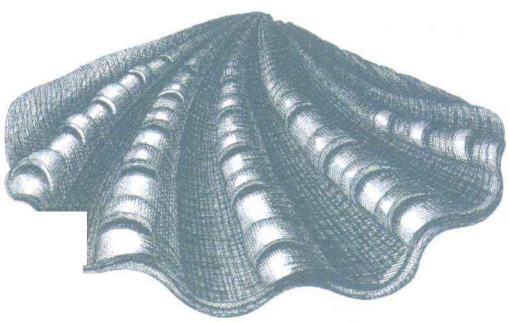
工业和信息化普通高等教育“十二五”规划教材立项项目
21世纪高等学校计算机规划教材
21st Century University Planned Textbooks of Computer Science

面向对象程序设计及C++实验指导（第2版）

The Answer and Practice of Object-Oriented Programming and C++ (2nd Edition)

朱立华 俞琼 主编
郭剑 朱建 副主编

- 配合主教材，透彻解析典型习题
- 紧扣新大纲，全面展现各种题型
- 训练重能力，系统设计编程实验



高校系列

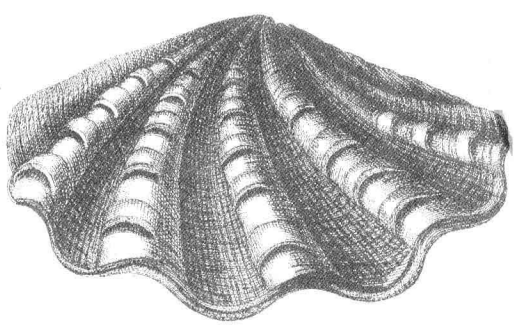
人民邮电出版社
POSTS & TELECOM PRESS

等教育“十二五”规划教材立项项目
规划教材
Computer Science

面向对象程序设计及C++实验指导（第2版）

The Answer and Practice of Object-Oriented Programming and C++ (2nd Edition)

朱立华 俞琼 主编
郭剑 朱建 副主编



高校系列

人民邮电出版社
北京

图书在版编目（CIP）数据

面向对象程序设计及C++实验指导 / 朱立华，俞琼主编. -- 2版. -- 北京 : 人民邮电出版社，2012.2
21世纪高等学校计算机规划教材
ISBN 978-7-115-26905-8

Ⅰ. ①面… Ⅱ. ①朱… ②俞… Ⅲ. ①C语言－程序设计－高等学校－教学参考资料 Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2012)第002499号

内 容 提 要

C++语言同时支持面向过程及面向对象的程序设计，是目前绝大部分高校程序设计课程及计算机编程爱好者首选的编程语言之一。学好C++语言程序设计重点是通过系统的实验训练巩固知识，掌握编程方法。

本书是《面向对象程序设计及C++（第2版）》（朱立华主编，人民邮电出版社2012年出版）的配套教辅用书，其特点是解析清晰透彻，习题面广量大，实验指导详细。全书由4部分组成：第1部分是主教材中例题的思考与练习解析，方便有余力的读者深入学习；第2部分是主教材每章后的习题参考答案与解析，帮助读者正确解题；第3部分给出了与主教材每一章内容配套的补充习题，以弥补主教材因篇幅所限而习题量较少和题型不全面的缺憾，并给出了对应的参考答案；第4部分是实验指导，安排了10个与教材配套的实验，这些实验对初学者全面掌握面向对象的程序设计及C++语言大有帮助。

本书可作为高校学生学习面向对象程序设计及C++语言的辅导教材，也适合单独作为学习C++语言的辅导书。

工业和信息化普通高等教育“十二五”规划教材立项项目

21世纪高等学校计算机规划教材

面向对象程序设计及C++实验指导（第2版）

◆ 主　　编　朱立华　俞　琼
副 主 编　郭　剑　朱　建
责任编辑　武恩玉

◆ 人民邮电出版社出版发行　　北京市崇文区夕照寺街14号
邮编　100061　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京昌平百善印刷厂印刷

◆ 开本：787×1092　1/16
印张：14　　2012年2月第2版
字数：370千字　　2012年2月北京第1次印刷

ISBN 978-7-115-26905-8

定价：29.80元

读者服务热线：(010)67170985　印装质量热线：(010)67129223
反盗版热线：(010)67171154
广告经营许可证：京崇工商广字第0021号

前 言

无论学习何种程序设计语言，仅仅掌握理论知识是不够的，学习程序设计语言最终的目的是能够编写出结构清晰、功能完整的程序，因此，程序设计语言课程对实践环节要求很高。初学者想要真正掌握用C++语言进行面向过程及面向对象的程序设计的知识，必须抓住两个重要环节：一是多做习题，通过各种习题从理论上全面掌握所学的基本知识；二是多上机实践，进而掌握面向对象的编程方法。这两个方面缺一不可，没有坚实的基本知识作铺垫，想要设计出高水平的程序是不可能的；掌握基础知识的最终目的是为了能够编写出更好的程序。为此，我们编写了这本习题解答及实验指导书，作为与主教材配套使用的辅导教材。

