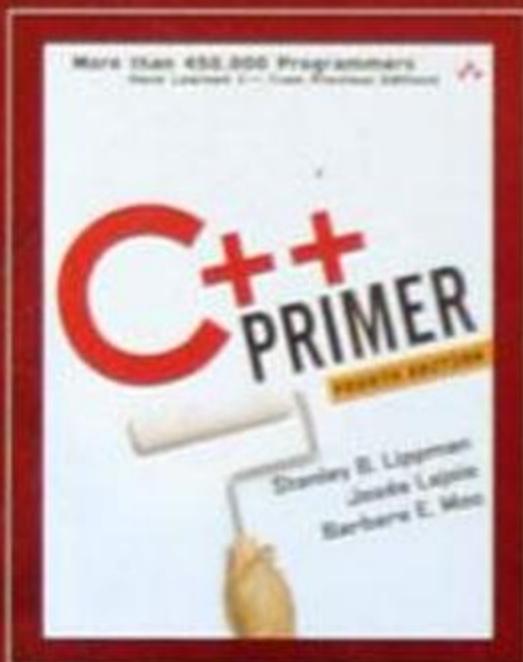


# C++ Primer (第4版) 习题解答

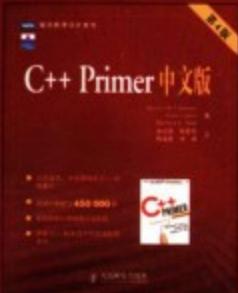
蒋爱军 李师贤 梅晓勇 编著

- 学习 C++ 的最佳伴侣
- 涵盖所有习题的答案
- 代码均配有详细注释

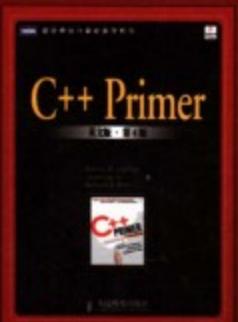


人民邮电出版社  
POSTS & TELECOM PRESS

## 本书配套



书名: C++ Primer中文版(第4版)  
ISBN: 7-115-14554-7/TP·5273  
定价: 99.00元



书名: C++ Primer(英文版·第4版)  
ISBN: 7-115-15169-5/TP·5642  
定价: 99.00元

本书相关信息请访问: 图灵网站 <http://www.turingbook.com>  
读者/作者热线: (010)88503802  
反馈/投稿/推荐信箱: contact@turingbook.com

**分类建议** 计算机 / 程序设计 / C++

人民邮电出版社网址 [www.ptpress.com.cn](http://www.ptpress.com.cn)

ISBN 978-7-115-15510-8



9 787115 155108 >

定价: 45.00 元

图灵社区  
老书新读  
PDG

**TURING** 图灵程序设计丛书

# C++ Primer (第4版) 习题解答

蒋爱军 李师贤 梅晓勇 编著

人民邮电出版社  
北京



## 图书在版编目 (CIP) 数据

C++ Primer (第 4 版) 习题解答 / 蒋爱军, 李师贤, 梅晓勇著. —北京: 人民邮电出版社, 2007.2  
(图灵程序设计丛书)

ISBN 978-7-115-15510-8

I. C... II. ①蒋...②李...③梅... III. C 语言—程序设计—解题 IV. TP312-44  
中国版本图书馆 CIP 数据核字 (2006) 第 139483 号

## 内 容 提 要

*C++ Primer*(第 4 版)是 C++ 大师 Stanley B. Lippman 丰富的实践经验和 C++ 标准委员会原负责人 Josée Lajoie 对 C++ 标准深入理解的完美结合, 更加入了 C++ 先驱 Barbara E. Moo 在 C++ 教学方面的真知灼见, 是初学者的最佳 C++ 指南, 而且对于中高级程序员, 也是不可或缺的参考书。本书正是这部久负盛名的 C++ 经典教程的配套习题解答。书中提供了 *C++ Primer*(第 4 版) 中所有习题的参考答案。

本书对使用 *C++ Primer*(第 4 版) 学习 C++ 程序设计语言的读者是非常理想的参考书。

图灵程序设计丛书

### C++ Primer (第 4 版) 习题解答

- 
- ◆ 编 著 蒋爱军 李师贤 梅晓勇
  - 责任编辑 杨海玲
  - ◆ 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 电子函件 315@ptpress.com.cn
  - 网址 <http://www.ptpress.com.cn>
  - 北京顺义振华印刷厂印刷
  - 新华书店总店北京发行所经销
  - ◆ 开本: 800 × 1000 1/16
  - 印张: 21.75
  - 字数: 528 千字 2007 年 2 月第 1 版
  - 印数: 1—5 000 册 2007 年 2 月北京第 1 次印刷
- 

ISBN 978-7-115-15510-8/TP · 5831

定价: 45.00 元

读者服务热线: (010) 88593802 印装质量热线: (010) 67129223



# 前 言

C++是一门非常实用的程序设计语言，既支持过程式程序设计，也支持面向对象程序设计，因而也是目前应用极为广泛的一门程序设计语言。

在层出不穷的介绍 C++ 语言的书籍中，*C++ Primer* 是一本广受欢迎的权威之作。强大的作者阵容、全面的内容介绍、新颖的组织方式，使之深受 C++ 爱好者的青睐。本书编者在翻译 *C++ Primer*（第 4 版）的过程中也深深地感受到了这一点。

在学习一门程序设计语言的过程中，亲自动手编写代码是一种极其有效的学习方式，可以对语言的理解和应用达到事半功倍的效果，因此，*C++ Primer*（第 4 版）<sup>1</sup> 中提供了许多习题，以帮助读者加深对书中内容的理解。

本书试图成为 *C++ Primer*（第 4 版）的配套书籍，根据 *C++ Primer*（第 4 版）中所介绍的内容提供配套习题的解答，书中所给出的“见 xx 节”，均指参见 *C++ Primer*（第 4 版）的相应章节。

本书中给出的程序均已通过 Microsoft Visual C++ .NET 2003 的编译。源文件（实现文件）以.cpp 为扩展名，头文件为了与此对应采用.hpp 为扩展名（而没有采用编译器的默认扩展名.h）。为了节省篇幅，有些程序中将类的定义与使用类的主函数放在同一实现文件中。包含主函数的源文件根据习题编号命名。大多数模板的定义都没有区分头文件和实现文件（因为编者所用的编译器支持模板的包含编译模型）。另外，使用 Visual C++ .NET 2003 编译器的默认设置会自动连接一些默认库，因此可能有某些所用到的库函数或库类型没有显式指明相应的头文件。使用其他编译器的读者需特别注意，必要时应加上相应的#include 指示。

衷心希望本书能对使用 *C++ Primer*（第 4 版）学习 C++ 语言的读者有所帮助。

由于编者水平所限，书中不当之处在所难免，恳请读者批评指正。

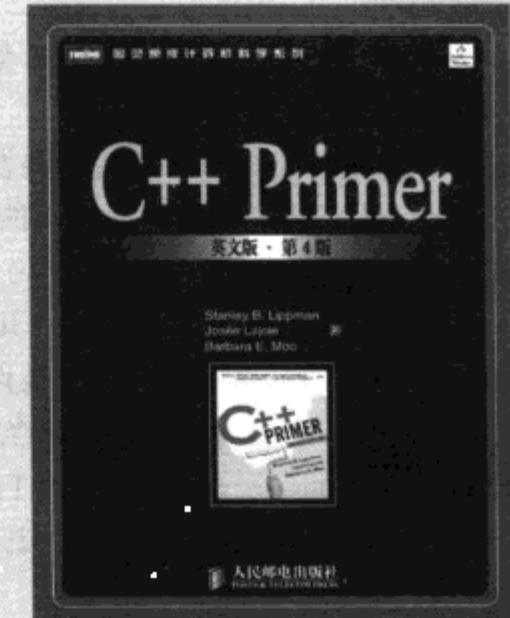
编 者

2006 年 10 月

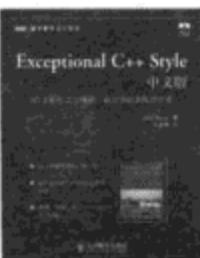
1. *C++ Primer*（第 4 版）的中文版及英文影印版均已由人民邮电出版社引进出版。——编者注



书名: C++ Primer中文版(第4版)  
原书名: C++ Primer  
作者: Stanley B.Lippman, Josee Lajoie, Barbara E.Moo  
译者: 李师贤 蒋爱军 梅晓勇 林瑛  
书号: 7-115-14554-7  
定价: 99.00元  
出版时间: 2006年3月



书名: C++ Primer(英文版·第4版)  
原书名: C++ Primer  
作者: Stanley B. Lippman, Josee Lajoie, Barbara E. Moo  
书号: 7-115-15169-7  
定价: 99.00元  
出版时间: 2006年10月



书名: Exceptional C++ Style中文版  
原书名: Exceptional C++ Style: 40 New Engineering Puzzles, Programming Problems, and Solutions  
作者: Herb Sutter  
译者: 刘未鹏  
书号: 7-115-14225-4  
定价: 39.00元  
出版时间: 2005年12月



书名: C++编程规范: 101条规则、准则与最佳实践  
原书名: C++ Coding Standards: 101 Rules, Guidelines, and Best Practices  
作者: Herb Sutter, Andrei Alexandrescu  
译者: 刘基诚  
书号: 7-115-14205-X  
定价: 35.00元  
出版时间: 2005年12月



书名: C++ 必知必会  
原书名: C++ Common Knowledge: Essential Intermediate Programming  
作者: Stephen C. Dewhurst  
译者: 荣耀  
书号: 7-115-14101-0  
定价: 29.00元  
出版时间: 2005年11月



书名: C++程序设计(英文版·第5版)  
原书名: Small C++ How to Program  
作者: H.M.Deitel, P.J.Deitel  
书号: 7-115-14151-7  
定价: 59.00元  
出版时间: 2005年12月



书名: C++Primer[英文版·第3版]  
原书名: C++ Primer  
作者: Stanley B. Lippman, Josee Lajoie  
书号: 7-115-14056-1  
定价: 69.00元  
出版时间: 2005年9月



书名: C语言程序设计(英文版·第3版)  
原书名: Programming in C  
作者: Stephen G. Kochan  
书号: 7-115-14763-9  
定价: 55.00元  
出版时间: 2006年5月

# 目 录

第 1 章 快速入门	1
第 2 章 变量和基本类型	11
第 3 章 标准库类型	22
第 4 章 数组和指针	36
第 5 章 表达式	53
第 6 章 语句	64
第 7 章 函数	81
第 8 章 标准 IO 库	98
第 9 章 顺序容器	108
第 10 章 关联容器	132
第 11 章 泛型算法	154
第 12 章 类	169
第 13 章 复制控制	189
第 14 章 重载操作符与转换	208
第 15 章 面向对象编程	234
第 16 章 模板与泛型编程	269
第 17 章 用于大型程序的工具	306
第 18 章 特殊工具与技术	323



# 第1章

## 快 速 入 门

### 习题1.1

查看所用的编译器文档，了解它所用的文件命名规范。编译并运行本节的main程序。

#### 【解答】

一般而言，C++编译器要求待编译的程序保存在文件中。C++程序中一般涉及两类文件：头文件和源文件。大多数系统中，文件的名字由文件名和文件后缀（又称扩展名）组成。文件后缀通常表明文件的类型，如头文件的后缀可以是.h或.hpp等；源文件的后缀可以是.cc或.cpp等，具体的后缀与使用的编译器有关。通常可以通过编译器所提供的联机帮助文档了解其文件命名规范。

### 习题1.2

修改程序使其返回-1。返回值-1通常作为程序运行失败的指示器。然而，系统不同，如何（甚至是否）报告main函数运行失败也不同。重新编译并再次运行程序，看看你的系统如何处理main函数的运行失败指示器。

#### 【解答】

笔者所使用的Windows操作系统并不报告main函数的运行失败，因此，程序返回-1或返回0在运行效果上没有什么区别。但是，如果在DOS命令提示符方式下运行程序，然后再键入echo %ERRORLEVEL%命令，则系统会显示返回值-1。

### 习题1.3

编一个程序，在标准输出上打印“Hello, World”。

#### 【解答】

```
#include<iostream>

int main()
{
    std::cout << "Hello, World" << std::endl;
    return 0;
}
```

### 习题1.4

我们的程序利用内置的加法操作符“+”来产生两个数的和。编写程序，使用乘法操作符“\*”产生两个数的积。

**【解答】**

```
#include <iostream>

int main()
{
    std::cout << "Enter two numbers:" << std::endl;
    int v1, v2;
    std::cin >> v1 >> v2;
    std::cout << "The product of " << v1 << " and " << v2
        << " is " << v1 * v2 << std::endl;

    return 0;
}
```

**习题1.5**

我们的程序使用了一条较长的输出语句。重写程序，使用单独的语句打印每一个操作数。

**【解答】**

```
#include <iostream>

int main()
{
    std::cout << "Enter two numbers:" << std::endl;
    int v1, v2;
    std::cin >> v1 >> v2;
    std::cout << "The sum of ";
    std::cout << v1;
    std::cout << " and ";
    std::cout << v2;
    std::cout << " is ";
    std::cout << v1 + v2 ;
    std::cout << std::endl;

    return 0;
}
```

**习题1.6**

解释下面的程序段：

```
std::cout << "The sum of " << v1;
    << " and " << v2;
    << " is " << v1 + v2
    << std::endl;
```

这段代码合法吗？如果合法，为什么？如果不合法，又为什么？

**【解答】**

这段代码不合法。

注意，第1、2、4行的末尾有分号，表示这段代码包含三条语句，即第1、2行各为一个语句，第3、4行构成一个语句。“<<”为二元操作符，在第2、3两条语句中，第一个“<<”缺少左操作数，因此不合法。

在第2、3行的开头加上“std::cout”，即可更正。

**习题1.7**

编译有不正确嵌套注释的程序。

**【解答】**

由注释对嵌套导致的编译器错误信息通常令人迷惑。例如，在笔者所用的编译器中编译1.3节中给出的带有不正确嵌套注释的程序：

```
#include <iostream>
/*
 * comment pairs /* */ cannot nest.
 * "cannot nest" is considered source code,
 * as is the rest of the program
 */
int main()
{
    return 0;
}
```

编译器会给出如下错误信息：

```
error C2143: syntax error : missing ';' before '<'
error C2501: 'include' : missing storage-class or type specifiers
warning C4138: '/*' found outside of comment      (第6行)
error C2143: syntax error : missing ';' before '('      (第8行)
error C2447: '(' : missing function header (old-style formal list?) (第8行)
```

**习题1.8**

指出下列输出语句哪些（如果有）是合法的。

```
std::cout << "/*";
std::cout << "*/";
std::cout << /* */ */;
```

预测结果，然后编译包含上述三条语句的程序，检查你的答案。纠正所遇到的错误。

**【解答】**

第一条和第二条语句合法。

第三条语句中<<操作符之后至第二个双引号之前的部分被注释掉了，导致<<操作符的右操作数不是一个完整的字符串，所以不合法。在分号之前加上一个双引号即可更正。

**习题1.9**

下列循环做什么？sum的最终值是多少？

```
int sum = 0;
for (int i = -100; i <= 100; ++i)
    sum += i;
```

**【解答】**

该循环求-100~100之间所有整数的和（包括-100和100）。

sum的最终值是0。

**习题1.10**

用for循环编程，求从50~100的所有自然数的和。然后用while循环重写该程序。

**【解答】**

用for循环编写的程序如下：

```
#include <iostream>

int main()
{
    int sum = 0;
    for (int i = 50; i <= 100; ++i)
        sum += i;
    std::cout << "Sum of 50 to 100 inclusive is "
              << sum << std::endl;

    return 0;
}
```

用while循环编写的程序如下：

```
#include <iostream>

int main()
{
    int sum = 0, int i = 50;
    while (i <= 100) {
        sum += i;
        ++i;
    }
    std::cout << "Sum of 50 to 100 inclusive is "
              << sum << std::endl;
    return 0;
}
```

**习题1.11**

用while循环编程，输出10~0递减的自然数。然后用for循环重写该程序。

**【解答】**

用while循环编写的程序如下：

```
#include <iostream>

int main()
{
    int i = 10;
    while (i >= 0) {
        std::cout << i << " ";
        --i;
    }

    return 0;
}
```

用for循环编写的程序如下：

```
#include <iostream>
int main()
{
    for (int i = 10; i >= 0; --i)
        std::cout << i << " ";
```

```

    return 0;
}

```

**习题1.12**

对比前面两个习题中所写的循环。两种形式各有何优缺点？

**【解答】**

在for循环中，循环控制变量的初始化和修改都放在语句头部分，形式较简洁，且特别适用于循环次数已知的情况。在while循环中，循环控制变量的初始化一般放在while语句之前，循环控制变量的修改一般放在循环体中，形式上不如for语句简洁，但它比较适用于循环次数不易预知的情况（用某一条件控制循环）。两种形式各有优点，但它们在功能上是等价的，可以相互转换。

**习题1.13**

编译器不同，理解其诊断内容的难易程度也不同。编写一些程序，包含本小节“再谈编译”部分讨论的那些常见错误。研究编译器产生的信息，这样你在编译更复杂的程序遇到这些信息时不会陌生。

**【解答】**

对于程序中出现的错误，编译器通常会给出简略的提示信息，包括错误出现的文件及代码行、错误代码、错误性质的描述。如果要获得关于该错误的详细信息，一般可以根据编译器给出的错误代码在其联机帮助文档中查找。

**习题1.14**

如果输入值相等，本节展示的程序将产生什么问题？

**【解答】**

sum的值即为输入值。因为输入的v1和v2值相等（假设为x），所以lower和upper相等，均为x。for循环中的循环变量val初始化为lower，从而val<=upper为真，循环体执行一次，sum的值为val（即输入值x）；然后val加1，val的值就大于upper，循环执行结束。

