 拨珠
            if(data[currentLocation] + lowOrderNumber < 10) {          //不进位的加
            if(lowOrderNumber > 0 && lowOrderNumber < 5)
                    if(data[currentLocation] + lowOrderNumber > = 5)    //满五加
                        cout << lowOrderNumber <<"下 5 去"<< 5 - lowOrderNumber << endl;
                    else                      //直加 1～4
                        cout << lowOrderNumber <<"上"<< lowOrderNumber << endl;
                    else                      //直加 5～9
                        cout << lowOrderNumber <<"上"<< lowOrderNumber << endl;
                data[currentLocation] = data[currentLocation] + lowOrderNumber;
            }
            else{                             //进位的加
            if(lowOrderNumber > = 6 && lowOrderNumber < = 9)
                if(data[currentLocation]> = 5)       //破五进十加
                        cout << lowOrderNumber <<"上"<< lowOrderNumber - 1 <<"去 5 进 1"<< endl;
                else                          //进十加 6～9
                        cout << lowOrderNumber <<"去"<< 10 - lowOrderNumber <<"进 1"<< endl;
                else                          //进十加 1～5
                        cout << lowOrderNumber <<"去"<< 10 - lowOrderNumber <<"进 1"<< endl;
                data[currentLocation] = data[currentLocation] + lowOrderNumber - 10;
                carry(currentLocation + 1);       //进位
            }
        }
        else
            cout << endl;                     //0 不拨珠
        currentLocation++;
        num = num/10;
    }
}
void Abacus::carry(int location){             //location 进位所在的位
    cout <<" "<< location <<"档:";            //进位缩进显示
    //进位的一定是 1
    if(data[location] + 1 < 10) {             //不进位的加
        if(data[location] + 1 > = 5)          //满五加
            cout << 1 <<"下 5 去"<< 4 << endl;
        else                                  //直加 1
            cout << 1 <<"上"<< 1 << endl;
```

```
            data[location] = data[location] + 1;
            }
        else{                                          //进位的加
        //进十加 1,一定加 9
            cout << 1 <<"去"<< 9 <<"进 1"<< endl;
            data[location] = data[location] + 1 - 10;
            carry(location + 1);                       //进位
        }
    }
    void Abacus::show(){
        int firstNonZero = 0;
        for(int i = 12;i > = 0;i -- )
        if(data[i] || firstNonZero){
                    cout << data[i];
                    firstNonZero = 1;
            }
        cout <<"\n";
    }
    int main(){
        Abacus a1;
        a1.add(19995);
        a1.show();
        a1.add(12805);
        a1.show();
    }
```

运行结果如下。