相比于第1版，本书作了一些改进，增加了书中第1部分内容，对主教材中例题后提出的思考与练习部分给出了详细的分析与解答，以帮助学有余力的读者更进一步深入理解问题；第2部分是主教材中的习题参考答案及解析，帮助初学者更好地疏理所学知识；第3部分给出了与主教材每一章内容配套的补充习题，以弥补主教材习题数量较少、题型相对有限的缺憾；第4部分是实验指导，增加了Visual Studio 2008集成开发环境使用的介绍，并精心设计了10个实验，对每个实验给出了实验目的、实验要求和实验题目，并对每一个题目提供了必要的实验分析和指导。认真完成这10个实验，对掌握用C++语言进行面向过程及面向对象的程序设计的知识非常有用。本书第2部分和第3部分的编程题只在书中给出了编程提示，所有的源代码均可以从人民邮电出版社教学服务与资源网的网站上免费下载（在www.ptpedu.com.cn的下载区）。

本书主教材第1章、第2章和第4章对应的思考与练习解析，主教材习题解答，补充习题以及实验1、实验4、实验10由朱立华编写；主教材第3章和第7章的对应内容以及实验2、实验3、实验8由俞琼编写；Visual Studio 2008集成开发环境使用的介绍，主教材第5章和第9章的对应内容以及实验5、实验6由郭剑编写；主教材第6章和第8章的对应内容以及实验7、实验9由朱建编写；全书由朱立华统稿。

由于编者水平有限，错误和不当之处在所难免，恳请广大读者批评指正。

编　者

2011年10月

目　录

第 1 部分 教材思考与练习解析

第 2 章　C++语言对 C 语言的改进及扩展

1．例 2.1 的思考与练习。

请查阅相关资料，对程序 example2_01.cpp 中变量 f 的输出进行控制，达到无论输入值是什么，输出 f 变量时，小数点后只保留一位。

【分析与解答】C++程序中如果直接用 cout<<f 这种形式输出变量 f 的值时采用了系统默认的格式，输出的是变量的实际值，不能控制输出精度。如果希望控制只输出小数点后面一位，可以利用 ios 类中有关格式控制的成员函数 **cout.precision(n)；** 或操纵函数 **setprecision(n)**进行格式控制。这里的 n 包含了整数位数及小数点后的位数，不包含正负号及小数点。因此，这个思考题最关键的就是需要设计一个函数，使得无论输入的值是什么，都能得到最后 f 变量的整数部分的位数，有了这个位数，再加上 1，就得到了参数 n 需要的值。

【参考程序】完整的程序如下：

```
/*think2_1.cpp:C++源程序 example2_01.cpp 的思考与练习,精度控制*/
#include <iostream>
#include <iomanip>                          //进行格式控制需要包含此头文件
using namespace std;
int digit(int n);                           //返回整数 n 的位数
int main()
{
     char c ;                               //定义变量 c、a、f 的值
     int a ;
     float f ;
     cin>>c>>a>>f ;                         //输入变量 c、a、f 的值
     a=a+c;
     f=f+2*a;
     cout.precision(digit((int)(f))+1);//方法 1: 用 ios 类中的成员函数 cout.precision(n);
     cout<<"c="<<c<<"  a="<<a<<"  f="<<f<<endl ; //输出变量 c、a、f 的值，f 的小数点后面输出 1 位
     return 0;
}
int digit(int n)
{
```

```
    int d=0;    //统计n的位数
    do
    {
        n=n/10;
        d++;
    }while (n);
    return d;
}
```

另外一种方法是，用操纵函数 **setprecision(n)**控制，将上面主函数中有底纹的两条语句删除，用下面一条语句替代：

```
cout<<"c="<<c<<"  a="<<a<<"  f="<<setprecision(digit((int)(f))+1)<<f<<endl ;
```

说明：此程序中用到了类型的强制转换(int)(f)，将此作为调用 digit 函数的实参。事实上，在C++中类型强制转换更常用的形式是 int(f)，这将会在 2.1.4 小节中介绍。

2．例 2.4 的思考与练习。

在程序 example2_04.cpp 的 using namespace one ; 和 int main( )之前增加一条语句：using namespace two ;，其余语句不变，编译修改后的程序观察结果并分析原因。在此基础上只修改一处，使得输出结果与程序 example2_04.cpp 的完全相同，应当如何修改？

【分析与解答】当增加了一条名字空间声明语句 using namespace two ;之后，就出现了一个问题：空间 one 和 two 均用方法 1 进行了声明，两个空间中的内容在 main()函数中都是可以不加前缀直接使用的。现在的问题是，这两个空间中均有一个变量 inf，对于这个同名的变量，main()函数的语句 cout<<inf<<endl;中的 inf 究竟来自于哪一个名字空间就说不清楚了，既然如此，在这个位置编译必定无法通过。怎么解决呢？one 和 two 两个名字空间都声明了，因此，所有 inf 出现的位置都一定要用方法 2 指明究竟是来自哪一个名字空间，这样肯定不出错。因此需要将 main()函数中的语句 cout<<inf<<endl;修改为 cout<<one::inf<<endl;，同时可以删除多余的语句 using two::x;，再编译链接运行程序，一切正常，结果与程序 example2_04.cpp 的完全一样。