**习题1.15**

用两个相等的值作为输入编译并运行本节中的程序。将实际输出与你在习题1.14中所做的预测相比较，解释实际结果和你预计的结果间的不相符之处。

**【解答】**

运行1.4.3节中给出的程序，输入两个相等的值（例如3,3），则程序输出为：

```
Sum of 3 to 3 inclusive is 3
```

与习题1.14中给出的预测一致。

**习题1.16**

编写程序，输出用户输入的两个数中的较大者。

**【解答】**

```
#include <iostream>
```

```

int main()
{
    std::cout << "Enter two numbers:" << std::endl;
    int v1, v2;
    std::cin >> v1 >> v2; // 读入数据

    if (v1 >= v2)
        std::cout << "The bigger number is" << v1 << std::endl;
    else
        std::cout << "The bigger number is" << v2 << std::endl;

    return 0;
}

```

**习题1.17**

编写程序，要求用户输入一组数。输出信息说明其中有多少个负数。

**【解答】**

```

#include <iostream>

int main()
{
    int amount = 0, value;

    // 读入数据直到遇见文件结束符，计算所读入的负数的个数
    while (std::cin >> value)
        if (value <= 0)
            ++amount;

    std::cout << "Amount of all negative values read is"
        << amount << std::endl;

    return 0;
}

```

**习题1.18**

编写程序，提示用户输入两个数并将这两个数范围内的每个数写到标准输出。

**【解答】**

```

#include <iostream>

int main()
{
    std::cout << "Enter two numbers:" << std::endl;
    int v1, v2;
    std::cin >> v1 >> v2; // 读入两个数

    // 用较小的数作为下界lower、较大的数作为上界upper
    int lower, upper;
    if (v1 <= v2) {
        lower = v1;
        upper = v2;
    } else {
        lower = v2;
        upper = v1;
    }

    // 输出从lower到upper之间的值
    std::cout << "Values of " << lower << "to "

```



```

        << upper << "inclusive are: " << std::endl;
for (int val = lower; val <= upper; ++val)
    std::cout << val << " ";
return 0;
}

```

**习题1.19**

如果上题给定数1000和2000，程序将产生什么结果？修改程序，使每一行输出不超过10个数。

**【解答】**

所有数的输出连在一起，不便于阅读。

程序修改如下：

```

#include <iostream>

int main()
{
    std::cout << "Enter two numbers:" << std::endl;
    int v1, v2;
    std::cin >> v1 >> v2; // 读入两个数

    // 用较小的数作为下界lower、较大的数作为上界upper
    int lower, upper;
    if (v1 <= v2) {
        lower = v1;
        upper = v2;
    } else {
        lower = v2;
        upper = v1;
    }

    // 输出从lower到upper之间的值
    std::cout << "Values of " << lower << "to "
        << upper << "inclusive are: " << std::endl;
    for (int val = lower, count=1; val <= upper; ++val, ++count) {
        std::cout << val << " ";
        if (count % 10 == 0)           //每行输出10个值
            std::cout << std::endl;
    }

    return 0;
}

```

粗黑体部分为主要的修改：用变量count记录已输出的数的个数；若count的值为10的整数倍，则输出一个换行符。

**习题1.20**

编写程序，求用户指定范围内的数的和，省略设置上界和下界的if测试。假定输入数是7和3，按照这个顺序，预测程序运行结果。然后按照给定的数是7和3运行程序，看结果是否与你预测的相符。如果不相符，反复研究关于for和while循环的讨论直到弄清楚其中的原因。

**【解答】**

可编写程序如下：

```

// 1-20.cpp
// 省略设置上界和下界的if测试，求用户指定范围内的数的和

#include <iostream>

int main()
{
    std::cout << "Enter two numbers:" << std::endl;
    int v1, v2;
    std::cin >> v1 >> v2; // 读入数据

    int sum = 0;
    // 求和
    for (int val = v1; val <= v2; ++val)
        sum += val; // sum = sum + val

    std::cout << "Sum of " << v1
        << " to " << v2
        << " inclusive is "
        << sum << std::endl;

    return 0;
}

```

如果输入数据为7和3，则v1值为7，v2值为3。for语句头中将val的初始值设为7，第一次测试表达式val <= v2时，该表达式的值为false，for语句的循环体一次也不执行，所以求和结果sum为0。

### 习题1.21

本书配套网站 ([http://www.awprofessional.com/cpp\\_primer](http://www.awprofessional.com/cpp_primer)) 的第1章的代码目录下有Sales\_item.h源文件。复制该文件到你的工作目录。编写程序，循环遍历一组书的销售交易，读入每笔交易并将交易写至标准输出。

#### 【解答】

```

#include <iostream>
#include "Sales_item.h"

int main()
{
    Sales_item book;

    // 读入ISBN，售出书的本数，销售价格
    std::cout << "Enter transactions:" << std::endl;
    while (std::cin >> book) {
        // 输出ISBN，售出书的本数，总收入，平均价格
        std::cout << "ISBN, number of copies sold,"
            << "total revenue, and average price are:"
            << std::endl;
        std::cout << book << std::endl;
    }

    return 0;
}

```

### 习题1.22

编写程序，读入两个具有相同ISBN的Sales\_item对象并产生它们的和。

**【解答】**

```
#include <iostream>
#include "Sales_item.h"

int main()
{
    Sales_item trans1, trans2;

    // 读入交易
    std::cout << "Enter two transactions:" << std::endl;
    std::cin >> trans1 >> trans2;

    if (trans1.same_isbn(trans2))
        std::cout << "The total information: " << std::endl
            << "ISBN, number of copies sold, "
            << "total revenue, and average price are:"
            << std::endl << trans1 + trans2;
    else
        std::cout << "The two transactions have different ISBN."
            << std::endl;

    return 0;
}
```

**习题1.23**

编写程序，读入几个具有相同ISBN的交易，输出所有读入交易的和。

**【解答】**

```
#include <iostream>
#include "Sales_item.h"

int main()
{
    Sales_item total, trans;

    // 读入交易
    std::cout << "Enter transactions:" << std::endl;

    if (std::cin >> total) {
        while (std::cin >> trans)
            if (total.same_isbn(trans)) // ISBN相同
                total = total + trans;
            else { // ISBN不同
                std::cout << "Different ISBN." << std::endl;
                return -1;
            }
        // 输出交易之和
        std::cout << "The total information: " << std::endl
            << "ISBN, number of copies sold, "
            << "total revenue, and average price are:"
            << std::endl << total;
    }
    else {
        std::cout << "No data?!" << std::endl;
        return -1;
    }

    return 0;
}
```

**习题1.24**

编写程序，读入几笔不同的交易。对于每笔新读入的交易，要确定它的ISBN是否和以前的交易的ISBN一样，并且记下每一个ISBN的交易的总数。通过给定多笔不同的交易来测试程序。这些交易必须代表多个不同的ISBN，但是每个ISBN的记录应分在同一组。

**【解答】**

```
#include <iostream>
#include "Sales_item.h"

int main()
{
    // 声明变量以保存交易记录以及具有相同ISBN的交易的数目
    Sales_item trans1, trans2;
    int amount;

    // 读入交易
    std::cout << "Enter transactions:" << std::endl;
    std::cin >> trans1;
    amount=1;
    while (std::cin >> trans2)
        if (trans1.same_isbn(trans2)) // ISBN相同
            ++amount;
        else {
            // ISBN不同
            std::cout << "Transaction amount of previous ISBN: "
                << amount << std::endl;
            trans1 = trans2;
            amount=1;
        }

    // 输出最后一个ISBN的交易数目
    std::cout << "Transaction amount of the last ISBN: "
        << amount << std::endl;
}

return 0;
}
```

**习题1.25**

使用源自本书配套网站的Sales\_item.h头文件，编译并执行1.6节给出的书店程序。

**【解答】**

可从C++ Primer (第4版) 的配套网站 ([http://www.awprofessional.com/cpp\\_primer](http://www.awprofessional.com/cpp_primer)) 下载头文件 Sales\_item.h，然后使用该头文件编译并执行1.6节给出的书店程序。

**习题1.26**

在书店程序中，我们使用了加法操作符而不是复合赋值操作符将trans加到total中，为什么我们不使用复合赋值操作符？

**【解答】**

因为在1.5.1节中提及的 Sales\_item 对象上的操作中只包含了+和=，没有包含+=操作。（但事实上，使用 Sales\_item.h 文件，已经可以用+=操作符取代=和+操作符的复合使用。）

## 变量和基本类型

### 习题2.1

int、long和short类型之间有什么差别？

#### 【解答】

它们的最小存储空间不同，分别为16位、32位和16位。一般而言，short类型为半个机器字（word）长，int类型为一个机器字长，而long类型为一个或两个机器字长（在32位机器中，int类型和long类型的字长通常是相同的）。因此，它们的表示范围不同。

### 习题2.2

unsigned和signed类型有什么差别？

#### 【解答】

前者为无符号类型，只能表示大于或等于0的数。后者为带符号类型，可以表示正数、负数和0。

### 习题2.3

如果在某机器上short类型占16位，那么可以赋给short类型的最大数是什么？unsigned short类型的最大数又是什么？

#### 【解答】

若在某机器上short类型占16位，那么可以赋给short类型的最大数是 $2^{15}-1$ ，即32767；而unsigned short类型的最大数为 $2^{16}-1$ ，即65535。

### 习题2.4

当给16位的unsigned short对象赋值100000时，赋的值是什么？

#### 【解答】

34464。

100000超过了16位的unsigned short类型的表示范围，编译器对其二进制表示截取低16位，相当于对65536求余（求模，%），得34464。

### 习题2.5

float类型和double类型有什么差别？

**【解答】**

二者的存储位数不同(一般而言, float类型为32个二进制位, double类型为64个二进制位), 因而取值范围不同, 精度也不同(float类型只能保证6位有效数字, 而double类型至少能保证10位有效数字)。

**习题2.6**

要计算抵押贷款的偿还金额, 利率、本金和付款额应分别选用哪种类型? 解释你选择的理由。

**【解答】**

利率可以选择float类型, 因为利率通常为百分之几。一般只保留到小数点后两位, 所以6位有效数字就足以表示了。

本金可以选择long类型, 因为本金通常为整数。long类型可表示的最大整数一般为 $2^{31}-1$ (即2147483647), 应该足以表示了。

付款额一般为实数, 可以选择double类型, 因为float类型的6位有效数字可能不足以表示。

**习题2.7**

解释下列字面值常量的不同之处。

- (a) 'a', L'a', "a", L"a"
- (b) 10, 10u, 10L, 10uL, 012, 0xC
- (c) 3.14, 3.14f, 3.14L

**【解答】**

- (a) 'a', L'a', "a", L"a"

'a'为char型字面值, L'a'为wchar\_t型字面值, "a"为字符串字面值, L"a"为宽字符串字面值。

- (b) 10, 10u, 10L, 10uL, 012, 0xC

10为int型字面值, 10u为unsigned型字面值, 10L为long型字面值, 10uL为unsigned long型字面值, 012为八进制表示的int型字面值, 0xC为十六进制表示的int型字面值。

- (c) 3.14, 3.14f, 3.14L

3.14为double型字面值, 3.14f为float型字面值, 3.14L为long double型字面值。

**习题2.8**

确定下列字面值常量的类型:

- (a) -10 (b) -10u (c) -10. (d) -10e-2

**【解答】**

- (a) int型

- (b) unsigned int型

- (c) double型

- (d) double型

**习题2.9**

下列哪些(如果有)是非法的?

- (a) "Who goes with F\145rgus?\012"
- (b) 3.14e1L
- (c) "two" L"some"
- (d) 1024f
- (e) 3.14UL
- (f) "multiple line  
comment"

**【解答】**

- (c) 非法。因为字符串字面值与宽字符串字面值的连接是未定义的。
- (d) 非法。因为整数1024后面不能带后缀f。
- (e) 非法。因为浮点字面值不能带后缀U。
- (f) 非法。因为分两行书写的字符串字面值必须在第一行的末尾加上反斜线。

**习题2.10**

使用转义字符编写一段程序，输出2M，然后换行。修改程序，输出2，跟着一个制表符，然后是M，最后是换行符。

**【解答】**

输出2M、然后换行的程序段：

```
// 输出"2M"和换行字符
std::cout << "2M" << '\n';
```

修改后的程序段：

```
// 输出'2', '\t', 'M'和换行字符
std::cout << '2' << '\t' << 'M' << '\n';
```

**习题2.11**

编写程序，要求用户输入两个数——底数（base）和指数（exponent），输出底数的指数次方的结果。

**【解答】**

```
#include <iostream>

int main()
{
    // 局部对象
    int base, exponent;
    long result=1;

    // 读入底数(base) 和指数(exponent)
    std::cout << "Enter base and exponent:" << std::endl;
    std::cin >> base >> exponent;

    if (exponent < 0) {
        std::cout << "Exponent can't be smaller than 0" << std::endl;
        return -1;
    }

    if (exponent > 0) {
        // 计算底数的指数次方
        for (int cnt = 1; cnt <= exponent; ++cnt)
            result *= base;
    }
}
```

```

        std::cout << base
            << " raised to the power of "
            << exponent << ":" "
            << result << std::endl;

    return 0;
}

```

**习题2.12**

区分左值和右值，并举例说明。

**【解答】**

左值 (lvalue) 就是变量的地址，或者是一个代表“对象在内存中的位置”的表达式。

右值 (rvalue) 就是变量的值，见2.3.1节。

变量名出现在赋值运算符的左边，就是一个左值；而出现在赋值运算符右边的变量名或字面常量就是一个右值。

例如：

```
val1=val2/8
```

这里的val1是个左值，而val2和8都是右值。

**习题2.13**

举出一个需要左值的例子。

**【解答】**

赋值运算符的左边（被赋值的对象）需要左值，见习题2.12。

**习题2.14**

下面哪些（如果有）名字是非法的？更正每个非法的标识符名字。

- |                           |                       |
|---------------------------|-----------------------|
| (a) int double = 3.14159; | (b) char _;           |
| (c) bool catch-22;        | (d) char 1_or_2 ='1'; |
| (e) float Float = 3.14f;  |                       |

**【解答】**

(a) double是C++语言中的关键字，不能用作用户标识符，所以非法。此语句可改为：double dval = 3.14159;。

(c) 名字catch-22中包含在字母、数字和下划线之外的字符“-”，所以非法。可将其改为：catch\_22;。

(d) 名字1\_or\_2非法，因为标识符必须以字母或下划线开头，不能以数字开头。可将其改为：one\_or\_two;。

**习题2.15**

下面两个定义是否不同？有何不同？



```
int month = 9, day = 7;
int month =09, day = 07;
```

如果上述定义有错的话，那么应该怎样改正呢？

### 【解答】

这两个定义不同。前者定义了两个int型变量，初值分别为9和7；后者也定义了两个int型变量，其中day被初始化为八进制值7；而month的初始化有错：试图将month初始化为八进制值09，但八进制数字范围为0~7，所以出错。可将第二个定义改为：

```
int month =011, day = 07;
```

### 习题2.16

假设calc是一个返回double对象的函数。下面哪些是非法定义？改正所有的非法定义。

- (a) int car = 1024, auto = 2048;
- (b) int ival = ival;
- (c) std::cin >> int input\_value;
- (d) double salary = wage = 9999.99;
- (e) double calc = calc();

### 【解答】

(a) 非法：auto是关键字，不能用作变量名。使用另一变量名，如aut即可更正。

(c) 非法：>>运算符后面不能进行变量定义。改为：

```
int input_value;
std::cin >> input_value;
```

(d) 非法：同一定义语句中不同变量的初始化应分别进行。改为：

```
double salary = 9999.99, wage = 9999.99;
```

注意，(b)虽然语法上没有错误，但这个初始化没有实际意义，ival仍是未初始化的。

### 习题2.17

下列变量的初始值（如果有）是什么？

```
std::string global_str;
int global_int;
int main()
{
    int local_int;
    std::string local_str;
    // ...
    return 0;
}
```

### 【解答】

global\_str和local\_str的初始值均为空字符串，global\_int的初始值为0，local\_int没有初始值。

**习题2.18**

解释下列例子中name的意义:

```
extern std::string name;
std::string name("exercise 3.5a");
extern std::string name("exercise 3.5a");
```

**【解答】**

第一条语句是一个声明，说明std::string变量name在程序的其他地方定义。

第二条语句是一个定义，定义了std::string变量name，并将name初始化为"exercise 3.5a"。

第三条语句也是一个定义，定义了std::string变量name，并将name初始化为"exercise 3.5a"，但这个语句只能出现在函数外部（即，name是一个全局变量）。

**习题2.19**

下列程序中j的值是多少？

```
int i = 42;
int main()
{
    int i = 100;
    int j = i;
    // ...
}
```

**【解答】**

j的值是100。j的赋值所使用到的i应该是main函数中定义的局部变量i，因为局部变量的定义会屏蔽全局变量的定义。

**习题2.20**

下列程序段将会输出什么？

```
int i = 100, sum = 0;
for (int i = 0; i != 10; ++i)
    sum += i;
std::cout << i << " " << sum << std::endl;
```

**【解答】**

输出为：

100 45

for语句中定义的变量i，其作用域仅限于for语句内部。输出的i值是for语句之前所定义的变量i的值。

**习题2.21**

下列程序合法吗？

```
int sum = 0;
for (int i = 0; i != 10; ++i)
    sum += i;
std::cout << "Sum from 0 to " << i
```

```
<< " is " << sum << std::endl;
```