```
0 档:5 上 5
1 档:9 上 9
2 档:9 上 9
3 档:9 上 9
4 档:1 上 1
19995
0 档:5 去 5 进 1
    1 档:1 去 9 进 1
    2 档:1 去 9 进 1
    3 档:1 去 9 进 1
    4 档:1 上 1
1 档:
2 档:8 上 8
3 档:2 上 2
4 档:1 上 1
32800
```

此程序中有以下几点需要进一步思考。

(1) 超出 13 档情形如何处理?

(2) 加数程序中用了 int 型,不太合适,不能描述比较长的整数,是不是用数字字符数组更合适?限于篇幅,书上代码没有去处理。

(3) 算盘的输出表示,使用模拟算盘输出更合理。

(4) 思考一下,若不考虑 int 范围问题,算盘类的数据成员不采用 int 数组,直接用 int 型是否合适?

3.13 UML 类图简介

自从面向对象技术广泛应用以来，人们已经开发了许多建模方法和工具辅助设计软件系统，其中较流行的当属 UML(Unified Modeling Language)。UML 是一种可视化的面向对象的建模语言，它用规范的符号描述软件模型的静态结构、动态行为以及模块组织和管理。这里无法详细讲解 UML 的细则和全部用法，仅介绍其中最基本的静态结构图——类图，本书使用 UML 类图描述类与对象的结构以及联系。

类图可以展示软件系统的静态结构、类的内部结构以及与其他类的关系，是由类和与之相关的各种关系组成的图形。类图使用图形与符号简明地表示类的标识、成员以及访问控制属性，并用特定的线段和箭头表示多个类之间的关系。UML 中一个类的表示为一个矩形，该矩形垂直地分为三个区：顶部区域显示类的名字，中间区域列举类的成员数据，底部区域列出类的成员函数，如图 3.13 所示。除了顶端类的标识部分，下面的两个部分是可选的，即当类图重点描述类的关系而不关心类的内部细节时，可以用一个标注类名的矩形代表该类。

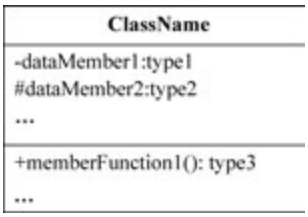

图 3.13 类图示例

类图除了描述类的标识即类名之外，还需显示类中所有成员以及特性，根据图的详细程度不同，需给出其访问控制属性、类型、默认值和约束特性。下面介绍完整表示类成员的方法。

1. 成员数据

类图中间区域中列举该类的所有成员数据，UML 规定成员数据的表示语法为

[访问控制属性] 名称 [多重性] [:类型] [= 默认值] [{约束特性}]

名称：标识成员数据的字符串，标识符可以为变量名、数组名或者对象名。

访问控制属性：包括“+”、“−”和“#”，分别表示 public、private 与 protected。

类型：表示数据类型，可以是基本类型或用户自定义类型。

多重性：指明该属性可能的个数以及唯一性，如 3，1，…，5，0，…，* 等(* 表任意非负整数)。

默认值：赋予该成员数据的初值。

约束特征：对成员性质的说明字符串。

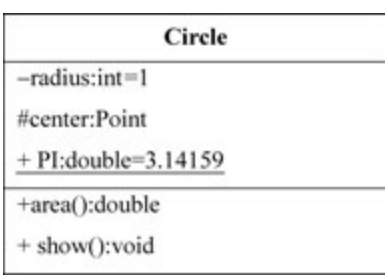

图 3.14 圆形类图

该部分至少要指定每个数据成员的名称，其余[ ]中为可选项。

一个描述图形圆的类图如图 3.14 所示。Circle 类包含三个数据成员：描述半径的 int 类型数据 radius，这个私有成员的默认值为 1；描述圆心位置的数据 center，它是 Point 类型的保护成员，其中，Point 为描述平面坐标的类类型；表示圆周率的 static 成员 PI，为该类的共享数据，类图中用数据有下画线

的表示静态数据成员。此外，Circle 类中还包含两个公有的成员函数 area()和 show()。

2. 成员函数

类图中最底部区域列举类中所有的成员函数，每个函数的说明包括名称、访问控制属性、参数列表、返回值以及约束特性。最简单的形式只给出函数名，其余为可选项。UML 规定成员函数的表示语法为

[访问控制属性] 名称 [(参数列表)] [:返回值] [{约束特性}]

名称：标识成员函数的字符串，即函数名。

参数列表：列举所有参数，多个参数用逗号分隔，每个参数的表示方法为

[方向] 参数名:类型 = 默认值

其中，参数方向可以为输入(in)、输出(out)和输入输出(inout)。

返回类型：表示函数的返回值类型，可以是基本类型或用户自定义类型。

访问控制属性与约束特性的意义同数据成员。

对于一个简化的银行账户类 BankAccount，其类图见图 3.15。其中列举如下公有成员函数：构造函数 BankAccount，参数为 string 类型的 no 和 double 型的 money，用以初始化成员数据 countNo 和余额 balance；进行存款的操作 deposit()，传递参数为 double 类型的 amount 与 Date 类型的 date，该函数无返回值；获取账户余额的操作 getBalance()，返回 double 型的余额；输出账户信息的操作 show()，没有参数与返回值，构造型<< const >>表示它为常成员函数；获取存款利率 rate 的静态成员函数 getRate()，用构造型<< static >>来表示。

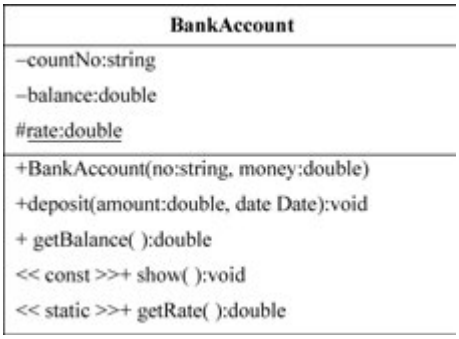

图 3.15　银行账户类图

UML 中使用尖括号“<< >>”表示构造型，即扩展原模型语义的建模元素。类图中使用的一些构造类型包括<< const >><< static >>和<< friend >>。此外，使用构造型<< virtual >>声明虚成员函数，纯虚函数用<< abstract >>表示或描述该成员的字体为斜体。

其他类间关系的描述见后面章节。

习　　题

1. 说明类与对象的关系。

2. 定义一个日期类 Date，它提供由年、月、日组成的私有日期数据和修改日期，以及打印出日期的公有函数，并提供多个重载的构造函数。

3. 编写一个计数器类,定义一个私有整型数据成员,通过两个成员函数分别使其完成加 1 和减 1 操作,构造函数使数据成员初始化为 0,输出函数可以输出数据成员的值。

4. 定义一个分数类如下,要求实现各个成员函数。满足如下要求。

(1) 加减乘除函数的运算过程不要更改两个运算数。

(2) 运算结果生成一个新的分数,并且结果为简分数。

(3) 在主函数中测试两个分数的加减乘除等运算。

(4) 思考约分函数为什么设计成私有的。