【参考程序】完整的程序如下：

```
//think2_2.cpp: example2_04.cpp 名字空间使用示例思考题
#include <iostream>
using namespace std;        //using声明使用一个完整的名字空间 std，C++中提供的名字
                            //空间 std 涵盖了所有标准 C++的定义和声明
namespace one               //定义一个名字空间 one，里面有 1 个常量 M 和 1 个变量 inf
{
   const int M=200;
   int inf=10;
}                           //后面未加分号
namespace two               //定义一个名字空间 two，里面有 2 个变量：x 和 inf
{
   int x;
   int inf=-100 ;
}                           //后面未加分号
using namespace one ;       //方法 1：using 声明使用一个完整的名字空间 one
using namespace two ;       //增加这一条语句
int  main()
{                           //此处原来语句删除
```

```
    x=-100 ;                    //直接访问，相当于 two::x=-100;
    cout<<one::inf<<endl;       //修改过,用 one::inf 的形式访问空间 one 中的同名变量
    cout<<M<<endl;
    two::inf*=2;                //方法 2:使用名字空间名::局部内容名操作未使用 using 声明的内容
    cout<<two::inf<<endl;       //同样是 two 中的内容，但是访问方式不一样
    cout<<x<<endl ;             //已用 using 声明了 two 中内容 x，可以直接访问
    return  0;
}
```

3．例 2.5 的思考与练习。

（1）将程序 example2_05.cpp 中的 sum 变量的定义移到 main()函数体的一开始，再重新编译链接运行程序，观察结果是否有变化，并作解释。

（2）在程序 example2_05.cpp 的源代码中删除标有底纹的一对大括号，再重新编译，观察编译的结果并作解释。

【分析与解答】（1）C++中的局部变量可以随用随定义，局部变量的定义和声明可以在程序块的任何位置出现，因此，如果将 sum 的定义移至函数的开头，对程序的运行结果不会有任何影响，只是在函数的前半段未进行求和时，该变量一直在其作用域内未发挥作用。

（2）在程序 example2_05.cpp 的源代码中删除标有底纹的一对大括号，则 main()函数就只有一个程序块了，第 2 个局部变量 i 和第 1 个局部变量 i 就处于同一个程序块内，而在同一个程序块内是不允许出现同名标识符的，因此在编译时会出现：**error C2374: 'i' : redefinition; multiple initialization**，表明第 2 个 i 重复定义，以及多个初始化。其实，本质上的错误就是在同一程序块内定义了一个同名变量 i。

读者也可以尝试一下在同一个程序块内同一个标识符既被定义为常量又被定义为变量的情况，只要是同样的标识符，在同一个程序块内就永远只能被定义一次。

4．例 2.6 的思考与练习。

将程序 example2_06.cpp 中的语句 int sum=200；移到 main()函数体的最开头，重新编译程序，是否正确？如果程序正确，请观察运行结果并作分析。

【参考程序】按要求移动之后的完整程序如下：

```
//think2_4.cpp: example2_06.cpp 的思考题，移动语句 int sum=200; 到函数开头
#include <iostream>
#include <iomanip>
using namespace std;                    //使用 C++的标准名字空间
int sum=5000;                           //定义全局变量 sum
int main()
{
     int sum=200;                       //定义并初始化一个局部变量 sum，此句按要求移动到此
    int arr[3]={10,20,30};
    {                                   //一个小程序块的开始
         int i,sum=0;                   //定义另一个局部变量 sum
         for (i=0;i<3;i++)
         sum+=arr[i];                   //求和，结果存于第 2 个局部变量 sum 中
         cout<<"sum="<<sum<<endl;       //输出第 2 个局部变量 sum 的值
          ::sum+=sum;                   //通过域解析符在同名局部变量的作用域内对全局 sum 访问
         cout<<"全局 sum="<<::sum<<endl;  //输出全局 sum 变量的值
```

```
    }                                   //小程序块的结束,第 2 个局部变量 sum 的作用域结束
    sum*=2;                             //这是第 1 个局部变量 sum 的值扩大两倍
    cout<<"sum="<<sum<<endl;            //输出第 1 个局部变量 sum 的值
    cout<<"sum="<<sum<<endl;            //还是输出第 1 个局部变量 sum 的值
    ::sum+=sum;                         //通过域解析符使全局 sum 在同名局部 sum 的作用域可见
    cout<<"全局 sum="<<::sum<<endl;     //输出全局 sum 变量的值
    return 0;
}
```

【分析与解答】该程序仍然是正确的程序，与原来的程序 example2_06.cpp 相比，原来定义在后面的第 2 个局部变量 int sum=200; 调整到最前面之后就变成了第 1 个被定义的局部变量，在小程序块内又定义了第 2 个局部变量 sum，因此第 1 条输出语句产生的输出是求和的结果，也就是第 2 个局部变量 sum 的值 60；小程序块内的全局变量得到了修改，第 2 条输出语句输出修改后的全局变量 sum 的值 5060；小程序块结束也就是第 2 个局部变量 sum 作用域结束之处，此后直接书写的 sum 就是第 1 个局部变量，因此后面的 sum*=2 是对第 1 个局部变量 sum 作扩大两倍的修改，接下去的两条输出语句输出的内容均为第 1 个局部变量 sum 的当前值 400；最后一条输出语句显然是输出修改后的全局变量 sum 的值，语句**::sum+=sum;** 赋值号左边是全局变量，右边是第 1 个局部变量 sum，因此全局变量是用 5060 加上 400 得到 5460，最后输出。