**【解答】**

不合法。因为变量*i*具有语句作用域，只能在for语句中使用，输出语句中使用*i*属非法。

**习题2.22**

下列程序段虽然合法，但是风格很糟糕。有什么问题呢？怎样改善？

```
for (int i = 0; i < 100; ++i)
// process i
```

**【解答】**

问题主要在于使用了具体值100作为循环上界：100的意义在上下文中没有体现出来，导致程序的可读性差；若100这个值在程序中出现多次，则当程序的需求发生变化（如将100改变为200）时，对程序代码的修改复杂且易出错，导致程序的可维护性差。

改善方法：设置一个const变量（常量）取代100作为循环上界使用，并为该变量选择有意义的名字。

**习题2.23**

下列哪些语句合法？对于那些不合法的使用，解释原因。

- (a) const int buf;
- (b) int cnt = 0;
   
const int sz = cnt;
- (c) cnt++; sz++;

**【解答】**

(a) 不合法。因为定义const变量（常量）时必须进行初始化，而buf没有初始化。

(b) 合法。

(c) 不合法。因为修改了const变量sz的值。

**习题2.24**

下列哪些定义是非法的？为什么？如何改正？

- (a) int ival = 1.01;              (b) int &rval1 = 1.01;
- (c) int &rval2 = ival;              (d) const int &rval3 = 1;

**【解答】**

(b)非法。

因为rval1是一个非const引用，非const引用不能绑定到右值，而1.01是一个右值。可改正为：

```
int &rval1 = ival;
```

（假设ival是一个已定义的int变量）。

**习题2.25**

在习题2.24给出的定义下，下列哪些赋值是非法的？如果赋值合法，解释赋值的作用。

- (a) rval2 = 3.14159;              (b) rval2 = rval3;
- (c) ival = rval3;                  (d) rval3 = ival;

**【解答】**

(d)非法。因为rval3是一个const引用，不能进行赋值。

合法赋值的作用：

(a)将一个double型字面值赋给int型变量ival，发生隐式类型转换，ival得到的值为3。

(b)将int值1赋给变量ival。

(c)将int值1赋给变量ival。

**习题2.26**

(a)中的定义和(b)中的赋值存在哪些不同？哪些是非法的？

(a) `int ival = 0;`                    (b) `ival = ri;`  
`const int &ri = 0;`                    `ri = ival;`

**【解答】**

<code>int ival = 0;</code>	定义ival为int变量，并将其初始化为0。
<code>const int &amp;ri = 0;</code>	定义ri为const引用，并将其绑定到右值0。
<code>ival = ri;</code>	将0值赋给ival。
<code>ri = ival;</code>	试图对ri赋值，这是非法的，因为ri是const引用，不能赋值。

**习题2.27**

下列代码输出什么？

```
int i, &ri = i;
i = 5; ri = 10;
std::cout << i << " " << ri << std::endl;
```

**【解答】**

输出：

10 10

ri是i的引用，对ri进行赋值，实际上相当于对i进行赋值，所以输出i和ri的值均为10。

**习题2.28**

编译以下程序，确定你的编译器是否会警告遗漏了类定义后面的分号。

```
class Foo {
    // empty
} // Note: no semicolon
int main()
{
    return 0;
}
```

如果编译器的诊断结果难以理解，记住这些信息以备后用。

**【解答】**

在笔者所用的编译器中编译上述程序，编译器会给出如下错误信息：

```

error C2628: 'Foo' followed by 'int' is illegal (did you forget a ';'?)      (第4行)
warning C4326: return type of 'main' should be 'int or void' instead of 'Foo' (第5行)
error C2440: 'return' : cannot convert from 'int' to 'Foo'                  (第6行)

```

也就是说，该编译器会对遗漏了类定义后面的分号给出提示。

### 习题2.29

区分类中的public部分和private部分。

#### 【解答】

类中public部分定义的成员在程序的任何部分都可以访问。通常在public部分放置操作，以便程序中的其他部分可以执行这些操作。

类中private部分定义的成员只能被作为类的组成部分的代码（以及该类的友元）访问。通常在private部分放置数据，以对对象的内部数据进行隐藏。

### 习题2.30

定义表示下列类型的类的数据成员：

- |           |            |
|-----------|------------|
| (a) 电话号码  | (b) 地址     |
| (c) 员工或公司 | (d) 某大学的学生 |

#### 【解答】

(a) 电话号码

```

class Tel_number {
public:
    //...对象上的操作
private:
    std::string country_number;
    std::string city_number;
    std::string phone_number;
};

```

(b) 地址

```

class Address {
public:
    //...对象上的操作
private:
    std::string country;
    std::string city;
    std::string street;
    std::string number;
};

```

(c) 员工或公司

```

class Employee {
public:
    // ...对象上的操作
private:
    std::string ID;
    std::string name;
    char sex;
    Address addr;
    Tel_number tel;
};
class Company {

```

```

public:
    // ...对象上的操作
private:
    std::string name;
    Address addr;
    Tel_number tel;
};

```

## (d) 某大学的学生

```

class Student {
public:
    // ...对象上的操作
private:
    std::string ID;
    std::string name;
    char sex;
    std::string dept; // 所在系
    std::string major;
    Address home_addr;
    Tel_number tel;

};

```

注意，在不同的具体应用中，类的设计会有所不同，这里给出的只是一般性的简单例子。

**习题2.31**

判别下列语句哪些是声明，哪些是定义，请解释原因。

- (a) `extern int ix = 1024 ;`
- (b) `int iy ;`
- (c) `extern int iz ;`
- (d) `extern const int &ri ;`

**【解答】**

- (a) 是定义，因为`extern`声明进行了初始化。
- (b) 是定义，变量定义的常规形式。
- (c) 是声明，`extern`声明的常规形式。
- (d) 是声明，声明了一个`const`引用。

**习题2.32**

下列声明和定义哪些应该放在头文件中？哪些应该放在源文件中？请解释原因。

- (a) `int var ;`
- (b) `const double pi = 3.1416;`
- (c) `extern int total = 255 ;`
- (d) `const double sq2 = sqrt (2.0) ;`

**【解答】**

(a)、(c)、(d)应放在源文件中，因为(a)和(c)是变量定义，定义通常应放在源文件中。(d)中的`const`变量`sq2`不是用常量表达式初始化的，所以也应该放在源文件中。

(b)中的`const`变量`pi`是用常量表达式初始化的，应该放在头文件中。

参见2.9.1节。

**习题2.33**

确定你的编译器提供了哪些提高警告级别的选项。使用这些选项重新编译以前选择的程序，查看是否会报告新的问题。

**【解答】**

在笔者所用的编译器（Microsoft Visual C++ .NET 2003）中，在 Project 菜单中选择 Properties 菜单项，在 Configuration Properties→C/C++→General→Warning Level 中可以选择警告级别。



# 第3章

## 标准库类型

### 习题3.1

用适当的using声明，而不用std::前缀，访问标准库中的名字，重新编写2.3节的程序，计算一个给定数的给定次幂的结果。

#### 【解答】

```
#include <iostream>
using std::cin;
using std::cout;

int main()
{
    // 局部对象
    int base, exponent;
    long result=1;

    // 读入底数和指数
    cout << "Enter base and exponent:" << endl;
    cin >> base >> exponent;

    if (exponent < 0) {
        cout << "Exponent can't be smaller than 0" << endl;
        return -1;
    }

    if (exponent > 0) {
        // 计算底数的指数次方
        for (int cnt = 1; cnt <= exponent; ++cnt)
            result *= base;
    }

    cout << base
        << " raised to the power of "
        << exponent << ":" "
        << result << endl;
}

return 0;
}
```

### 习题3.2

什么是默认构造函数？

#### 【解答】

默认构造函数（default constructor）就是在没有显式提供初始化式时调用的构造函数。它由不带参数的构造函数，或者为所有形参提供默认实参的构造函数定义。如果定义某个类的变量时没有提供

初始化式，就会使用默认构造函数。

如果用户定义的类中没有显式定义任何构造函数，编译器就会自动为该类生成默认构造函数，称为合成的默认构造函数（synthesized default constructor）。

### 习题3.3

列举出三种初始化string对象的方法。

#### 【解答】

(1) 不带初始化式，使用默认构造函数初始化string对象。

(2) 使用一个已存在的string对象作为初始化式，将新创建的string对象初始化为已存在对象的副本。

(3) 使用字符串字面值作为初始化式，将新创建的string对象初始化为字符串字面值的副本。

### 习题3.4

s和s2的值分别是什么？

```
string s;
int main() {
    string s2;
}
```

#### 【解答】

s和s2的值均为空字符串。

### 习题3.5

编写程序实现从标准输入每次读入一行文本。然后改写程序，每次读入一个单词。

#### 【解答】

```
//从标准输入每次读入一行文本
#include <iostream>
#include <string>

using namespace std;
int main()
{
    string line;
    // 一次读入一行，直至遇见文件结束符
    while (getline(cin, line))
        cout << line << endl;    // 输出相应行以进行验证

    return 0;
}
```

修改后程序如下：

```
//从标准输入每次读入一个单词
#include <iostream>
#include <string>

using namespace std;
int main()
{
```



```

string word;
// 一次读入一个单词，直至遇见文件结束符
while (cin >> word)
    cout << word << endl; // 输出相应单词以进行验证

return 0;
}

```

注意，一般而言，应该尽量避免使用using指示而使用using声明（参见17.2.4节），因为如果应用程序中使用了多个库，使用using指示引入这些库中定义的名字空间，容易导致名字冲突。但本书中的程序都只使用了标准库，没有使用其他库。使用using指示引入名字空间std中定义的所有名字不会发生名字冲突。因此为了使得代码更为简洁以节省篇幅，本书的许多代码中都使用了using指示using namespace std;来引入名字空间std。另外，本题中并未要求输出，加入输出是为了更清楚地表示读入的结果。本书后面部分有些地方与此类似处理，不再赘述。

### 习题3.6

解释string类型的输入操作符和getline函数分别如何处理空白字符。

#### 【解答】

string类型的输入操作符对空白字符的处理：读取并忽略有效字符（非空白字符）之前所有的空白字符，然后读取字符直至再次遇到空白字符，读取终止（该空白字符仍留在输入流中）。

getline函数对空白字符的处理：不忽略行开头的空白字符，读取字符直至遇到换行符，读取终止并丢弃换行符（换行符从输入流中去掉但并不存储在string对象中）。

### 习题3.7

编一个程序读入两个string对象，测试它们是否相等。若不相等，则指出两个中哪个较大。接着，改写程序测试它们的长度是否相等，若不相等，则指出两个中哪个较长。

#### 【解答】

测试两个string对象是否相等的程序：

```

#include <iostream>
#include <string>
using namespace std;

int main()
{
    string s1, s2;

    // 读入两个string对象
    cout << "Enter two strings:" << endl;
    cin >> s1 >> s2;

    // 测试两个string对象是否相等
    if (s1 == s2)
        cout << "They are equal." << endl;
    else if (s1 > s2)
        cout << "\"" << s1 << "\"" is bigger than"
            << " \"" << s2 << "\"" << endl;
    else
        cout << "\"" << s2 << "\"" is bigger than"

```

```

        << " \\" << s1 << "\\" << endl;

    return 0;
}

```

测试两个string对象的长度是否相等的程序：

```

#include <iostream>
#include <string>
using namespace std;

int main()
{
    string s1, s2;

    // 读入两个string对象
    cout << "Enter two strings:" << endl;
    cin >> s1 >> s2;

    // 比较两个string对象的长度
    string::size_type len1, len2;
    len1 = s1.size();
    len2 = s2.size();
    if (len1 == len2)
        cout << "They have same length." << endl;
    else if (len1 > len2)
        cout << "\\" << s1 << "\" is longer than"
            << "\\" << s2 << "\\" << endl;
    else
        cout << "\\" << s2 << "\" is longer than"
            << "\\" << s1 << "\\" << endl;

    return 0;
}

```

### 习题3.8

编一个程序，从标准输入读取多个string对象，把它们连接起来存放到一个更大的string对象中，并输出连接后的string对象。接着，改写程序，将连接后相邻string对象以空格隔开。

#### 【解答】

```

#include <iostream>
#include <string>
using namespace std;

int main()
{
    string result_str, str;

    // 读入多个string对象并进行连接
    cout << "Enter strings(Ctrl+Z to end):" << endl;
    while (cin >> str)
        result_str = result_str + str;

    // 输出连接后的string对象
    cout << "String equal to the concatenation of these strings is:"
        << endl << result_str << endl;

    return 0;
}

```

}

改写后的程序：

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
    string result_str, str;

    // 读入多个string对象并进行连接
    cout << "Enter strings(Ctrl+Z to end):" << endl;
    cin >> result_str;//读入第一个string对象，放到结果对象中
    while (cin >> str)
        result_str = result_str + ' ' + str;

    // 输出连接后的string对象
    cout << "String equal to the concatenation of these strings is:"
        << endl << result_str << endl;

    return 0;
}
```

### 习题3.9

下列程序实现什么功能？实现合法吗？如果不合法，说明理由。

```
string s;
cout << s[0] << endl;
```

#### 【解答】

该程序段输出string对象s所对应字符串的第一个字符。

实现不合法。因为s是一个空字符串，其长度为0，因此s[0]是无效的。

注意，在一些编译器（如Microsoft Visual C++ .NET 2003）的实现中，该程序段并不出现编译错误。

### 习题3.10

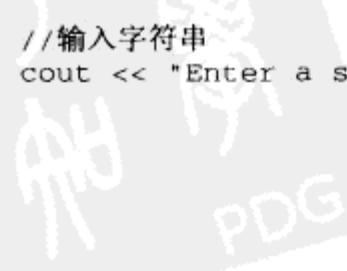
编一个程序，从string对象中去掉标点符号。要求输入到程序的字符串必须含有标点符号，输出结果则是去掉标点符号后的string对象。

#### 【解答】

```
#include <iostream>
#include <string>
#include <cctype>
using namespace std;

int main()
{
    string s, result_str;
    bool has_punct = false;//用于标记字符串中有无标点
    char ch;

    //输入字符串
    cout << "Enter a string:" << endl;
```



```

getline(cin, s);

//处理字符串：去掉其中的标点
for (string::size_type index = 0; index != s.size(); ++index)
{
    ch = s[index];
    if (ispunct(ch))
        has_punct = true;
    else
        result_str += ch;
}

if (has_punct)
    cout << "Result:" << endl << result_str << endl;
else {
    cout << "No punctuation character in the string?!" << endl;
    return -1;
}

return 0;
}

```

**习题3.11**

下面哪些vector定义不正确？

- (a) vector< vector<int> > ivec;
- (b) vector<string> svec = ivec ;
- (c) vector<string> svec(10,"null");

**【解答】**

(b) 不正确。因为svec定义为保存string对象的vector对象，而ivec是保存vector <int>对象的vector对象（即ivec是vector的vector），二者的元素类型不同，所以不能用ivec来初始化svec。

**习题3.12**

下列每个vector对象中元素个数是多少？各元素的值是什么？

- (a) vector<int> ivec1;
- (b) vector<int> ivec2(10);
- (c) vector<int> ivec3(10,42);
- (d) vector<string> svec1;
- (e) vector<string> svec2(10);
- (f) vector<string> svec3(10,"hello");

**【解答】**

- (a) 元素个数为0。
- (b) 元素个数为10，各元素的值均为0。
- (c) 元素个数为10，各元素的值均为42。
- (d) 元素个数为0。
- (e) 元素个数为10，各元素的值均为空字符串。
- (f) 元素个数为10，各元素的值均为"hello"。

**习题3.13**

读一组整数到vector对象，计算并输出每对相邻元素的和。如果读入元素个数为奇数，则提示用户最后一个元素没有求和，并输出其值。然后修改程序：头尾元素两两配对（第一个和最后一个，第二个和倒数第二个，以此类推），计算每对元素的和，并输出。

**【解答】**

```
//读一组整数到vector对象，计算并输出每对相邻元素的和
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    vector<int> ivec;
    int ival;

    // 读入数据到vector对象
    cout << "Enter numbers(Ctrl+Z to end):" << endl;
    while (cin>>ival)
        ivec.push_back(ival);

    // 计算相邻元素的和并输出
    if (ivec.size() == 0) {
        cout << "No element?!" << endl;
        return -1;
    }
    cout << "Sum of each pair of adjacent elements in the vector:" << endl;
    for (vector<int>::size_type ix = 0; ix < ivec.size()-1;
         ix = ix + 2) {
        cout << ivec[ix] + ivec[ix+1] << "\t";
        if ((ix+1) % 6 == 0) // 每行输出6个和
            cout << endl;
    }

    if (ivec.size() % 2 != 0) // 提示最后一个元素没有求和
        cout << endl
            << "The last element is not been summed"
            << "and its value is"
            << ivec[ivec.size()-1] << endl;
}

return 0;
}
```

修改后的程序：

```
//读一组整数到vector对象，计算首尾配对元素的和并输出
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    vector<int> ivec;
    int ival;

    // 读入数据到vector对象
    cout << "Enter numbers:" << endl;
    while (cin>>ival)
```

```

ivec.push_back(ival);

//计算首尾配对元素的和并输出
if (ivec.size() == 0) {
    cout << "No element?!" << endl;
    return -1;
}
cout << "Sum of each pair of counterpart elements in the vector:" 
    << endl;
vector<int>::size_type cnt = 0;
for (vector<int>::size_type first = 0, last = ivec.size() - 1;
     first < last; ++first, --last) {
    cout << ivec[first] + ivec[last] << "\t";
    ++cnt;
    if (cnt % 6 == 0) //每行输出6个和
        cout << endl;
}

if (first == last) //提示居中元素没有求和
    cout << endl
        << "The center element is not been summed "
        << "and its value is "
        << ivec[first] << endl;

return 0;
}

```

### 习题3.14

读入一段文本到vector对象，每个单词存储为vector中的一个元素。把vector对象中每个单词转化为大写字母。输出vector对象中转化后的元素，每8个单词为一行输出。

#### 【解答】