```
class Rational
{
public:
    Rational(int nn = 1,int mm = 1);          //构造函数,注意约分
    Rational R_add(const Rational & A);       //加
    Rational R_sub(const Rational & A);       //减
    Rational R_mul(const Rational & A);       //乘
    Rational R_div(const Rational & A);       //除
    void print();                             //以简分数形式显示
private:
    void simple();                            //约分
    int m;                                    //分母
    int n;                                    //分子
};
```

5. 定义一个类包含一个整型数的指针变量,在构造函数中用 new 分配 10 个整型数的内存空间,在析构函数中用 delete 释放内存空间,并编写给内存空间赋值和输出的成员函数。

6. 完成 String 类。

```
class String
{
public:
    String(const char * str = NULL);          //普通构造函数
    String(const String &other);              //拷贝构造函数
    ~ String();                               //析构函数
private:
    char * m_data;                            //用于保存字符串
};
```

完成 String 的三个成员函数,并编写一个主函数,在主函数中对所编写的成员函数进行测试。

7. 完成 Array 类。

```
class Array
{
public:
    Array();                                  //所有数组元素初始化为 0
    int& getData(int i);                      //返回下标为 i 的数组元素的引用
    void print();                             //打印出所有数组元素的值
    void input();                             //对所有数组元素进行输入
private:
    int m_data[10];
};
```

完成 Array 的成员函数，并编写一个主函数，在主函数中对所编写的成员函数进行测试。

8. 拷贝构造函数哪些情况下会被调用？分别举例说明。

9. 什么是 this 指针？它的主要作用是什么？

10. 设计一个类，实现两个复数的四则运算，要求用友元函数实现。

11. 定义圆类(由圆心坐标点类和半径组成)，利用友元函数判断两个圆的位置关系(圆间关系包括相交、相切、相离)。

12. 编写一个类 Node，声明一个数据成员 member 和静态数据成员 count(初始化值为 0)，构造函数初始化数据成员 member，并把静态数据成员 count 加 1，析构函数把静态数据成员 count 减 1，print 函数输出数据成员 member 和静态数据成员 count；在 main 函数中通过定义 Node 类的对象和用 new 申请对象，通过 print 函数显示它们的数据成员和静态成员。

13. 分别介绍 const 修饰数据成员、函数成员、对象的作用。

14. 定义自然对数类(LN)，其数据成员包括 e(底)和 x，e 为 const 的；函数成员为 compute，返回 x 的自然对数值。

15. 定义类体系，描述一个老师可以有多个助教、每个助教可以辅导多名学生的问题。