程序的运行结果：

```
sum=60
全局 sum=5060
sum=400
sum=400
全局 sum=5460
```

5．例 2.7 的思考与练习。

将程序 example2_07.cpp 中的函数原型声明语句 **void Fun(int i,int j=5,int k=10)**；分别改成以下 4 条语句：

（1）void Fun(int i,int j,int k=10);

（2）void Fun(int i=1,int j=5,int k=10);

（3）void Fun(int i,int j=5,int k);

（4）void Fun(int i=1,int j,int k=10);

每次修改后重新编译，观察程序是否正确。如果程序正确，则运行程序得到相应的运行结果。无论程序正确与否，均需要作出分析。

【原来的程序】

```
// example2_07.cpp：形式参数带默认参数值的函数定义及调用示例
#include <iostream>
using namespace std;
void Fun(int i,int j=5,int k=10) ; //原型声明中形参 j 和 k 分别指定了默认参数值 5 和 10
int main()
{
    Fun(20) ;                       //实际参数个数少于形式参数，20 与 i 对应,至少有一个实参
                                    //形式参数 j 和 k 分别使用默认参数值 5 和 10
    Fun(20,30) ;                    //形式参数 k 使用默认参数值 10
    Fun(20,30,40);                  //实际参数个数等于形参个数，都不使用默认参数值
```

```
    return 0;
}
void Fun(int i,int j,int k)              //原型中已指定了默认参数值，在定义的首部不能再指定
{
    cout<<i<<"  "<<j<<"  "<<k<<endl;
}
```

【分析与解答】如果将函数原型改为（1）所规定的语句 **void Fun(int i,int j,int k=10);**，此时原型声明是正确的，只有最右边一个默认参数值，因此调用语句所需要的实际参数个数至少为 2 个，最多为 3 个，因此在编译的时候会将错误定位于语句 Fun(20);处，同时显示的报错信息是：**error C2660: 'Fun' : function does not take 1 parameters**，指函数不只带有一个实际参数。

如果将函数原型改为（2）所规定的语句 **void Fun(int i=1,int j=5,int k=10);**，此时原型声明是正确的，3 个形参均有默认参数值，因此调用语句所需要的实际参数个数至少为 0 个，最多为 3 个，函数调用语句是正确的，因此编译无错。运行程序，根据实际参数值优先的原则，在实际参数不足的情况下才使用默认参数值，因此该程序的运行结果与原来的程序运行结果完全一样。

如果将函数原型改为（3）所规定的语句 **void Fun(int i,int j=5,int k);**，此时原型声明是错误的，因为只有第 2 个形参有默认参数值，违反了默认参数值的设定按从右至左的顺序依次的原则，编译的时候显示有 3 个错误，分别是：

① error C2548: 'Fun' : missing default parameter for parameter 3，该错误定位在原型声明语句处，意思是，缺少第 3 个形参的默认参数值；

② error C2660: 'Fun' : function does not take 1 parameters，该错误定位在函数调用语句 Fun(20); 处，意思是，函数调用不可以只有 1 个实际参数；

③ error C2660: 'Fun' : function does not take 2 parameters，该错误定位在函数调用语句 Fun(20, 30); 处，意思是，函数调用不可以只有两个实际参数；

以上虽然显示 3 个错误信息，但本质上是由于第 1 个错误，即默认参数不正确的设置引起的。

如果将函数原型改为（4）所规定的语句 **void Fun(int i=1,int j,int k=10);**，此时原型声明是错误的，因为第 1 和 3 个形式参数具有默认参数值而第 2 个形参没有默认参数值，中间有间隔，违反了默认参数值的设定按从右至左的顺序依次的原则，编译的时候显示有 3 个错误，分别是：

① error C2548: 'Fun' : missing default parameter for parameter 2，该错误定位在原型声明语句处，意思是，缺少第 2 个形参的默认参数值；

② error C2660: 'Fun' : function does not take 1 parameters，该错误定位在函数调用语句 Fun(20); 处，意思是，函数调用不可以只有一个实际参数；

③ error C2660: 'Fun' : function does not take 2 parameters，该错误定位在函数调用语句 Fun(20, 30); 处，意思是，函数调用不可以只有两个实际参数；

以上虽然显示 3 个错误信息，但本质上是由于第 1 个错误，即默认参数不正确的设置引起的。

6．例 2.8 的思考与练习。

如何修改 example2_08_1.cpp 中的宏定义，其他代码都不变，使其运行结果得到 35?