```

//读入一段文本到vector对象，每个单词存储为vector中的一个元素。
//把vector对象中每个单词转化为大写字母。
//输出vector对象中转化后的元素，每8个单词为一行输出
#include <iostream>
#include <string>
#include <vector>
#include <cctype>
using namespace std;

int main()
{
    vector<string> svec;
    string str;

    // 读入文本到vector对象
    cout << "Enter text(Ctrl+Z to end):" << endl;
    while (cin >> str)
        svec.push_back(str);

    //将vector对象中每个单词转化为大写字母，并输出
    if (svec.size() == 0) {
        cout << "No string?!" << endl;
        return -1;
    }

    cout << "Transformed elements from the vector:" 
        << endl;
    for (vector<string>::size_type ix = 0; ix != svec.size(); ++ix) {

```

```

        for (string::size_type index = 0; index != svec[ix].size(); ++index)
            if (islower(svec[ix][index]))
                //单词中下标为index的字符为小写字母
                svec[ix][index] = toupper(svec[ix][index]);
        cout << svec[ix] << " ";
        if ((ix + 1) % 8 == 0)//每8个单词为一行输出
            cout << endl;
    }
    return 0;
}

```

**习题3.15**

下面程序合法吗？如果不合法，如何更正？

```
vector<int> ivec;
ivec[0] = 42;
```

**【解答】**

不合法。因为ivec是空的vector对象，其中不含任何元素，而下标操作只能用于获取已存在的元素。

更正：将赋值语句改为语句ivec.push\_back(42);。

**习题3.16**

列出三种定义vector对象的方法，给定10个元素，每个元素值为42。指出是否还有更好的实现方法，并说明为什么。

**【解答】**

方法一：

```
vector<int> ivec(10, 42);
```

方法二：

```
vector<int> ivec(10);
for (ix = 0; ix < 10; ++ix)
    ivec[ix] = 42;
```

方法三：

```
vector<int> ivec(10);
for (vector<int>::iterator iter = ivec.begin();
                 iter != ivec.end(); ++iter)
    *iter = 42;
```

方法四：

```
vector<int> ivec;
for (cnt = 1; cnt <= 10; ++cnt)
    ivec.push_back(42);
```

方法五：

```
vector<int> ivec;
vector<int>::iterator iter = ivec.end();
for (int i = 0; i != 10; ++i) {
    ivec.insert(iter, 42);
    iter = ivec.end();
```

}

各种方法都可达到目的，也许最后两种方法更好一些。它们使用标准库中定义的容器操作在容器中增添元素，无需在定义vector对象时指定容器的大小，比较灵活而且不容易出错。

### 习题3.17

重做3.3.2节的习题，用迭代器而不是下标操作来访问vector中的元素。

#### 【解答】

重做习题3.13如下：

```
//读一组整数到vector对象，计算并输出每对相邻元素的和
//使用迭代器访问vector中的元素
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    vector<int> ivec;
    int ival;

    //读入数据到vector对象
    cout << "Enter numbers(Ctrl+Z to end):" << endl;
    while (cin>>ival)
        ivec.push_back(ival);

    //计算相邻元素的和并输出
    if (ivec.size() == 0) {
        cout << "No element?!" << endl;
        return -1;
    }
    cout << "Sum of each pair of adjacent elements in the vector:" 
         << endl;
    vector<int>::size_type cnt = 0;
    for (vector<int>::iterator iter = ivec.begin();
                     iter < ivec.end()-1;
                     iter = iter + 2) {
        cout << *iter + *(iter+1) << "\t";
        ++cnt;
        if (cnt % 6 == 0) //每行输出6个和
            cout << endl;
    }

    if (ivec.size() % 2 != 0) //提示最后一个元素没有求和
        cout << endl
            << "The last element is not been summed "
            << "and its value is "
            << *(ivec.end()-1) << endl;
}

//读一组整数到vector对象，计算首尾配对元素的和并输出
//使用迭代器访问vector中的元素
#include <iostream>
#include <vector>
using namespace std;

int main()
```

```

{
    vector<int> ivec;
    int ival;

    //读入数据到vector对象
    cout << "Enter numbers(Ctrl+Z to end):" << endl;
    while (cin>>ival)
        ivec.push_back(ival);

    //计算首尾配对元素的和并输出
    if (ivec.size() == 0) {
        cout << "No element?!" << endl;
        return -1;
    }
    cout << "Sum of each pair of counterpart elements in the vector:"
        << endl;
    vector<int>::size_type cnt=0;
    for (vector<int>::iterator first = ivec.begin(),
                           last = ivec.end() - 1;
                           first < last;
                           ++first, --last) {
        cout << *first + *last << "\t";
        ++cnt;
        if (cnt % 6 == 0) //每行输出6个和
            cout << endl;
    }

    if (first == last) //提示居中元素没有求和
        cout << endl
            << "The center element is not been summed "
            << "and its value is "
            << *first << endl;
}

return 0;
}

```

重做习题3.14如下：

```

//读入一段文本到vector对象，每个单词存储为vector中的一个元素。
//把vector对象中每个单词转化为大写字母。
//输出vector对象中转化后的元素，每8个单词为一行输出。
//使用迭代器访问vector中的元素
#include <iostream>
#include <string>
#include <vector>
#include <cctype>
using namespace std;

int main()
{
    vector<string> svec;
    string str;

    //读入文本到vector对象
    cout << "Enter text(Ctrl+Z to end):" << endl;
    while (cin>>str)
        svec.push_back(str);

    //将vector对象中每个单词转化为大写字母，并输出
    if (svec.size() == 0) {
        cout << "No string?!" << endl;
        return -1;
    }
}

```

```

cout << "Transformed elements from the vector:"
     << endl;
vector<string>::size_type cnt = 0;
for (vector<string>::iterator iter = svec.begin();
     iter != svec.end(); ++iter) {
    for (string::size_type index = 0; index != (*iter).size();
         ++index)
        if (islower((*iter)[index]))
            //单词中下标为index的字符为小写字母
            (*iter)[index] = toupper((*iter)[index]);
    cout << *iter << " ";
    ++cnt;
    if (cnt % 8 == 0)//每8个单词为一行输出
        cout << endl;
}
return 0;
}

```

**习题3.18**

编写程序来创建有10个元素的vector对象。用迭代器把每个元素值改为当前值的2倍。

**【解答】**

```

//创建有10个元素的vector对象,
//然后使用迭代器将每个元素值改为当前值的2倍
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    vector<int> ivec(10, 20); //每个元素的值均为20

    //将每个元素值改为当前值的2倍
    for (vector<int>::iterator iter = ivec.begin();
         iter != ivec.end(); ++iter)
        *iter = (*iter)*2;

    return 0;
}

```

**习题3.19**

验证习题3.18的程序，输出vector的所有元素。

**【解答】**

```

//创建有10个元素的vector对象,
//然后使用迭代器将每个元素值改为当前值的2倍并输出
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    vector<int> ivec(10, 20); //每个元素的值均为20

    //将每个元素值改为当前值的2倍并输出
    for (vector<int>::iterator iter = ivec.begin();
         iter != ivec.end(); ++iter) {
        *iter = (*iter)*2;
        cout << *iter << " ";
    }
}

```

```

    }
    return 0;
}

```

**习题3.20**

解释一下在上几个习题的程序实现中你用了哪种迭代器，并说明原因。

**【解答】**

上述几个习题的程序实现中使用了类型分别为`vector<int>::iterator`和`vector<string>::iterator`的迭代器，通过这些迭代器分别访问元素类型为`int`和`string`的`vector`对象中的元素。

**习题3.21**

何时使用`const`迭代器？又在何时使用`const_iterator`？解释两者的区别。

**【解答】**

`const`迭代器是迭代器常量，该迭代器本身的值不能修改，即该迭代器在定义时需要初始化，而且初始化之后，不能再指向其他元素。若需要指向固定元素的迭代器，则可以使用`const`迭代器。

`const_iterator`是一种迭代器类型，对这种类型的迭代器解引用会得到一个指向`const`对象的引用，即通过这种迭代器访问到的对象是常量。该对象不能修改，因此，`const_iterator`类型只能用于读取容器内的元素，不能修改元素的值。若只需遍历容器中的元素而无需修改它们，则可以使用`const_iterator`。

**习题3.22**

如果采用下面的方法来计算`mid`会产生什么结果？

```
vector<int>::iterator mid = (vi.begin() + vi.end()) / 2;
```

**【解答】**

将两个迭代器相加的操作是未定义的，因此用这种方法计算`mid`会出现编译错误。

**习题3.23**

解释下面每个`bitset`对象包含的位模式：

- (a) `bitset<64> bitvec(32);`
- (b) `bitset<32> bv(1010101);`
- (c) `string bstr; cin >> bstr; bitset<8> bv(bstr);`

**【解答】**

(a) `bitvec`有64个二进制位，（位编号从0开始）第5位置为1，其余位置均为0。

(b) `bv`有32个二进制位，（位编号从0开始）第0、2、4、5、7、8、11、13、14、16、17、18、19位置为1，其余位置均为0。因为十进制数1010101对应的二进制数为00000000000011110110100110110101。

(c) `bv`有8个二进制位，（位编号从0开始）用读入的字符串的从右至左的8个字符对`bv`的0~7位进行初始化。

**习题3.24**

考虑这样的序列1,2,3,5,8,13,21，并初始化一个将该序列数字所对应的位置设置为1的bitset<32>对象。然后换个方法，给定一个空的bitset对象，编写一小段程序把相应的数位设置为1。

**【解答】**

bitset<32>对象的初始化：

```
bitset<32> bv(0x20212e)
```

方法二：

```
bitset<32> bv;
int x = 0, y = 1, z;
z = x + y;
while (z <= 21) {
    bv.set(z);
    x = y;
    y = z;
    z = x + y;
}
```

注意，设置为1的数位的位编号符合斐波那契数列的规律。



## 第 4 章

# 数组和指针

### 习题4.1

假设get\_size是一个没有参数并返回int值的函数，下列哪些定义是非法的？为什么？

```
unsigned buf_size = 1024
```

- (a) int ia[buf\_size];
- (b) int ia[get\_size()];
- (c) int ia[4\*7-14];
- (d) char st[11] = "fundamental" ;

### 【解答】

- (a)非法，buf\_size是一个变量，不能用于定义数组的维数（维长度）。
- (b)非法，get\_size()是函数调用，不是常量表达式，不能用于定义数组的维数（维长度）。
- (d)非法，存放字符串"fundamental"的数组必须有12个元素，st只有11个元素。

### 习题4.2

下列数组的值是什么？

```
string sa[10];
int ia[10];
int main(){
    string    sa2[10];
    int       ia2[10];
}
```

### 【解答】

sa和sa2为元素类型为string的数组，自动调用string类的默认构造函数将各元素初始化为空字符串；ia为在函数体外定义的内置数组，各元素初始化为0；ia2为在函数体内定义的内置数组，各元素未初始化，其值不确定。

### 习题4.3

下列哪些定义是错误的？

- (a) int ia[7] = {0, 1, 1, 2, 3, 5, 8};
- (b) vector<int> ivec = {0, 1, 1, 2, 3, 5, 8};
- (c) int ia2[] = ia;
- (d) int ia3[] = ivec;



**【解答】**

- (b) 错误。vector对象不能用这种方式进行初始化。
- (c) 错误。不能用一个数组来初始化另一个数组。
- (d) 错误。不能用vector对象来初始化数组。

**习题4.4**

如何初始化数组的一部分或全部元素？

**【解答】**

定义数组时可使用初始化列表（用花括号括住的一组以逗号分隔的元素初值）来初始化数组的部分或全部元素。如果是初始化全部元素，可以省略定义数组时方括号中给出的数组维数值。如果指定了数组维数，则初始化列表提供的元素个数不能超过维数值。如果数组维数大于列出的元素初值个数，则只初始化前面的数组元素，剩下的其他元素，若是内置类型则初始化为0，若是类类型则调用该类的默认构造函数进行初始化。字符数组既可以用一组由花括号括起来、逗号隔开的字符字面值进行初始化，也可以用一个字符串字面值进行初始化。

**习题4.5**

列出使用数组而不是vector的缺点。

**【解答】**

与vector类型相比，数组具有如下缺点：数组的长度是固定的，而且数组不提供获取其容量大小的size操作，也不提供自动添加元素的push\_back操作。因此，程序员无法在程序运行时知道一个给定数组的长度，而且如果需要更改数组的长度，程序员只能创建一个更大的新数组，然后把原数组的所有元素复制到新数组的存储空间中去。与使用vector类型的程序相比，使用内置数组的程序更容易出错且难以调试。

**习题4.6**

下面的程序段企图将下标值赋给数组的每个元素，其中在下标操作上有一些错误，请指出这些错误。

```
const size_t array_size = 10 ;
int ia[array_size];
for (size_t ix = 1; ix <= array_size; ++ix)
    ia[ix] = ix ;
```

**【解答】**

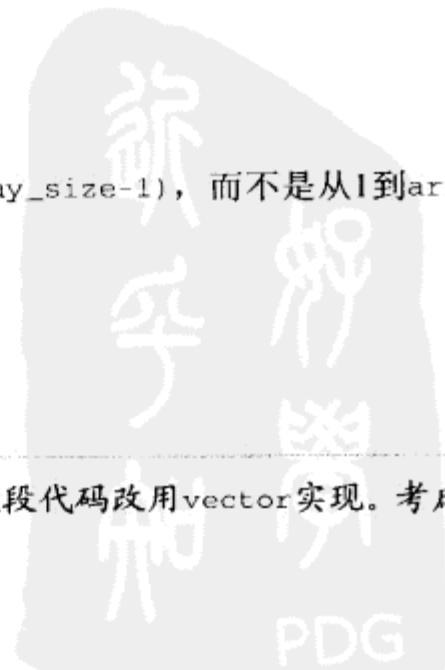
该程序段的错误是：数组下标使用越界。

根据数组ia的定义，该数组的下标值应该是0~9（即array\_size-1），而不是从1到array\_size，因此其中的for语句出错，可更正如下：

```
for (size_t ix = 0; ix < array_size; ++ix)
    ia[ix] = ix ;
```

**习题4.7**

编写必要的代码将一个数组赋给另一个数组，然后把这段代码改用vector实现。考虑如何将一个



vector赋给另一个vector。

### 【解答】

将一个数组赋给另一个数组，就是将一个数组的元素逐个赋值给另一数组的对应元素，可用如下代码实现：

```
int main()
{
    const size_t array_size = 10;
    int ia1[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    int ia2[array_size];

    for (size_t ix = 0; ix != array_size; ++ix)
        ia2[ix] = ia1[ix];

    return 0;
}
```

将一个vector赋给另一个vector，也是将一个vector的元素逐个赋值给另一vector的对应元素，可用如下代码实现：

```
//将一个vector赋值给另一个vector
//使用迭代器访问vector中的元素
#include <vector>
using namespace std;

int main()
{
    vector<int> ivec1(10, 20); //每个元素初始化为20
    vector<int> ivec2;

    for (vector<int>::iterator iter = ivec1.begin();
         iter != ivec1.end(); ++iter)
        ivec2.push_back(*iter);

    return 0;
}
```

### 习题4.8

编写程序判断两个数组是否相等，然后编写一段类似的程序比较两个vector。

### 【解答】

判断两个数组是否相等，可用如下程序：

```
//判断两个数组是否相等
#include <iostream>
using namespace std;

int main()
{
    const int arr_size = 6;
    int ia1[arr_size], ia2[arr_size];
    size_t ix;

    //读入两个数组的元素值
    cout << "Enter " << arr_size
        << " numbers for array1:" << endl;
```



```

for (ix = 0; ix != arr_size; ++ix)
    cin >> ia1[ix];
cout << "Enter " << arr_size
    << " numbers for array2:" << endl;
for (ix = 0; ix != arr_size; ++ix)
    cin >> ia2[ix];

//判断两个数组是否相等
for (ix = 0; ix != arr_size; ++ix)
    if (ia1[ix] != ia2[ix]) {
        cout << "Array1 is not equal to array2." << endl;
        return 0;
    }
cout << "Array1 is equal to array2." << endl;

return 0;
}

```

判断两个vector是否相等，可用如下程序：

```

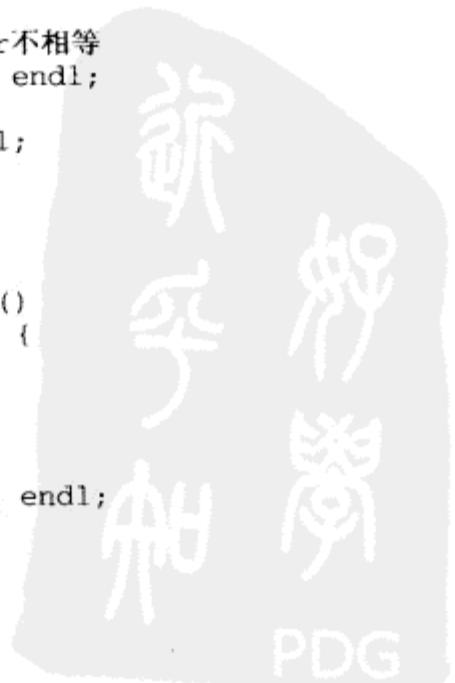
//判断两个vector是否相等
//使用迭代器访问vector中的元素
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    vector<int> ivec1, ivec2;
    int ival;