【分析与解答】只要将宏定义修改为：**#define Multiply(x,y) (x) * (y)**，也就是说将宏参数的两边加上一对括号，因为，这时候 **int a=Multiply(3+4,2+3);**宏展开后就变成 **int a= (3+4) * (2+3)**，就能得到 35 的结果了。因此宏定义时，比较保险的做法是将所有参数都加上括号。

【参考程序】修改后的完整程序如下：

```
//think2_6.cpp: example2_08_1.cpp 用宏定义实现两数相乘修改后的程序
```

```
#include <iostream>
using namespace std;
#define  Multiply(x,y)  (x) * (y)    //注意:此处x和y两边未加括号
int  main()
{
     int a=Multiply(3+4,2+3);        //展开后为:int a=(3+4) * (2+3)
     cout<<"a="<<a<<endl;
     return 0;
}
```

7．例2.9的思考与练习。

将程序example2_09.cpp中的函数定义首部**int square(int x)**修改为 **int square(int x=10)**，即增加一个默认参数值，重新编译程序，观察程序正确与否，并作分析。

【分析与解答】当整型参数版本的函数增加一个默认值变成int square(int x=10)时，调用该函数将不需要提供任何实参，而程序中早已定义的另一个函数double square(double x=1.5)调用时也是不需要提供任何实参的，这样，编译时将会报错 error C2668: 'square' : ambiguous call to overloaded function，并且错误位置定位于语句cout<<"square()="<<square()<<endl; 处，因为编译程序无法确定**square()**调用的是**int square(int x=10)**还是**double square(double x=1.5)**，引起歧义。

【参考程序】修改后的完整程序如下：

```
//think2_7.cpp：例2.9重载函数示例思考题答案
#include <iostream>
using namespace std;
int square(int x=10)                              //重载函数的第1版本，int型参数
{
       return x*x;
}
float square(float x )                            //重载函数的第2版本，float型参数
{
       return x*x;
 }
double square(double x=1.5 )                      //重载函数的第3版本，double型参数
{   return x*x;  }
int main()
{  cout<<"square()="<<square()<<endl;             //调用第3版本函数，用默认值，结果为2.25
   cout<<"square(10)="<<square(10)<<endl;         //调用第1版本函数，结果为100
   cout<<"square(2.5f)="<<square(2.5f)<<endl;     //调用第2版本函数，结果为6.25
   cout<<"square(1.1)="<<square(1.1)<<endl ;      //调用第3版本函数，结果为1.21
   return 0;
}
```

8．例2.10的思考与练习。

将程序example2_10.cpp中的语句int &r=x；修改为int &r；，即不对r进行初始化，重新编译程序，观察程序正确与否，并作分析。

【分析与解答】将语句这样修改之后重新编译，会显示一个错误信息：**error C2530: 'r' : references must be initialized**，意思很清楚，引用必须要被初始化。这是因为，引用是另一个已存在变量的别名而不是一个独立的变量，系统不会为它单独分配空间，因此在定义引用时就必须明确指出它是哪一个变量的别名，从而才能确定引用与哪一个变量共享空间。

【参考程序】

```
//think2_8.cpp：主教材 example2_10.cpp 的思考题,引用的声明及使用示例
#include <iostream>
using namespace std;
int x=5,y=10;
int &r;                                          //定义一个引用 r 作为变量 x 的别名
void print()                                     //定义一个专门用于输出的函数
{
     cout<<"x="<<x<<" y="<<y<<" r="<<r<<endl ;   //输出 x、y、r 的值
     cout<<"Address of x "<<&x<<endl;            //输出变量 x 的内存地址
     cout<<"Address of y "<<&y<<endl;            //输出变量 y 的内存地址
     cout<<"Address of r "<<&r<<endl ;           //输出引用 r 的内存地址
}
int main()
{
     print();                                    //第 1 次调用输出函数
     r=y ;                                       //相当于 x=y，将 y 的值赋给 x
                                                 //而不是 r 改变为变量 y 的别名
     y=100 ;                                     //对 y 重新赋值不会影响引用 r 的值
     print() ;                                   //再次调用输出函数
     x=200;                                      //对 x 重新赋值，r 随之改变,不会影响变量 y 的值
     print() ;                                   //再次调用输出函数
     return 0;
}
```

9. 例 2.11 的思考与练习。

将程序 example2_11.cpp 中的函数首部 **void swap(int &x,int &y)**修改为 **void swap(int x,int &y)**，即第 1 个形式参数由引用参数改为值形式参数，重新编译程序，观察程序正确与否，如果正确请运行程序并分析。

【分析与解答】将首部这样修改之后，第 1 参数变为值形式参数，与值形式参数对应的实参可以是常量、变量、表达式，因此程序正确通过。由于 x 是值形式参数，在 swap()函数中对 x 的修改不能引起对应实参变量的修改，因此，程序的运行结果一定反映为第 1 个实际参数在调用前后不变，第 2 个实际参数的值在调用前后改变了。