    //读入两个vector的元素值
    cout << "Enter numbers for vector1(-1 to end):" << endl;
    cin >> ival;
    while (ival != -1) {
        ivec1.push_back(ival);
        cin >> ival;
    }
    cout << "Enter numbers for vector2(-1 to end):" << endl;
    cin >> ival;
    while (ival != -1) {
        ivec2.push_back(ival);
        cin >> ival;
    }

    //判断两个vector是否相等
    if (ivec1.size() != ivec2.size()) //长度不等的vector不相等
        cout << "Vector1 is not equal to vector2." << endl;
    else if (ivec1.size() == 0) //长度都为0的vector相等
        cout << "Vector1 is equal to vector2." << endl;
    else { //两个vector长度相等且不为0
        vector<int>::iterator iter1, iter2;
        iter1 = ivec1.begin();
        iter2 = ivec2.begin();
        while (*iter1 == *iter2 && iter1 != ivec1.end()
                && iter2 != ivec2.end()) {
            ++iter1;
            ++iter2;
        }
        if (iter1 == ivec1.end()) //所有元素都相等
            cout << "Vector1 is equal to vector2." << endl;
        else
    }
}

```



```

        cout << "Vector1 is not equal to vector2." << endl;
    }

    return 0;
}

```

**习题4.9**

编写程序定义一个有10个int型元素的数组，并以元素在数组中的位置作为各元素的初值。

**【解答】**

```

// 定义一个有10个int型元素的数组,
// 并以元素在数组中的位置(1~10)作为各元素的初值
int main()
{
    const int array_size = 10;
    int ia[array_size];

    for (size_t ix = 0; ix != array_size; ++ix)
        ia[ix] = ix+1;

    return 0;
}

```

**习题4.10**

下面提供了两种指针声明的形式，解释宁愿使用第一种形式的原因：

```

int *ip; // good practice
int* ip; // legal but misleading

```

**【解答】**

第一种形式强调了ip是一个指针，这种形式在阅读时不易引起误解，尤其是当一个语句中同时定义了多个变量时。

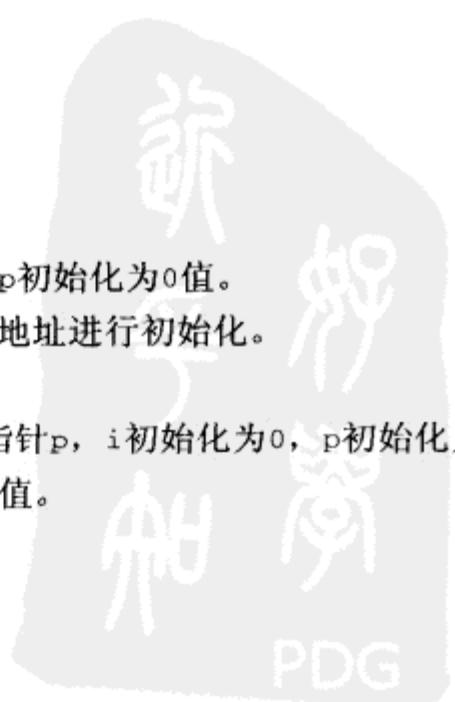
**习题4.11**

解释下列声明语句，并指出哪些是非法的，为什么？

- (a) int\* ip;
- (b) string s, \*sp = 0;
- (c) int i; double\* dp = &i;
- (d) int\* ip, ip2;
- (e) const int i = 0, \*p = i;
- (f) string \*p = NULL;

**【解答】**

- (a) 合法。定义了一个指向int型对象的指针ip。
- (b) 合法。定义了string对象s和指向string型对象的指针sp，sp初始化为0值。
- (c) 非法。dp为指向double型对象的指针，不能用int型对象i的地址进行初始化。
- (d) 合法。定义了int对象ip2和指向int型对象的指针ip。
- (e) 合法。定义了const int型对象i和指向const int型对象的指针p，i初始化为0，p初始化为0。
- (f) 合法。定义了指向string型对象的指针p，并将其初始化为0值。



**习题4.12**

已知一指针p，你可以确定该指针是否指向一个有效的对象吗？如果可以，如何确定？如果不可以，请说明原因。

**【解答】**

无法确定某指针是否指向一个有效对象。因为，在C++语言中，无法检测指针是否未被初始化，也无法区分一个地址是有效地址，还是由指针所分配的存储空间中存放的不确定值的二进制位形成的地址。

**习题4.13**

下列代码中，为什么第一个指针的初始化是合法的，而第二个则不合法？

```
int i = 42;
void *p = &i;
long *lp = &i;
```

**【解答】**

具有void\*类型的指针可以保存任意类型对象的地址，因此p的初始化是合法的；而指向long型对象的指针不能用int型对象的地址来初始化，因此lp的初始化不合法。

**习题4.14**

编写代码修改指针的值；然后再编写代码修改指针所指对象的值。

**【解答】**

下列代码修改指针的值：

```
int *ip;
int ival1, ival2;
ip = &ival1;
ip = &ival2;
```

下列代码修改指针所指对象的值：

```
int ival = 0;
int *ip = &ival;
*ip = 8;
```

**习题4.15**

解释指针和引用的主要区别。

**【解答】**

使用引用(reference)和指针(pointer)都可间接访问另一个值，但它们之间存在两个重要区别：(1)引用总是指向某个确定对象(事实上，引用就是该对象的别名)，定义引用时没有进行初始化会出现编译错误；(2)赋值行为上存在差异：给引用赋值修改的是该引用所关联的对象的值，而不是使该引用与另一个对象关联。引用一经初始化，就始终指向同一个特定对象。给指针赋值修改的是指针对象本身，也就是使该指针指向另一对象，指针在不同时刻可指向不同的对象(只要保证类型匹配)。

**习题4.16**

下列程序段实现什么功能？

```
int i = 42, j = 1024;
int *p1 = &i, *p2 = &j;
*p2 = *p1 * *p2;
*p1 *= *p1;
```

**【解答】**

该程序段使得i被赋值为42的平方，j被赋值为42与1024的乘积。

**习题4.17**

已知p1和p2指向同一个数组中的元素，下面语句实现什么功能？

```
p1 += p2 - p1;
```

当p1和p2具有什么值时这个语句是非法的？

**【解答】**

此语句使得p1也指向p2原来所指向的元素。原则上说，只要p1和p2的类型相同，则该语句始终是合法的。只有当p1和p2不是同类型指针时，该语句才不合法（不能进行-操作）。

但是，如果p1和p2不是指向同一个数组中的元素，则这个语句的执行结果可能是错误的。因为-操作的结果类型ptrdiff\_t只能保证足以存放同一数组中两个指针之间的差距。如果p1和p2不是指向同一个数组中的元素，则-操作的结果有可能超出ptrdiff\_t类型的表示范围而产生溢出，从而该语句的执行结果不能保证p1指向p2原来所指向的元素（甚至不能保证p1为有效指针）。

**习题4.18**

编写程序，使用指针把一个int型数组的所有元素设置为0。

**【解答】**

```
// 使用指针把一个int型数组的所有元素设置为0

int main()
{
    const size_t arr_size = 8;
    int int_arr[arr_size] = { 0, 1, 2, 3, 4, 5, 6, 7 };

    // pbegin指向第一个元素，pend指向最后一个元素的下一内存位置
    for (int *pbegin = int_arr, *pend = int_arr + arr_size;
         pbegin != pend; ++pbegin)
        *pbegin = 0; // 当前元素置0

    return 0;
}
```

**习题4.19**

解释下列5个定义的含义，指出其中哪些定义是非法的：

- (a) int i;
- (b) const int ic;
- (c) const int \*pic;

- (d) int \*const cpi;  
 (e) const int \*const cpic;

**【解答】**

- (a) 合法：定义了int型对象i。  
 (b) 非法：定义const对象时必须进行初始化，但ic没有初始化。  
 (c) 合法：定义了指向int型const对象的指针pic。  
 (d) 非法：因为cpi被定义为指向int型对象的const指针，但该指针没有初始化。  
 (e) 非法：因为cpic被定义为指向int型const对象的const指针，但该指针没有初始化。

**习题4.20**

下列哪些初始化是合法的？为什么？

- (a) int i = -1;  
 (b) const int ic = i ;  
 (c) const int \*pic = &ic;  
 (d) int \*const cpi = &ic;  
 (e) const int \*const cpic = &ic;

**【解答】**

- (a) 合法：定义了一个int型对象i，并用int型字面值-1对其进行初始化。  
 (b) 合法：定义了一个int型const对象ic，并用int型对象对其进行初始化。  
 (c) 合法：定义了一个指向int型const对象的指针pic，并用ic的地址对其进行初始化。  
 (d) 不合法：cpi是一个指向int型对象的const指针，不能用const int型对象ic的地址对其进行初始化。  
 (e) 合法：定义了一个指向int型const对象的const指针cpic，并用ic的地址对其进行初始化。

**习题4.21**

根据上述定义，下列哪些赋值运算是合法的？为什么？

- |                 |                 |
|-----------------|-----------------|
| (a) i = ic;     | (b) pic = &ic;  |
| (c) cpi = pic;  | (d) pic = cpic; |
| (e) cpic = &ic; | (f) ic = *cpic; |

**【解答】**

- (a)、(b)、(d)合法。  
 (c)、(e)、(f)均不合法，因为cpi、cpic和ic都是const变量（常量），常量不能被赋值。

**习题4.22**

解释下列两个while循环的差别：

```
const char *cp = "hello";
int cnt;
while (cp) { ++cnt; ++cp; }
while (*cp) { ++cnt; ++cp; }
```

**【解答】**

两个while循环的差别为：前者的循环结束条件是cp为0值（即指针cp为0值）；后者的循环结束条件是cp所指向的字符为0值（即cp所指向的字符为字符串结束符null（即'\0'））。因此后者能正确地计算出字符串"hello"中有效字符的数目（放在cnt中），而前者的执行是不确定的。

注意，题目中的代码还有一个小问题，即cnt没有初始化为0值。

**习题4.23**

下列程序实现什么功能？

```
const char ca[] = {'h', 'e', 'l', 'l', 'o'};
const char *cp = ca;
while (*cp) {
    cout << *cp << endl;
    ++cp;
}
```

**【解答】**

该程序段从数组ca的起始地址（即字符'h'的存储地址）开始，输出一段内存中存放的字符，每行输出一个字符，直至存放0值(null)的字节为止。（注意，输出的内容一般来说要多于5个字符，因为字符数组ca中没有null结束符。）

**习题4.24**

解释strcpy和strncpy的差别在哪里，各自的优缺点是什么？

**【解答】**

strcpy和strncpy的差别在于：前者复制整个指定的字符串，后者只复制指定字符串中指定数目的字符。

strcpy比较简单，而使用strncpy可以适当地控制复制字符的数目，因此比strcpy更为安全。

**习题4.25**

编写程序比较两个string类型的字符串，然后编写另一个程序比较两个C风格字符串的值。

**【解答】**

比较两个string类型的字符串的程序如下：

```
//比较两个string类型的字符串
#include <iostream>
#include <string>
using namespace std;

int main()
{
    string str1, str2;

    //输入两个字符串
    cout << "Enter two strings:" << endl;
    cin >> str1 >> str2;

    //比较两个字符串
    if (str1 > str2)
```

```

        cout << "\\" << str1 << "\\" << " is bigger than "
        << "\\" << str2 << "\\" << endl;
    else if (str1 < str2)
        cout << "\\" << str2 << "\\" << " is bigger than "
        << "\\" << str1 << "\\" << endl;
    else
        cout << "They are equal" << endl;

    return 0;
}

```

比较两个C风格字符串的程序如下：

```

//比较两个C风格字符串的值
#include <iostream>
#include <cstring>
using namespace std;

int main()
{
    //char *str1 = "string1", *str2 = "string2";
    const int str_size = 80;
    char *str1, *str2;

    //为两个字符串分配内存
    str1 = new char[str_size];
    str2 = new char[str_size];
    if (str1 == NULL || str2 == NULL) {
        cout << "No enough memory!" << endl;
        return -1;
    }

    //输入两个字符串
    cout << "Enter two strings:" << endl;
    cin >> str1 >> str2;

    //比较两个字符串
    int result;
    result = strcmp(str1, str2);
    if (result > 0)
        cout << "\\" << str1 << "\\" << " is bigger than "
        << "\\" << str2 << "\\" << endl;
    else if (result < 0)
        cout << "\\" << str2 << "\\" << " is bigger than "
        << "\\" << str1 << "\\" << endl;
    else
        cout << "They are equal" << endl;

    //释放字符串所占用的内存
    delete [] str1;
    delete [] str2;

    return 0;
}

```

注意，此程序中使用了内存的动态分配与释放（见4.3.1节）。如果不用内存的动态分配与释放，可将主函数中第2、3两行代码、有关内存分配与释放的代码以及输入字符串的代码注释掉，再将主函数中第一行代码

```
//char *str1 = "string1", *str2 = "string2";
```

前的双斜线去掉即可。

### 习题4.26

编写程序从标准输入设备读入一个string类型的字符串。考虑如何编程实现从标准输入设备读入一个C风格字符串。

#### 【解答】

从标准输入设备读入一个string类型字符串的程序段：

```
string str;
cin >> str;
```

从标准输入设备读入一个C风格字符串可如下实现：

```
const int str_size = 80;
char str[str_size];
cin >> str;
```

### 习题4.27

假设有下面的new表达式，请问如何释放pa？

```
int *pa = new int[10];
```

#### 【解答】

用语句`delete [] pa;`释放pa所指向的数组空间。

### 习题4.28

编写程序由从标准输入设备读入的元素数据建立一个int型vector对象，然后动态创建一个与该vector对象大小一致的数组，把vector对象的所有元素复制给新数组。

#### 【解答】

```
// 从标准输入设备读入的元素数据建立一个int型vector对象,
// 然后动态创建一个与该vector对象大小一致的数组,
// 把vector对象的所有元素复制给新数组
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    vector<int> ivec;
    int ival;

    // 读入元素数据并建立vector
    cout << "Enter numbers:(Ctrl+Z to end)" << endl;
    while (cin >> ival)
        ivec.push_back(ival);

    // 动态创建数组
    int *pia = new int[ivec.size()];

    // 复制元素
    int *tp = pia;
    for (vector<int>::iterator iter = ivec.begin();
```

```

        iter != ivec.end(); ++iter, ++tp)
    *tp = *iter;

    //释放动态数组的内存
    delete [] pia;

    return 0;
}

```

**习题4.29**

对本节第5条框中的两段程序：

- (a) 解释这两段程序实现的功能。
- (b) 平均来说，使用string类型的程序执行速度要比用C风格字符串的快很多，在我们用了5年的PC机上其平均执行速度分别是：

```

user 0.47 # string class
user 2.55 # C-style character string

```

你预计的也一样吗？请说明原因。

**【解答】**

- (a) 这两段程序的功能是：执行一个循环次数为1000000的循环，在该循环的循环体中：创建一个新字符串，将一个已存在的字符串复制给新字符串，然后比较两个字符串，最后释放新字符串。
- (b) 使用C风格字符串的程序需要自己管理内存的分配和释放，而使用string类型的程序由系统自动进行内存的分配和释放，因此比使用C风格字符串的程序要简短，执行速度也要快一些。

**习题4.30**

编写程序连接两个C风格字符串字面值，把结果存储在一个C风格字符串中。然后再编写程序连接两个string类型字符串，这两个string类型字符串与前面的C风格字符串字面值具有相同的内容。

**【解答】**

连接两个C风格字符串字面值的程序如下：

```

// 连接两个C风格字符串字面值,
// 把结果存储在一个C风格字符串中
#include <cstring>

int main()
{
    const char *cp1 = "Mary and Linda ";
    const char *cp2 = "are friends.";

    size_t len = strlen(cp1) + strlen(cp2);
    char *result_str = new char[len+1];
    strcpy(result_str, cp1);
    strcat(result_str, cp2);
    delete [] result_str;

    return 0;
}

```

相应的连接两个string类型字符串的程序如下：

```
// 连接两个string类型字符串
#include <string>
using namespace std;

int main()
{
    const string str1("Mary and Linda ");
    const string str2("are firends.");
    string result_str;
    result_str = str1;
    result_str += str2;

    return 0;
}
```

**习题4.31**

编写程序从标准输入设备读入字符串，并把该串存放在字符数组中。描述你的程序如何处理可变长的输入。提供比你分配的数组长度长的字符串数据测试你的程序。

**【解答】**

```
// 从标准输入设备读入字符串，并把该串存放在字符数组中
#include <iostream>
#include <string>
#include <cstring>
using namespace std;

int main()
{
    string in_str;// 用于读入字符串的string对象
    const size_t str_size = 10;
    char result_str[str_size+1];

    // 读入字符串
    cout << "Enter a string(<=" << str_size
        << " characters):" << endl;
    cin >> in_str;

    // 计算需复制的字符的数目
    size_t len = strlen(in_str.c_str());
    if (len > str_size) {
        len = str_size;
        cout << "String is longer than " << str_size
            << " characters and is stored only "
            << str_size << " characters!" << endl;
    }

    // 复制len个字符至字符数组result_str
    strncpy(result_str, in_str.c_str(), len);

    // 在末尾加上一个空字符(null字符)
    result_str[len+1] = '\0';

    return 0;
}
```

为了接受可变长的输入，程序中用一个string对象存放读入的字符串，然后使用strncpy函数将该对象的适当内容复制到字符数组中。因为字符数组的长度是固定的，因此首先计算字符串的长度。若该长度小于或等于字符数组可容纳字符串的长度，则复制整个字符串至字符数组，否则，根据数组

的长度，复制字符串中前面部分的字符，以防止溢出。

注意，上述给出的是满足题目要求的一个解答，事实上，如果希望接受可变长的输入并完整地存放到字符数组中，可以采用动态创建数组来实现。

### 习题4.32

编写程序用int型数组初始化vector对象。

#### 【解答】

```
// 用int型数组初始化vector对象
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    const size_t arr_size = 8;
    int int_arr[arr_size];

    // 输入数组元素
    cout << "Enter " << arr_size << " numbers:" << endl;
    for (size_t ix = 0; ix != arr_size; ++ix)
        cin >> int_arr[ix];

    // 用int型数组初始化vector对象
    vector<int> ivec(int_arr, int_arr + arr_size);

    return 0;
}
```

### 习题4.33

编写程序把int型vector复制给int型数组。

#### 【解答】

```
// 把int型vector复制给int型数组
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    vector<int> ivec;
    int ival;

    // 输入vector元素
    cout << "Enter numbers: (Ctrl+Z to end)" << endl;
    while (cin >> ival)
        ivec.push_back(ival);

    // 创建数组
    int *parr = new int[ivec.size()];

    // 复制元素
    size_t ix = 0;
    for (vector<int>::iterator iter = ivec.begin();
         iter != ivec.end(); ++iter, ++ix)
        parr[ix] = *iter;
}
```

```

    // 释放数组
    delete [] parr;

    return 0;
}

```

**习题4.34**

编写程序读入一组string类型的数据，并将它们存储在vector中。接着，把该vector对象复制给一个字符指针数组。为vector中的每个元素创建一个新的字符数组，并把该vector元素的数据复制到相应的字符数组中，最后把指向该数组的指针插入字符指针数组。

**【解答】**

```

//4-34.cpp
//读入一组string类型的数据，并将它们存储在vector中。
//接着，把该vector对象复制给一个字符指针数组。
//为vector中的每个元素创建一个新的字符数组，
//并把该vector元素的数据复制到相应的字符数组中，
//最后把指向该数组的指针插入字符指针数组
#include <iostream>
#include <vector>
#include <string>
using namespace std;

int main()
{
    vector<string> svec;
    string str;

    // 输入vector元素
    cout << "Enter strings:(Ctrl+Z to end)" << endl;
    while (cin >> str)
        svec.push_back(str);

    // 创建字符指针数组
    char **parr = new char*[svec.size()];

    // 处理vector元素
    size_t ix = 0;
    for (vector<string>::iterator iter = svec.begin();
         iter != svec.end(); ++iter, ++ix) {
        // 创建字符数组
        char *p = new char[(*iter).size()+1];
        // 复制vector元素的数据到字符数组
        strcpy(p, (*iter).c_str());
        // 将指向该字符数组的指针插入到字符指针数组
        parr[ix] = p;
    }

    // 释放各个字符数组
    for (ix = 0; ix != svec.size(); ++ix)
        delete [] parr[ix];
    // 释放字符指针数组
    delete [] parr;

    return 0;
}

```

**习题4.35**

输出习题4.34中建立的vector对象和数组的内容。输出数组后，记得释放字符数组。

**【解答】**

```
//4-35.cpp
//读入一组string类型的数据，并将它们存储在vector中。
//接着，把该vector对象复制给一个字符指针数组：
//为vector中的每个元素创建一个新的字符数组，
//并把该vector元素的数据复制到相应的字符数组中，
//然后把指向该数组的指针插入字符指针数组。
//输出建立的vector对象和数组的内容
#include <iostream>
#include <vector>
#include <string>
using namespace std;

int main()
{
    vector<string> svec;
    string str;

    // 输入vector元素
    cout << "Enter strings:(Ctrl+Z to end)" << endl;
    while (cin >> str)
        svec.push_back(str);

    // 创建字符指针数组
    char **parr = new char*[svec.size()];

    // 处理vector元素
    size_t ix = 0;
    for (vector<string>::iterator iter = svec.begin();
         iter != svec.end(); ++iter, ++ix) {
        // 创建字符数组
        char *p = new char[(*iter).size()+1];
        // 复制vector元素的数据到字符数组
        strcpy(p, (*iter).c_str());
        // 将指向该字符数组的指针插入到字符指针数组
        parr[ix] = p;
    }

    // 输出vector对象的内容
    cout << "Content of vector:" << endl;
    for (vector<string>::iterator iter2 = svec.begin();
         iter2 != svec.end(); ++iter2)
        cout << *iter2 << endl;

    // 输出字符数组的内容
    cout << "Content of character arrays:" << endl;
    for (ix = 0; ix != svec.size(); ++ix)
        cout << parr[ix] << endl;

    // 释放各个字符数组
    for (ix = 0; ix != svec.size(); ++ix)
        delete [] parr[ix];

    // 释放字符指针数组
    delete [] parr;

    return 0;
}
```

**习题4.36**

重写程序输出ia数组的内容，要求在外层循环中不能使用typedef定义的类型。

**【解答】**

```
//4-36.cpp
//重写程序输出ia数组的内容
//在外层循环中不使用typedef定义的类型
#include <iostream>
using namespace std;

int main()
{
    int ia[3][4] = {           // 3个元素，每个元素是一个有4个int元素的数组
        {0, 1, 2, 3},         // 0行的初始化列表
        {4, 5, 6, 7},         // 1行的初始化列表
        {8, 9, 10, 11}        // 2行的初始化列表
    };

    int (*p)[4];
    for (p = ia; p != ia + 3; ++p)
        for (int *q = *p; q != *p + 4; ++q)
            cout << *q << endl;

    return 0;
}
```



## 表达式

### 习题5.1

在下列表达式中，加入适当的圆括号以标明其计算顺序。编译该表达式并输出其值，从而检查你的回答是否正确。

12 / 3 \* 4 + 5 \* 15 + 24 % 4 / 2

### 【解答】

加入如下所示的圆括号以标明该表达式的计算顺序：

((12 / 3) \* 4) + (5 \* 15)) + ((24 % 4) / 2)

### 习题5.2

计算下列表达式的值，并指出哪些结果值依赖于机器？

-30 \* 3 + 21 / 5  
-30 + 3 \* 21 / 5  
30 / 3 \* 21 % 5  
-30 / 3 \* 21 % 4

### 【解答】

各表达式的值分别为-86、-18、0、-2。其中，最后一个表达式的结果值依赖于机器，因为该表达式中除操作只有一个操作数为负数。

### 习题5.3

编写一个表达式判断一个int型数值是偶数还是奇数。

### 【解答】

如下表达式可以判断一个int型数值（假设为ival）是偶数还是奇数：

ival % 2 == 0

若ival是偶数，则该表达式的值为真（true），否则为假（false）。

### 习题5.4

定义术语“溢出”的含义，并给出导致溢出的三个表达式。

### 【解答】

溢出：表达式的求值结果超出了其类型的表示范围。

如下表达式会导致溢出（假设int类型为16位）：

```
1000 * 1000
32766 + 5
3276 * 20
```

在这些表达式中，各操作数均为int类型，因此这些表达式的类型也是int，但它们的计算结果均超出了16位int型的表示范围（-32768~32767），导致溢出。

### 习题5.5

解释逻辑与操作符、逻辑或操作符以及相等操作符的操作数在什么时候计算。

#### 【解答】

逻辑与、逻辑或操作符采用称为“短路求值”（short-circuit evaluation）的求值策略，即先计算左操作数，再计算右操作数，且只有当仅靠左操作数的值无法确定该逻辑运算的结果时，才会计算右操作数。相等操作符的左右操作数均需进行计算。

### 习题5.6

解释下列while循环条件的行为：

```
char *cp = "Hello World";
while ( cp && *cp )
```

#### 【解答】

该while循环的条件为：当指针cp为非空指针并且cp所指向的字符不为空字符null（'\0'）时执行循环体。即该循环可以对字符串"Hello World"中的字符进行逐个处理。

### 习题5.7

编写while循环条件从标准输入设备读入整型(int)数据，当读入值为42时循环结束。

#### 【解答】

```
int val;
cin >> val;
while (val != 42)
```

或者，while循环条件也可以写成

```
while (cin >> ival && ival != 42)
```

### 习题5.8

编写表达式判断4个值a、b、c和d是否满足a大于b、b大于c而且c大于d的条件。

#### 【解答】

表达式如下：

```
a > b && b > c && c > d
```

### 习题5.9

假设有下面两个定义：

```
unsigned long ull = 3, ul2 = 7;
```

下列表达式的结果是什么？

- (a) `ull & ul2`
- (b) `ull && ul2`
- (c) `ull | ul2`
- (d) `ull || ul2`

### 【解答】

各表达式的结果分别为3、`true`、7、`true`。

### 习题5.10

重写bitset表达式：使用下标操作符对测验结果进行置位（置1）和复位（置0）。

### 【解答】

```
bitset<30> bitset_quiz1;
bitset_quiz1[27] = 1;
bitset_quiz1[27] = 0;
```

### 习题5.11

请问每次赋值操作完成后，`i`和`d`的值分别是多少？

```
int i; double d;
d = i = 3.5;
i = d = 3.5;
```

### 【解答】

赋值语句`d=i=3.5;`完成后，`i`和`d`的值均为3。因为赋值操作具有右结合性，所以首先将3.5赋给`i`（此时发生隐式类型转换，将`double`型字面值3.5转换为`int`型值3，赋给`i`），然后将表达式`i=3.5`的值（即赋值后`i`所具有的值3）赋给`d`。

赋值语句`i=d=3.5;`完成后，`d`的值为3.5，`i`的值为3。因为先将字面值3.5赋给`d`，然后将表达式`d=3.5`的值（即赋值后`d`所具有的值3.5）赋给`i`（这时也同样发生隐式类型转换）。

### 习题5.12

解释每个if条件判断产生什么结果？

```
if ( 42 = i ) // ...
if ( i = 42 ) // ...
```

### 【解答】

前者发生语法错误，因为其条件表达式`42=i`是一个赋值表达式，赋值操作符的左操作数必须为一个左值，而字面值42不能作为左值使用。

后者代码合法，但其条件表达式`i=42`是一个永真式（即其逻辑值在任何情况下都为`true`），因为该赋值表达式的值为赋值操作完成后的`i`值（42），而42为非零值，解释为逻辑值`true`。

### 习题5.13

下列赋值操作是不合法的，为什么？怎样改正？

```
double dval; int ival; int *pi;
dval = ival = pi = 0;
```

**【解答】**

该赋值语句不合法，因为该语句首先将0值赋给pi，然后将pi的值赋给ival，再将ival的值赋给dval。pi、ival和dval的类型各不相同，因此要完成赋值必须进行隐式类型转换，但系统无法将int型指针pi的值隐式转换为ival所需的int型值。

可改正如下：

```
double dval; int ival; int *pi;
dval = ival = 0;
pi = 0;
```

**习题5.14**

虽然下列表达式都是合法的，但并不是程序员期望的操作，为什么？怎样修改这些表达式以使其能反映程序员的意图？

- (a) if ( ptr = retrieve\_pointer() != 0 )
- (b) if ( ival = 1024 )
- (c) ival += ival + 1;

**【解答】**

对于表达式(a)，程序员的意图应该是将retrieve\_pointer()的值赋给ptr，然后判断ptr的值是否为0，但因为操作符“=”的优先级比“!=”低，所以该表达式实际上是将retrieve\_pointer()是否为0的判断结果true或false赋给ptr，因此不是程序员期望的操作。

对于表达式(b)，程序员的意图应该是判断ival的值是否与1024相等，但误用了赋值操作符。

对于表达式(c)，程序员的意图应该是使ival的值增加1，但误用了操作符“+=”。

各表达式可修改如下：

- (a) if ( (ptr = retrieve\_pointer()) != 0 )
- (b) if ( ival == 1024 )
- (c) ival += 1; 或 ival++; 或 ++ival;

**习题5.15**

解释前自增操作和后自增操作的差别。

**【解答】**

前自增操作和后自增操作都使其操作数加1，二者的差别在于：前自增操作将修改后操作数的值作为表达式的结果值；而后自增操作将操作数原来的、未修改的值作为表达式的结果值。

**习题5.16**

你认为为什么C++不叫作++C？

**【解答】**

C++之名是Rick Mascitti在1983年夏天定名的（参见《The C++ Programming Language(Special Edition)》1.4节），C说明它本质上是从C语言演化而来的，“++”是C语言的自增操作符。C++语言是C语言的超集，是在C语言基础上进行的扩展（引入了new、delete等C语言中没有的操作符，增加了对面向对象程序设计的直接支持，等等），是先有C语言，再进行++。根据自增操作符前、后置形式的差别（参见习题5.15的解答），C++表示对C语言进行扩展之后，还可以使用C语言的内容；而

写成`++C`则表示无法再使用C的原始值了，也就是说C++不能向下兼容C了，这与实际情况不符。

### 习题5.17

如果输出vector内容的while循环使用前自增操作符，那会怎么样？

#### 【解答】

将导致错误的结果：ivec的第一个元素没有输出，并企图对一个多余的元素进行解引用。

### 习题5.18

编写程序定义一个vector对象，其每个元素都是指向string类型的指针，读取该vector对象，输出每个string的内容及其相应的长度。

#### 【解答】

```
// 定义一个vector对象，其每个元素都是指向string类型的指针
// 读取该vector对象，输出每个string的内容及其相应的长度
#include <iostream>
#include <string>
#include <vector>
using namespace std;

int main()
{
    vector<string*> spvec;

    // 读取vector对象
    string str;
    cout << "Enter some strings(Ctrl+Z to end)" << endl;
    while (cin >> str) {
        string *pstr = new string; // 指向string对象的指针
        *pstr = str;
        spvec.push_back(pstr);
    }

    // 输出每个string的内容及其相应的长度
    vector<string*>::iterator iter = spvec.begin();
    while (iter != spvec.end()) {
        cout << **iter << (**iter).size() << endl;
        iter++;
    }

    // 释放各个动态分配的string对象
    iter = spvec.begin();
    while (iter != spvec.end()) {
        delete *iter;
        iter++;
    }
}

return 0;
}
```

### 习题5.19

假设iter为`vector<string>::iterator`类型的变量，指出下面哪些表达式是合法的，并解释这些合法表达式的行为。

- (a) `*iter++;`      (b) `(*iter)++;`

- (c) `*iter.empty();`      (d) `iter->empty();`  
 (e) `++*iter;`      (f) `iter++->empty();`

**【解答】**

(a)、(d)、(f)合法。

这些表达式的执行结果如下：

- (a) 返回 `iter` 所指向的 `string` 对象，并使 `iter` 加 1。  
 (d) 调用 `iter` 所指向的 `string` 对象的成员函数 `empty`。  
 (f) 调用 `iter` 所指向的 `string` 对象的成员函数 `empty`，并使 `iter` 加 1。

**习题5.20**

编写程序提示用户输入两个数，然后报告哪个数比较小。

**【解答】**

可编写程序如下：

```
// 提示用户输入两个数，然后报告哪个数比较小
#include <iostream>
using namespace std;

int main()
{
    int val1, val2;

    // 提示用户输入两个数并接受输入
    cout << "Enter two integers:" << endl;
    cin >> val1 >> val2;

    // 报告哪个数比较小
    cout << "The smaller one is"
        << (val1 < val2 ? val1 : val2) << endl;

    return 0;
}
```

**习题5.21**

编写程序处理 `vector<int>` 对象的元素：将每个奇数值元素用该值的两倍替换。

**【解答】**

```
// 处理 vector<int> 对象的元素：
// 将每个奇数值元素用该值的两倍替换
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    vector<int> ivec(20, 1); // ivec 包含 20 个值为 1 的元素

    // 将每个奇数值元素用该值的两倍替换
    for (vector<int>::iterator iter = ivec.begin();
         iter != ivec.end(); ++iter)
        *iter = (*iter % 2 == 0 ? *iter : *iter * 2);
```



```

    return 0;
}

```

**习题5.22**

编写程序输出每种内置类型的长度。

**【解答】**

```

//输出每种内置类型的长度
#include <iostream>
using namespace std;

int main()
{
    cout << "type\t\t\t" << "size" << endl
        << "bool\t\t\t" << sizeof(bool) << endl
        << "char\t\t\t" << sizeof(char) << endl
        << "signed char\t\t" << sizeof(signed char) << endl
        << "unsigned char\t\t" << sizeof(unsigned char) << endl
        << "wchar_t\t\t\t" << sizeof(wchar_t) << endl
        << "short\t\t\t" << sizeof(short) << endl
        << "signed short\t\t" << sizeof(signed short) << endl
        << "unsigned short\t\t" << sizeof(unsigned short) << endl
        << "int\t\t\t" << sizeof(int) << endl
        << "signed int\t\t" << sizeof(signed int) << endl
        << "unsigned int\t\t" << sizeof(unsigned int) << endl
        << "long\t\t\t" << sizeof(long) << endl
        << "signed long\t\t" << sizeof(signed long) << endl
        << "unsigned long\t\t" << sizeof(unsigned long) << endl
        << "float\t\t\t" << sizeof(float) << endl
        << "double\t\t\t" << sizeof(double) << endl
        << "long double\t\t" << sizeof(long double) << endl;

    return 0;
}

```

**习题5.23**

预测下列程序的输出，并解释你的理由。然后运行该程序，输出的结果和你预测的一样吗？如果不一样，为什么？

```

int x[10]; int *p = x;
cout << sizeof(x)/sizeof(*x) << endl;
cout << sizeof(p)/sizeof(*p) << endl;

```

**【解答】**

在表达式`sizeof(x)`中，`x`是数组名，该表达式的结果为数组`x`所占据的存储空间的字节数，为10个`int`型元素所占据的字节数。

表达式`sizeof(*x)`的结果是指针常量`x`所指向的对象（数组中第一个`int`型元素）所占据的存储空间的字节数。

表达式`sizeof(p)`的结果是指针变量`p`所占据的存储空间的字节数。

表达式`sizeof(*p)`的结果是指针变量`p`所指向的对象（一个`int`型数据）所占据的存储空间的字节数。

各种数据类型在不同的系统中所占据的字节数不一定相同，因此在不同的系统中运行上述程序段

得到的结果不一定相同。在Microsoft Visual C++ .NET 2003系统中，一个int型数据占据4个字节，一个指针型数据也占据4个字节，因此运行上述程序得到的输出结果为：