【参考程序】按要求修改后的源程序：

```
//think2_9.cpp：主教材 example2_11.cpp 的思考，用引用作参数修改对应实际参数变量的值
#include <iostream>
using namespace std;
void swap(int x,int &y)              //x 改为值形参
{
     int t=x;                        // 互换 x 和 y，由于 x 是值参，对应实参
     x=y;                            // 无法改变，而 y 是引用参数，对应实参
     y=t;                            // 得到改变
}
int main()
{
     int a=3,b=5,c=10,d=20;
```

```
    cout<<"a="<<a<<"  b="<<b<<endl ;   //输出交换前的a、b值
    swap(a,b);                         //调用函数，参数传递相当于执行了
                                       //int x=a; int &y=b;
    cout<<"a="<<a<<"  b="<<b<<endl;    //输出交换后的a、b值
    cout<<"c="<<c<<"  d="<<d<<endl;    //输出交换前的c、d值
    swap(c,d) ;                        //调用函数，参数传递相当于执行了
                                       //int x=c; int &y=d;
    cout<<"c="<<c<<"  d="<<d<<endl;    //输出交换后的c、d值
    return 0;
}
```

运行结果：

```
a=3  b=5
a=3  b=3
c=10  d=20
c=10  d=10
```

10．例2.13的思考与练习。

对程序example2_13.cpp，分别作以下两种修改，重新编译，观察正误并分析原因。

（1）将Fun函数最后的一条语句**return y;**修改为**return z;**，

（2）将Fun函数最后的一条语句**return y;**修改为**return x;**，

【分析与解答】（1）如果将**return y;**修改为**return z;**，则Fun函数返回的是一个值形式参数，值形式参数是自动局部变量，其内存空间随着函数的调用结束将被系统收回，也就是说，回到主函数调用点的时候，z的空间已消失，而以引用返回的变量其要求就是在该函数调用结束之后其变量空间仍然存在，因此编译的时候系统会有一条告警信息：**warning C4172: returning address of local variable or temporary**。就是说，返回了局部或临时变量的地址，显然这是不允许的，如果一定要忽略这一告警直接运行，则有可能会产生意想不到的结果，无法保证运行结果正确。

（2）如果将**return y;**修改为**return x;**，重新编译，将会有一个报错：**error C2440: 'return' : cannot convert from 'const int' to 'int &'，Conversion loses qualifiers**，意思是不能将常引用转成普通引用，这种转换无效，即不允许这样做。因为常引用是为了保护对应的实际参数变量不被修改，但是引用返回的函数作为左值必定要引起变量的修改。此时程序无法运行。

11．例2.14的思考与练习。

在程序example2_14.cpp的代码**return 0;**之前加入一条语句：**cout<<*vp<<endl;**，重新编译程序，观察正误并分析原因。

【分析与解答】加入此句后再进行编译，会有一个报错：**error C2100: illegal indirection** ，即非法的间接引用。因此void型的指针变量可以获得任意类型的地址，但是必须要进行显式类型转换才能输出void类型指针所指向的内容。

第3章　类与对象

1．例3.1后的关于类AA的思考与练习。

在以上类AA的定义中，PI称之为常数据成员，请查阅资料解释，为什么const double PI=3.14;这条语句放在类外作为普通常量定义是正确的，而在类中却是错误的？怎样才能使类的常成员变

量 PI 值恒为 3.14?

【分析与解答】类中的数据成员也分为两种：值可变或不可变，不可变的就称之为常数据成员，可变的就是普通的数据成员。无论是哪一种数据成员，均不可以在类定义体内用赋值语句初始化，它们的初始化均需要通过调用类的构造函数（3.2 节介绍），只是，常数据成员只能在构造函数的初始化列表中完成初始化；而普通数据成员可以在构造函数的初始化列表或是函数体中用赋值语句给定初值。这些知识，将分别在 3.2 节及第 4 章详细介绍。

2．例 3.2 后的思考与练习。

在 VC++6.0 编译环境中如何建立一个头文件?

【分析与解答】在 visual C++的编辑界面中选择菜单 File 中选项 New，在弹出的菜单中选择 C/C++ Header File，在右侧的File:————中填入头文件名称，在Location:————中选择正确路径，最后单击右下方的[OK]按钮。这样就进入编辑界面，开始编辑头文件的内容，不要忘记保存。

3．程序 example3_05.cpp 的思考与练习。

（1）如果将 example3_05.cpp 中的#include"Date1.h"改为#include<Date1.h>，重新编译会怎样？为什么?

（2）如果将本例主函数中的语句 today.Display(); 改为如下形式：

cout<<today.year<<"-"<<today.month<<"-"<<today.day<<endl;，重新编译会怎样？为什么?

（3）如果需要显示单独的数据成员，应该怎样编写输出语句?

【分析与解答】（1）重新编译，程序会出现错误提示：fatal error C1083: Cannot open include file: 'Date1.h': No such file or directory。因为在文件包含指令中，如果用一对尖括号，则直接到系统文件夹下查找该文件，而不到当前文件夹下寻找，所以找不到此文件。该文件为用户自己定义的头文件，一般放在用户文件夹中，不属于名空间范围。

（2）重新编译这个程序时，编译系统会指示最后一条语句错误： cannot access private member declared in class 'Date'。原因很简单，因为在类外不能通过对象直接访问类的私有成员 year、month 和 day。

（3）因为是显示单独的数据成员，该例中数据成员均为私有成员，不能直接引用，因此应该通过公有成员函数访问私有成员，语句如下：

```
cout<< today.GetYear( ) << "-" << today.GetMonth( ) << "-" <<today.GetDay( ) << endl;
```

4．程序 example3_06.cpp 的思考与练习。

（1）如果将 example3_06.cpp 中的#include"Date1.h"改为#include"Date2.h"，重新编译会怎样？为什么?