```
10
1
```

### 习题5.24

本节的程序与5.5节在vector对象中添加元素的程序类似。两段程序都使用递减的计数器生成元素的值。本程序中，我们使用了前自减操作，而5.5节的程序则使用了后自减操作。解释为什么一段程序中使用前自减操作而在另一段程序中使用后自减操作。

#### 【解答】

5.5节的程序中必须使用后自减操作。如果使用前自减操作，则是用减1后的cnt值创建ivec的新元素，添加到ivec中的元素将不是10~1，而是9~0。

本节的程序中使用后自减操作或前自减操作均可，因为对cnt的自减操作和对cnt值的使用不是出现在同一表达式中，cnt自减操作的前置或后置形式不影响对cnt值的使用。

### 习题5.25

根据表5-4的内容，在下列表达式中添加圆括号说明其操作数分组的顺序（即计算顺序）：

- (a) ! ptr == ptr->next
- (b) ch = buf[ bp++ ] != '\n'

#### 【解答】

添加圆括号说明其计算顺序如下：

- (a) (! (ptr == (ptr->next)))
- (b) (ch = ((buf[ (bp++) ]) != '\n'))

### 习题5.26

习题5.25中的表达式的计算次序与你的意图不同，给它们加上圆括号使其以你所希望的操作次序求解。

#### 【解答】

添加圆括号获得与上题不同的操作次序如下：

- (a) ! (ptr == ptr->next)
- (b) (ch = buf[ bp++ ]) != '\n'

### 习题5.27

由于操作符优先级的问题，下列表达式编译失败。请参照表5-4解释原因，应该如何改正？

```
string s = "word";
// add an 's' to the end, if the word doesn't already end in 's'
string pl = s + s[s.size() - 1] == 's' ? "" : "s" ;
```

#### 【解答】

由表5-4可知，在语句string pl = s + s[s.size() - 1] == 's' ? "" : "s" ;中，赋值、

加法、条件操作符三者的操作次序为：先执行“+”操作，再用表达式`s + s[s.size() - 1]`的结果参与条件操作，最后将条件操作的结果赋给`pl`。但表达式`s + s[s.size() - 1]`的结果是一个`string`对象，不能与字符's'进行相等比较，所以编译失败。

改正为：`string pl = s + (s[s.size() - 1] == 's' ? "" : "s");`

### 习题5.28

除了逻辑与和逻辑或外，C++没有明确定义二元操作符的求解次序，编译器可自由地提供最佳的实现方式。只能在“实现效率”和程序语言使用中“潜在的缺陷”之间寻求平衡。你认为这可以接受吗？说出你的理由。

#### 【解答】

这可以接受。

因为，操作数的求解次序通常对结果没什么影响。只有当二元操作符的两个操作数涉及同一对象，并改变该对象的值时，操作数的求解次序才会影响计算结果；后一种情况只会在部分（甚至是少数）程序中出现。在实际使用中，这种“潜在的缺陷”可以通过程序员的努力得到弥补，但“实现效率”的提高却能使所有使用该编译器的程序受益，因此利大于弊。

### 习题5.29

假设`ptr`指向类类型对象，该类拥有一个名为`ival`的`int`型数据成员，`vec`是保存`int`型元素的`vector`对象，而`ival`、`jval`和`kval`都是`int`型变量。请解释下列表达式的行为，并指出哪些（如果有话）可能是不正确的，为什么？如何改正？

- |  |   |
|--|---|
| (a) <code>ptr-&gt;ival != 0</code>           | (b) <code>ival != jval &lt; kval</code> |
| (c) <code>ptr != 0 &amp;&amp; *ptr++</code>  | (d) <code>ival++ &amp;&amp; ival</code> |
| (e) <code>vec[ival++] &lt;= vec[ival]</code> |   |

#### 【解答】

表达式的行为如下：

- (a) 判断`ptr`所指向的对象的`ival`成员是否不等于0。
- (b) 判断`ival`是否不等于“`jval`是否小于`kval`”的判断结果，即判断`ival`是否不等于`true`(1)或`false`(0)。
- (c) 判断`ptr`是否不等于0。如果`ptr`不等于0，则求解`&&`操作的右操作数，即，`ptr`加1，且判断`ptr`原来所指向的对象是否为0。
- (d) 判断`ival`及`ival+1`是否为`true`(非0值)(注意，如果`ival`为`false`，则无需继续判断`ival+1`)。
- (e) 判断`vec[ival]`是否小于或等于`vec[ival+1]`。

其中，(d)和(e)可能不正确，因为二元操作符的两个操作数涉及同一对象，并改变该对象的值。

可改正如：

- (d) `ival && ival + 1`
- (e) `vec[ival] <= vec[ival + 1]`

### 习题5.30

下列语句哪些（如果有话）是非法的或错误的？

- (a) `vector<string> svec(10);`
- (b) `vector<string> *pvec1 = new vector<string>(10);`
- (c) `vector<string> **pvec2 = new vector<string>[10];`
- (d) `vector<string> *pvl = &svec;`
- (e) `vector<string> *pv2 = pvec1;`
- (f) `delete svec;`
- (g) `delete pvec1;`
- (h) `delete [] pvec2;`
- (i) `delete pvl;`
- (j) `delete pv2;`

**【解答】**

错误的有(c)和(f)。

(c)的错误在于：`pvec2`是指向元素类型为`string`的`vector`对象的指针的指针（即`pvec2`的类型为`vector<string> **`），而`new`操作返回的是一个指向元素类型为`string`的`vector`对象的指针，不能用于初始化`pvec2`。

(f)的错误在于：`svec`是一个`vector`对象，不是指针，不能对它进行`delete`操作。

**习题5.31**

根据5.12.2节的变量定义，解释在计算下列表达式的过程中发生了什么类型转换？

- (a) `if (fval)`
- (b) `dval = fval + ival;`
- (c) `dval + ival + cval;`

记住，你可能需要考虑操作符的结合性，以便在表达式含有多个操作符的情况下确定答案。

**【解答】**

- (a) 将`fval`的值从`float`类型转换为`bool`类型。
- (b) 将`ival`的值从`int`类型转换为`float`类型，再将`fval + ival`的结果值转换为`double`类型，赋给`dval`。
- (c) 将`ival`的值从`int`类型转换为`double`类型，`cval`的值首先提升为`int`类型，然后从`int`型转换为`double`型，与`dval + ival`的结果值相加。

**习题5.32**

给定下列定义：

```
char cval; int ival; unsigned int ui;
float fval; double dval;
```

指出可能发生的（如果有的话）隐式类型转换：

- |                                    |   |
|------------------------------------|---|
| (a) <code>cval = 'a' + 3;</code>   | (b) <code>fval = ui - ival * 1.0;</code>    |
| (c) <code>dval = ui * fval;</code> | (d) <code>cval = ival + fval + dval;</code> |

**【解答】**

- (a) '`a`'首先提升为`int`类型，再将'`a`' + 3的结果值转换为`char`型，赋给`cval`。
- (b) `ival`转换为`double`型与1.0相乘，`ui`转换为`double`型再减去`ival * 1.0`的结果值，减操作的

结果转换为float型，赋给fval。

- (c) ui转换为float型与fval相乘，结果转换为double型，赋给dval。  
(d) ival转换为float型与fval相加，结果转换为double型，再与dval相加，结果转换为char型，赋给cval。

### 习题5.33

给定下列定义：

```
int ival; double dval;
const string *ps; char *pc; void *pv;
```

用命名的强制类型转换符号重写下列语句：

- (a) `pv = (void*)ps;`      (b) `ival = int(*pc);`  
(c) `pv = &dval;`      (d) `pc = (char*) pv;`

### 【解答】

- (a) `pv = static_cast<void*>(const_cast<string*>(ps));`  
(b) `ival = static_cast<int>(*pc);`  
(c) `pv = static_cast<void*>(&dval);`  
(d) `pc = static_cast<char*>(pv);`

# 第 6 章

## 语句

### 习题6.1

什么是空语句？请给出一个使用空语句的例子。

#### 【解答】

空语句是由一个单独的分号构成的语句。例如，

```
int ival;
while (cin >> ival && ival != -1)
; // 空语句
```

该循环语句从输入流中读取数据，在遇到文件结束符或读到特定值（-1）前不做任何操作。

### 习题6.2

什么是块语句？请给出一个使用块的例子。

#### 【解答】

块语句是由一对花括号括住的语句序列（该语句序列可以为空）。例如，

```
int ival, cnt = 0, sum = 0;
while (cin >> ival) {
    sum += ival;
    ++cnt;
}
```

该循环语句从输入流中读取一系列数据，求读入数据的个数及其总和。

### 习题6.3

使用逗号操作符（5.9节）重写书店问题中while循环里的else分支，使它不再需要用块实现。解释一下重写后是提高了还是降低了该段代码的可读性。

#### 【解答】

该else 分支可改写为：

```
else
    std::cout << total << std::endl, total = trans;
```

重写后的语句太长，降低了该段代码的可读性。

### 习题6.4

在解决书店问题的while循环中，如果删去while后面的左花括号及相应的右花括号，将会给程序

带来什么影响？

### 【解答】

将会使得程序只计算并输出ISBN相同的第一组trans的总和，然后将total置为最后读入的那个trans。

### 习题6.5

改正下列代码：

- (a) if (ival1 != ival2)
 

```
    ival1 = ival2
else ival1 = ival2 = 0;
```
- (b) if (ival < minval)
 

```
    minval = ival; // remember new minimum
occurs = 1; // reset occurrence counter
```
- (c) if (int ival = get\_value())
 

```
    cout << "ival = " << ival << endl;
if (!ival)
    cout << "ival = 0\n";
```
- (d) if (ival = 0)
 

```
    ival = get_value();
```

### 【解答】

改正为：

- (a) if (ival1 != ival2)
 

```
    ival1 = ival2;
else ival1 = ival2 = 0;
```
- (b) if (ival < minval) {
 

```
    minval = ival; // remember new minimum
    occurs = 1; // reset occurrence counter
}
```
- (c) int ival;
 

```
if (ival = get_value())
    cout << "ival = " << ival << endl;
if (!ival)
    cout << "ival = 0\n";
```
- (d) if (ival == 0)
 

```
    ival = get_value();
```

### 习题6.6

什么是“悬垂else”？C++是如何匹配else子句的？

### 【解答】

所谓“悬垂else”，指的是当一个语句包含的if子句多于else子句时，各个else子句应该与哪个if子句相匹配。

C++中，将else子句与出现在它前面的距离最近的尚未匹配的if子句相匹配。

### 习题6.7

前面已实现的统计元音的程序存在一个问题：不能统计大写的元音字母。编写程序统计大小写的元音，也就是说，你的程序计算出来的acnt，既包括'a'也包括'A'出现的次数，其他四个元音也一样。

**【解答】**

```

// 统计读入的文本中大小写元音字母的个数
#include <iostream>
using namespace std;

int main()
{
    char ch;

    // 初始化每个元音的计数器
    int aCnt = 0, eCnt = 0, iCnt = 0, oCnt = 0, uCnt = 0;

    while (cin >> ch) {
        // 若ch是元音，将相应计数器加1
        switch (ch) {
            case 'a':
            case 'A':
                ++aCnt;
                break;
            case 'e':
            case 'E':
                ++eCnt;
                break;
            case 'i':
            case 'I':
                ++iCnt;
                break;
            case 'o':
            case 'O':
                ++oCnt;
                break;
            case 'u':
            case 'U':
                ++uCnt;
                break;
        }
    }

    // 输出结果
    cout << "Number of vowel a: \t" << aCnt << '\n'
        << "Number of vowel e: \t" << eCnt << '\n'
        << "Number of vowel i: \t" << iCnt << '\n'
        << "Number of vowel o: \t" << oCnt << '\n'
        << "Number of vowel u: \t" << uCnt << endl;

    return 0;
}

```

注意，初学者容易将诸如

```
case 'a';
case 'A':
```

这样的case分支（又称case标号）写成

```
case 'a', 'A':
```

这会出现编译错误，因为case标号中的值必须是常量表达式（不包括逗号表达式）。

**习题6.8**

修改元音统计程序使其可统计出读入的空格、制表符和换行符的个数。

**【解答】**

```
// 统计读入的文本中大小写元音字母以及空格、制表符和换行符的个数
#include <iostream>
using namespace std;

int main()
{
    char ch;

    // 初始化各个计数器
    int aCnt = 0, eCnt = 0, iCnt = 0, oCnt = 0, uCnt = 0;
    int spaceCnt = 0, tabCnt = 0, newlineCnt = 0;

    while (cin.get(ch)) {
        // 若ch是元音字母以及空格、制表符和换行符，将相应计数器加1
        switch (ch) {
            case 'a':
            case 'A':
                ++aCnt;
                break;
            case 'e':
            case 'E':
                ++eCnt;
                break;
            case 'i':
            case 'I':
                ++iCnt;
                break;
            case 'o':
            case 'O':
                ++oCnt;
                break;
            case 'u':
            case 'U':
                ++uCnt;
                break;
            case ' ':
                ++spaceCnt;
                break;
            case '\t':
                ++tabCnt;
                break;
            case '\n':
                ++newlineCnt;
                break;
        }
    }

    // 输出结果
    cout << "Number of vowel a: \t" << aCnt << '\n'
        << "Number of vowel e: \t" << eCnt << '\n'
        << "Number of vowel i: \t" << iCnt << '\n'
        << "Number of vowel o: \t" << oCnt << '\n'
        << "Number of vowel u: \t" << uCnt << '\n'
        << "Number of space character: \t" << spaceCnt << '\n'
        << "Number of Tab character: \t" << tabCnt << '\n'
        << "Number of newline character: \t" << newlineCnt << endl;
}
```

```

    return 0;
}

```

注意，此处读入数据时不能使用提取操作符(>>)，因为对>>而言，空格、制表符和换行符均为数据项分隔符，会被忽略掉，而cin对象的get成员函数则不会这样。

### 习题6.9

修改元音统计程序使其可统计以下双字符序列出现的次数：ff、fl以及fi。

#### 【解答】

```

// 统计读入文本中双字符序列ff、fl以及fi出现的次数
#include <iostream>
using namespace std;

int main()
{
    char currCh, preCh = '\0'; // 分别记录当前读入字符及其前一字符

    // 初始化各个计数器
    int ffCnt = 0, flCnt = 0, fiCnt = 0;
    while (cin >> currCh) {
        if (preCh == 'f') // 若前一字符是'f'
            // 根据当前读入字符，将相应计数器加1
            switch (currCh) {
                case 'f':
                    ++ffCnt;
                    break;
                case 'l':
                    ++flCnt;
                    break;
                case 'i':
                    ++fiCnt;
                    break;
            }
        preCh = currCh; // 将前一字符置为当前字符
    }

    // 输出结果
    cout << "Number of \"ff\": \t" << ffCnt << '\n'
        << "Number of \"fl\": \t" << flCnt << '\n'
        << "Number of \"fi\": \t" << fiCnt << endl;

    return 0;
}

```

### 习题6.10

下面每段代码都暴露了一个常见编程错误。请指出并修改之。

(a) `switch (ival) {  
 case 'a': aCnt++;  
 case 'e': eCnt++;  
 default: iouCnt++;  
}`

(b) `switch (ival) {  
 case 1:  
 int ix = get_value();  
 ivec[ ix ] = ival;`

```

        break;
    default:
        ix = ivec.size()-1;
        ivec[ ix ] = ival;
    }
(c) switch (ival) {
    case 1, 3, 5, 7, 9:
        oddcnt++;
        break;
    case 2, 4, 6, 8, 10:
        evencnt++;
        break;
}
(d) int ival=512, jval=1024, kval=4096;
    int bufsize;
    // ...
    switch(swt) {
        case ival:
            bufsize = ival * sizeof(int);
            break;
        case jval:
            bufsize = jval * sizeof(int);
            break;
        case kval:
            bufsize = kval * sizeof(int);
            break;
    }
}

```

**【解答】**

(a)的错误在于：各个case标号对应的语句块中缺少必要的break语句，从而当ival值为'a'时，aCnt、eCnt和iouCnt都会加1；ival值为'e'时，eCnt和iouCnt都会加1。

修改为：

```

switch (ival) {
    case 'a':
        aCnt++;
        break;
    case 'e':
        eCnt++;
        break;
    default:
        iouCnt++;
        break; //此语句省略亦可
}

```

(b)的错误在于：在case 1标号之后、default标号之前定义了变量ix。因为，对于switch结构，只能在它的最后一个case标号或default标号后面定义变量，以避免出现代码跳过变量的定义和初始化的情况。

修改为：

```

int ix;
switch (ival) {
    case 1:
        ix = get_value();
        ivec[ ix ] = ival;
        break;
    default:
        ix = ivec.size()-1;
}

```

```

    ivec[ ix ] = ival;
}

```

(c)的错误在于：case标号中出现了多个值。因为一个case标号只能与一个值相关联。

修改为：

```

switch (ival) {
    case 1:
    case 3:
    case 5:
    case 7:
    case 9:
        oddcnt++;
        break;
    case 2:
    case 4:
    case 6:
    case 8:
    case 10:
        evencnt++;
        break;
}

```

(d)的错误在于：case标号中不能使用ival、jval和kval。因为case标号中的值只能使用常量表达式，而ival、jval和kval都是变量。

将语句int ival=512, jval=1024, kval=4096;修改为：