（2）将本例语句 today.SetDate(2006,10,17);和 today.Display();改成指针访问形式。

【分析与解答】（1）重新编译，程序完全正确，结果也是相同的，因为这两个头文件的区别在于，类的成员函数实现代码所在的位置不同，但整个类的功能是一样的，无论成员函数的实现代码如何，都是类中封装和信息隐藏的成分，类的对外接口没有任何变化，因此包含哪一个都一样。

（2）将以上两句分别改为：(&today)->SetDate(2006,10,17);和 (&today)->Display( );即可，因为指针访问形式要求其前一定是指针或是指向对象的地址信息。用取地址符作用于对象之前就可以满足要求。

5．程序 example3_08.cpp 的思考与练习。

（1）将 example3_08.cpp 中的函数 SetDate 的 3 条语句均改写成带 this 指针的等效函数。

（2）将 example3_08.cpp 中的函数 Display( )中的最后一条语句改写成：
cout<<"year="<<year<<" ,month="<<month<<endl;，重新编译运行，结果是否有变化？

【分析与解答】（1）因为类的成员函数都自带 this 指针，在大部分程序中一般不显式地写出 this 指针，但实际上与显式写出 this 指针效果是一样的，因此，改写后带 this 指针的等效函数代码为：

```
void Date:: SetDate(int y,  int m,  int d)
{   this->year=y;
    this->month=m;
    this->day=d;
}
```

（2）Display( )中的最后一条语句改写之后重新编译运行，程序运行结果没有变化，因为 this 指针在函数中一般缺省，所以改写后结果不会有变化。

6．例 3.9 的思考与练习。

将例 3.9 的 main()函数第 1 条语句 Date today(2006,10,17);修改为 Date today;，重新编译肯定有错误信息，如何以最简单的方式修改程序，保证 main()函数第 1 条语句为 Date today;时程序运行结果与例 3.9 的运行结果相同？

【分析与解答】直接修改为 Date today;后编译器会指出这条语句非法，并给出错误原因“no appropriate default constructor available”，即找不到合适的构造函数。而这一点是初学者很容易疏漏的地方，在本章的后一节会重点讲述，在此提醒读者注意。在类的定义中，只定义了一个带 3 个形式参数的构造函数，要求对象提供对应的 3 个实参。此时，修改的方法，结合第 2 章的知识，函数是可以带默认参数值的，构造函数也一样。因此，最简单的解决方案就是在类内声明构造函数原型的时候提供 3 个 默认参数值，即将语句 Date(int , int , int); 修改成 Date(int=2006 , int=10 , int=17);，则程序的运行结果就与例 3.9 完全相同。

7．构造函数的思考与练习。

假设类定义中已有两个如下定义的构造函数：

```
Date(int y, int m, int d );
Date ( ) ;
```

那么下面的语句合法吗？为什么？

```
Date day1 ( 2011,5,1 ) ;
Date day2;
Date day3 (2011);
Date day4 ( ) ;
```

【分析与解答】

```
Date day1 ( 2011,5,1 ) ;      //合法
Date day2;                    //合法
Date day3 (2011);             //非法
Date day4 ( ) ;               //有问题
```

前两个对象的定义是合法的，因为分别可以自动调用有 3 个形式参数和无参的构造函数；对象 day3 的定义非法，因为程序中没有提供只需要 1 个实参的构造函数，已定义的带参构造函数不具有默认值，因此该对象找不到可以自动调用的构造函数版本；最后一条有问题的语句严格说并不是非法的，只是该语句实际的意思也许并不是编程者原本希望的那样。如果认为它是对名为

day4 的一个对象的说明，就是错误的。它实际上可以理解为一个名为 day4 的函数的声明，该函数不带参数，返回 Date 型的一个值。从理论上讲虽然合法，但是根据实际情况来说是非法的。如果是声明一个名为 day4 的对象不提供实参，则应采用与 day2 一样的方式定义，即为：Date day4;。

8．程序 example3_10.cpp 的思考与练习。

在 example3_10.cpp 的 main()函数一开始增加一条语句：Date otherday(2011,4);，同时在 return 0;之前增加一条语句：otherday.Display( );，重新编译程序，是否正确？新增的最后一行输出结果是什么？

【分析与解答】本程序中的构造函数提供了 3 个默认参数值，因此在定义对象时需要提供的实际参数的个数可以是 0 至 3 个，因此所增加的对象 otherday 提供的前两个实际参数是正确的，由于增加了最后一条调用 Display 的语句，最后一行新增加的输出为：**2011-4-1** 。

9．程序 example3_11.cpp 后的思考与练习。

（1）假设类 Point 有 3 个重载的构造函数，其类内声明的原型分别为：

```
Point( int x, int y) ;  Point ( ) ;  Point ( const Point &t) ;
```

有如下关于二维平面坐标点类型的定义语句，请分析各自调用那种构造函数：

（a）Point　p1(3,5)；　（b）Point p2;　（c）Point　p3(p1);

（d）Point　p4 = p1;　（e）p2= Point(4,5) ;

（2）拷贝构造函数仅能实现原样复制吗？如何实现在复制的基础上加以变化？

【分析与解答】（1）定义新对象时均要调用构造函数，而匹配重载函数的基本原则就是根据实际参数与形式参数对应的规则，根据实际参数是否是类的对象可以判断调用的是拷贝构造函数还是普通构造函数，因此以上定义的 5 个对象：（a）调用带参的构造函数；（b）调用无参构造函数；（c）调用拷贝构造函数；（d）调用拷贝构造函数，此语句等效于 Point　p4(p1)；（e）调用构造函数，p2 已经定义，由 Point(4,5)构造一个无名对象，并将对象的成员值赋值给 p2，通过默认的赋值符号重载函数完成（见第 7 章多态性），此处不是调用的拷贝构造函数。

（2）拷贝构造函数可以根据实际需要对函数体进行变化，从而以参数为基础复制出一个全新的对象。例如：

```
Date::Date( const Date &date)                    //拷贝构造函数的定义
{   year = date.year+1;
    month = date.month;
    day = date.day;
    cout<<"Copy Constructor called.\n";
}
```

通过语句 year = date.year+1;得到的新对象将是原对象一周年以后的日期，此时得到的新对象与实参就不完全一样了。

10．程序 example3_12.cpp 后的思考与练习。

将程序 example3_11.cpp 中的文件包含指令由 #include"Date3.h" 改为#include"Date4.h" ，重新编译链接运行，结果如何？试分析。

【分析与解答】程序运行结果如下：分析见注释

```
Constructor called.              //定义对象 day1 调用普通的带参构造函数
Constructor called.              //定义对象 day3 调用普通的构造函数，使用默认参数值
Copy Constructor called.         //定义对象 day2 调用拷贝构造函数
Copy Constructor called.         //定义对象 day4 调用拷贝构造函数
```

```
Copy Constructor called.          //调用 f 函数，实参 day2 传值给形参 Q 时调用拷贝构造函数
Copy Constructor called.          //f 函数内部，定义对象 R 时以 Q 为实参调用拷贝构造函数
Copy Constructor called.          //f 函数返回一个类对象时调用拷贝构造函数
Destructor called.                //以下 2 行是函数调用结束时引起的析构函数的调用，其
Destructor called.                //顺序与调用函数所引起的构造函数的调用顺序完全相反
Destructor called.                //析构用于存放函数返回值时的无名对象
2011-5-26                         //调用输出函数输出对象 day3 的值
Destructor called.                //对象 day4 生存期结束，调用析构函数
Destructor called.                //对象 day2 生存期结束，调用析构函数
Destructor called.                //对象 day3 生存期结束，调用析构函数
Destructor called.                //对象 day1 生存期结束，调用析构函数
```

11．程序 example3_17.cpp 的思考与练习。

将程序 example3_17.cpp 中的语句 Date &pDate=DateA; 修改为两条语句 Date &pDate; 和 pDate=DateA;，重新编译，观察错误信息并解释原因。

【分析与解答】无论是对象引用或是普通变量引用，都要求在定义之初作初始化，并且一旦经过初始化，在程序中的别名关系将不会被更改。以上的修改，是先定义一个引用 pDate，再对其进行赋值，试图通过赋值的方式使其获得指代的对象，但这是不对的，赋值与初始化有着本质的区别。因此上机编译时共出现 4 个报错信息：

```
example3_17.cpp(5) : error C2530: 'pDate' : references must be initiaexamplezed
example3_17.cpp(6) : error C2501: 'pDate' : missing storage-class or type specifiers
example3_17.cpp(6) : error C2040: 'pDate' : 'int' differs in levels of indirection from
'class Date &'
example3_17.cpp(6) : error C2440: 'initializing' : cannot convert from 'class Date'
to 'int'
```

12．程序 example3_21.cpp 的思考与练习。

（1）将主函数中作为左值的函数调用语句由 Fun(DateA)=Date(1998,10,5);改为

Fun(tDate)=Date(1998,10,5);，重新运行程序，请写出现在输出结果的最后 6 行，并分析与例题结果最后 6 行的区别及原因。

（2）将本例中的函数修改为

```
Date & Fun(Date  &pDate)          //普通函数 Fun()的定义，对象引用作形参，并返回引用
{   Date qDate;                   //定义局部自动对象 qDate
    qDate.ModifyDate(2011,  10,  20);
    return qDate;                 //引用返回 qDate，不安全，有告警
}
```

重新编译会产生告警信息，为什么？

【分析与解答】（1）语句的修改其实就体现在函数调用时实在参数由 dateA 改为了 tDate，这时 pDate 成为实参 tDate 的别名，因此函数内修改的是 tData，返回的其实也是 tDate，最后通过赋值 tDate 获得了无名对象的值 1998-10-5，而这一调用过程与 DateA 无关，因此它还保持前面得到的 2012-5-1 这个值。函数运行最后 6 行的结果为：

```
After left Fun, DateA:
2012-5-1                          //此结果与例 3.21 有变化
After left Fun, tDate:
1998-10-5                         //此结果与例 3.21 有变化
```