```
const int ival=512, jval=1024, kval=4096;
```

### 习题6.11

解释下面的循环，更正你发现的问题。

- (a) string bufString, word;  
    while (cin >> bufString >> word) { /\* ... \*/ }
- (b) while (vector<int>::iterator iter != ivec.end())  
    { /\*... \*/ }
- (c) while (ptr = 0)  
    ptr = find\_a\_value();
- (d) while (bool status = find(word))  
    { word = get\_next\_word(); }  
    if (!status)  
        cout << "Did not find any words\n";

#### 【解答】

- (a) 每次读入两个string对象，直到遇到文件结束符。
- (b) 依次处理vector中的每个元素。此循环有错误：iter没有赋初值。

更正为：

```

vector<int>::iterator iter = ivec.begin();
while (iter != ivec.end())
{ /*... */ }

```

- (c) 调用find\_a\_value函数，将返回值赋给ptr，直到ptr为0。此循环有错误：条件表达式中应使用比较操作符，而不是赋值操作符。

更正为：

```
while (ptr != 0)
ptr = find_a_value();
```

- (d) 每次调用get\_next\_word()获取一个word，然后调用find函数查找该word，直到找不到该word为止。此循环有错误：word没有赋初值。

更正为：

```
word = get_next_word();
while (bool status = find(word))
{ word = get_next_word(); }
if (!status)
cout << "Did not find any words\n";
```

### 习题6.12

编写一个小程序，从标准输入读入一系列string对象，寻找连续重复出现的单词。程序应该找出满足以下条件的单词的输入位置：该单词的后面紧跟着再次出现自己本身。跟踪重复次数最多的单词及其重复次数。输出重复次数的最大值，若没有单词重复则输出说明信息。例如，如果输入是：

```
how, now now brown cow cow
```

则输出应表明“now”这个单词出现了三次。

#### 【解答】

```
// 6-12.cpp
// 从标准输入读入一系列string对象，寻找连续重复出现的单词。
// 输出重复次数的最大值，若没有单词重复则输出说明信息
#include <iostream>
#include <string>
using namespace std;

int main()
{
    string preWord, currWord; // 当前输入的单词及其前一单词
    string repWord; // 重复次数最多的单词
    // 当前单词的重复次数及单词重复次数的最大值
    int currCnt = 0, maxCnt = 1;

    cout << "Enter some words(Ctrl+Z to end):" << endl;
    while (cin >> currWord) {
        if (currWord == preWord) // 当前单词是重复出现
            ++currCnt;
        else // 当前单词不是前一单词的重复出现
            if (currCnt > maxCnt) // 出现了重复次数更多的单词
                maxCnt = currCnt;
                repWord = preWord;
            }
        currCnt = 1;
    }
    preWord = currWord; // 修改对前一单词的记录
}

if (maxCnt != 1) // 有单词重复
    cout << '\'' << repWord << '\''
    << " repeated for " << maxCnt
    << " times." << endl;
else
```



```

    cout << "There is no repeated word." << endl;
    return 0;
}

```

**习题6.13**

详细解释下面while循环中的语句是如何执行的:

```
*dest++ = *source++;
```

**【解答】**

执行过程如下: (1)指针dest加1; (2)指针source加1; (3)将source原来所指向的对象赋给dest原来所指向的对象。

**习题6.14**

解释下面每个循环,更正你发现的任何问题。

- (a) 

```
for (int *ptr = &ia, ix = 0;
      ix != size && ptr != ia+size;
      ++ix, ++ptr) { /* ... */ }
```
- (b) 

```
for ( ; ; ) {
    if (some_condition) return;
    // ...
}
```
- (c) 

```
for (int ix = 0; ix != sz; ++ix) { /* ... */ }
if (ix != sz)
    // ...
```
- (d) 

```
int ix;
for (ix != sz; ++ix) { /* ... */ }
```
- (e) 

```
for (int ix = 0; ix != sz; ++ix, ++ sz) { /* ... */ }
```

**【解答】**

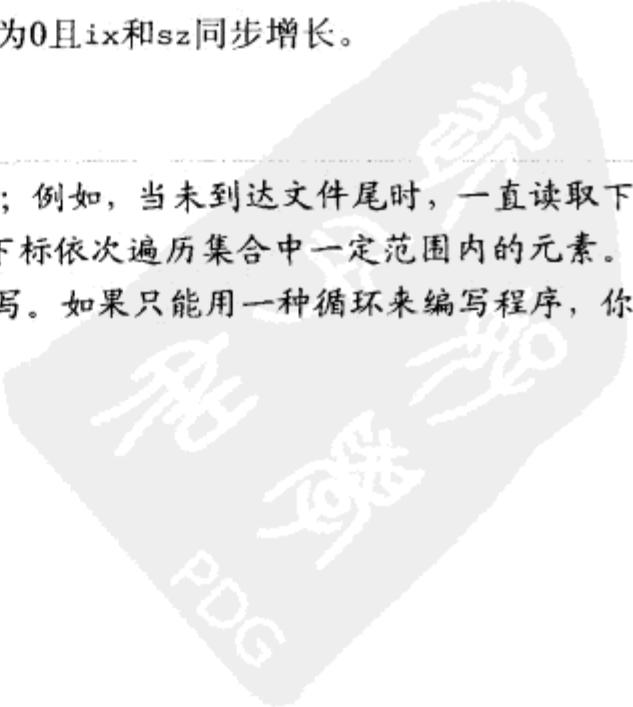
- (a) 依次处理ia中的每个元素。此循环有错误: ptr应初始化为ia,而不是&ia。此外,在ix != size && ptr != ia+size;语句中, ix != size和ptr != ia+size是等价的条件,重复了,可去掉其中一个。
- (b) 循环至some\_condition为真(true)时结束。
- (c) 当ix不等于sz(ix为0~sz-1)时执行循环体。(循环体中应该有break或return语句跳出循环,否则后面if语句中的条件表达式的值永远为假。)
- (d) 当ix不等于sz时执行循环体。此循环有错误: ix没有赋初值。更正为:

```
int ix;
for (ix = 0; ix != sz; ++ix) { /* ... */ }
```

- (e) 如果sz不等于0,这是一个死循环,因为ix的初始值为0且ix和sz同步增长。

**习题6.15**

while 循环特别擅长在某个条件保持为真时反复地执行;例如,当未到达文件尾时,一直读取下一个值。一般认为for循环是一种按步骤执行的循环:使用下标依次遍历集合中一定范围内的元素。按每种循环的习惯用法编写程序,然后再用另外一种结构重写。如果只能用一种循环来编写程序,你会选择哪种结构?为什么?



**【解答】**

若有如下定义：

```
const size_t size = 100;
int ia[size];
```

下面三个程序段等价：

```
//程序段1
size_t ix = 0;
while (ix != size) {
    /*...*/ // 对ia的元素进行处理
    ++ix;
}

//程序段2
for (size_t ix = 0; ix != size; ++ix) {
    /*...*/ // 对ia的元素进行处理
}

//程序段3
size_t ix = 0;
do {
    /*...*/ // 对ia的元素进行处理
    ++ix;
} while (ix != size);
```

如果只能用一种循环来编写程序，更愿意选择for结构。

三种循环结构可以实现功能上的等价，但for结构的形式最为简洁灵活。

**习题6.16**

给出两个int型的vector对象，编写程序判断一个对象是否是另一个对象的前缀。如果两个vector对象的长度不相同，假设较短的vector对象长度为n，则只对这两个对象的前面n个元素做比较。例如，对于(0,1,1,2)和(0,1,1,2,3,5,8)这两个vector，你的程序应该返回true。

**【解答】**

```
// 6-16.cpp
// 给出两个int型的vector对象，判断一个对象是否是另一个对象的前缀
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    vector<int> ivect1, ivect2;
    int ival;

    // 读入第一个vector对象的元素
    cout << "Enter elements for the first vector:(32767 to end)"
        << endl;
    cin >> ival;
    while (ival != 32767) {
        ivect1.push_back(ival);
        cin >> ival;
    }

    // 读入第二个vector对象的元素
    cout << "Enter elements for the second vector:(32767 to end)"
```

```

        << endl;
cin >> ival;
while (ival != 32767) {
    ivec2.push_back(ival);
    cin >> ival;
}

// 比较两个vector对象
vector<int>::size_type size1, size2;
size1 = ivec1.size();
size2 = ivec2.size();
bool result = true;
for (vector<int>::size_type ix = 0;
     ix != (size1 > size2 ? size2 : size1); ++ix)
    if (ivec1[ix] != ivec2[ix]) {
        result = false;
        break;
    }

// 输出结果
if (result)
    if (size1 < size2)
        cout << "The first vector is prefix of the second one."
            << endl;
    else if (size1 == size2)
        cout << "Two vectors are equal." << endl;
    else
        cout << "The second vector is prefix of the first one."
            << endl;
else
    cout << "No vector is prefix of the other one.";

return 0;
}

```

注意，在输入两个vector的元素时也可以不用特定值32767标记输入结束，而使用文件结束符（输入Ctrl+Z）来控制元素输入的结束。但是，使用后一种方法时，在第二个输入循环之前要记得将流cin恢复为有效状态（使用cin.clear()）。

### 习题6.17

解释下列循环。更正你发现的问题。

(a) do

```

int v1, v2;
cout << "Please enter two numbers to sum: ";
cin >> v1 >> v2;
if (cin)
    cout << "Sum is: "
        << v1 + v2 << endl;

```

while (cin);

(b) do {

// ...

} while (int ival = get\_response());

(c) do {

```

int ival = get_response();
if (ival == some_value())
    break;

```

} while (ival);

if (!ival)

```
// ...
```

**【解答】**

- (a) 每次循环读入两个整数，输出它们的和。此循环有错误：循环体应该用一对花括号括住。更正为：

```
do {
    int v1, v2;
    cout << "Please enter two numbers to sum: " ;
    cin >> v1 >> v2;
    if (cin)
        cout << "Sum is: "
            << v1 + v2 << endl;
} while (cin);
```

- (b) 当 `get_response` 函数返回真值时执行循环体。此循环有错误：`do while` 循环的条件中不能定义变量。更正为：

```
do {
    int ival = get_response();
    // ...
} while (ival);
```

- (c) 当 `get_response` 函数返回真值且该返回值不等于 `some_value` 函数的返回值时执行循环体。此循环有错误：循环体中定义的变量 `ival` 不能在 `do while` 语句外使用，因此不能在 `if` 语句中使用。更正为：

```
int ival;
do {
    ival = get_response();
    if (ival == some_value())
        break;
} while (ival);
if (!ival)
// ...
```

**习题6.18**

编写一个小程序，由用户输入两个 `string` 对象，然后报告哪个 `string` 对象按字母排列次序而言比较小（也就是说，哪个对象的字典序靠前）。继续要求用户输入，直到用户请求退出为止。请使用 `string` 类型、`string` 类型的小于操作符以及 `do while` 循环实现。

**【解答】**

```
// 6-18.cpp
// 报告由用户输入的两个string对象中哪个对象的字典序靠前。
// 连续要求用户输入，直到用户请求退出为止
#include <iostream>
#include <string>
using namespace std;

int main()
{
    string str1, str2, str3;
    do {
        cout << "Enter two strings:" << endl;
        cin >> str1 >> str2;
```

```

        if (str1 < str2)
            cout << "The first string is smaller than the second one."
                << endl;
        else if (str1 == str2)
            cout << "Two strings are equal." << endl;
        else
            cout << "The second string is smaller than the first one."
                << endl;

        cout << "Continue?(y-yes, n-no)" << endl;
        cin >> str3;
    } while (str3[0] != 'n' && str3[0] != 'N');

    return 0;
}

```

**习题6.19**

本节的第一个程序可以写得更简洁。事实上，该程序的所有工作可以全部包含在while的循环条件中。重写这个循环，使得它的循环体为空，并找出满足条件的元素。

**【解答】**

重写循环如下：

```
while (iter != vec.end() && value != *iter++) {
} // end of while
```

**习题6.20**

编写程序从标准输入读入一系列string对象，直到同一个单词连续出现两次，或者所有的单词都已读完，才结束读取。请使用while循环，每次循环读入一个单词。如果连续出现相同的单词，便以break语句结束循环，此时，请输出这个重复出现的单词；否则输出没有任何单词连续重复出现的信息。

**【解答】**

```

// 6-20.cpp
// 从标准输入读入一系列string对象,
// 直到同一个单词连续出现两次, 或者所有的单词都已读完, 才结束读取。
// 输出重复出现的单词, 或者输出没有任何单词连续重复出现的信息
#include <iostream>
#include <string>
using namespace std;

int main()
{
    string currWord, preWord; // 记录当前单词及其前一单词
    cout << "Enter some words:(Ctrl+Z to end)" << endl;
    while (cin >> currWord) {
        if (currWord == preWord) // 当前单词是重复出现
            break;
        else
            preWord = currWord;
    }

    // 输出结果
    if (currWord == preWord && !currWord.empty())
        cout << "The repeated word: " << currWord << endl;
    else
        cout << "There is no repeated word." << endl;
}

```

```

        return 0;
    }
}

```

**习题6.21**

修改6.10节最后一个习题的程序，使得它只寻找以大写字母开头的连续出现的单词。

**【解答】**

```

// 6-21.cpp
// 从标准输入读入一系列string对象,
// 直到同一个单词连续出现两次, 或者所有的单词都已读完, 才结束读取
// 输出重复出现的单词, 或者输出没有任何单词连续重复出现的信息
// 只寻找以大写字母开头的连续出现的单词
#include <iostream>
#include <string>
#include <cctype>
using namespace std;

int main()
{
    string currWord, preWord; // 记录当前单词及其前一单词
    cout << "Enter some words:(Ctrl+Z to end)" << endl;
    while (cin >> currWord) {
        if (currWord == preWord && isupper(currWord[0]))
            // 当前单词是重复出现且以大写字母开头
            break;
        else
            preWord = currWord;
    }

    // 输出结果
    if (currWord == preWord && !currWord.empty())
        cout << "The repeated word: " << currWord << endl;
    else
        cout << "There is no repeated word." << endl;
}

return 0;
}

```

**习题6.22**

对于本节的最后一个例子，跳回到begin标号的功能可以用循环更好地实现。请不使用goto语句重写这段代码。

**【解答】**

重写代码如下：

```

do {
    int sz = get_size();
} while (sz <= 0);

```

**习题6.23**

bitset类提供to\_ulong操作。如果bitset提供的位数大于unsigned long的长度，则抛出一个overflow\_error异常。编写产生这种异常的程序。

**【解答】**

```

// 6-23.cpp
// 编写使bitset类的to_ulong操作产生overflow_error异常的程序

```

```

#include <iostream>
#include <bitset>
using namespace std;

int main()
{
    bitset<100> bs; // 定义100位的bitset对象

    // 将bitset对象的每一位置为1
    for (size_t ix = 0; ix != bs.size(); ++ix)
        bs[ix] = 1;

    bs.to_ulong(); // 执行to_ulong操作将产生overflow_error异常

    return 0;
}

```

**习题6.24**

修改上述程序，使它能捕获这种异常并输出提示信息。

**【解答】**

使用try块来捕获异常并输出提示信息，程序如下：

```

// 6-24.cpp
// 编写使bitset类的to_ulong操作产生overflow_error异常的程序。
// 使用try块来捕获异常并输出提示信息
#include <iostream>
#include <bitset>
using namespace std;

int main()
{
    bitset<100> bs; // 定义100位的bitset对象

    // 将bitset对象的每一位置为1
    for (size_t ix = 0; ix != bs.size(); ++ix)
        bs[ix] = 1;

    try {
        bs.to_ulong(); // 执行to_ulong操作将产生overflow_error异常
    } catch (runtime_error err) { // 处理异常
        cout << err.what() << endl;
    }

    return 0;
}

```

**习题6.25**

修改6.11节习题所编写的程序，使其可以有条件地输出运行时的信息。例如，可以输出每一个读入的单词，用来判断循环是否正确地找到第一个连续出现的以大写字母开头的单词。分别在打开和关闭调试器的情况下编译和运行这个程序。

**【解答】**

```

// 6-25.cpp
// 从标准输入读入一系列string对象,
// 直到同一个单词连续出现两次，或者所有的单词都已读完，才结束读取。

```

```

// 输出重复出现的单词，或者输出没有任何单词连续重复出现的信息。
// 使用有条件的调试代码有条件地输出每一个读入的单词
#include <iostream>
#include <string>
#include <cctype>
using namespace std;

int main()
{
    string currWord, preWord; // 记录当前单词及其前一单词
    cout << "Enter some words:(Ctrl+Z to end)" << endl;
    while (cin >> currWord) {
        // 若未定义NDEBUG，则输出读入的单词
        #ifndef NDEBUG
        cout << currWord << " ";
        #endif
        if (!isupper(currWord[0])) // 单词不是以大写字母开头
            continue;
        if (currWord == preWord)
            // 当前单词是重复出现且以大写字母开头
            break;
        else
            preWord = currWord;
    }

    // 输出结果
    if (currWord == preWord && !currWord.empty())
        cout << "The repeated word: " << currWord << endl;
    else
        cout << "There is no repeated word that has initial capital."
        << endl;
}

return 0;
}

```

在打开调试器的情况下编译和运行该程序，会输出所读入的每个单词；如果在关闭调试器的情况下编译和运行该程序，则不会输出所读入的每个单词。

### 习题6.26

下面循环会导致什么现象的发生：

```

string s;
while (cin >> s) {
    assert(cin);
    // process s
}

```

解释这种用法是否是assert宏的一种恰当应用。

#### 【解答】

该循环输入一系列单词，进行处理。

此处assert宏的使用纯属多余。因为assert(cin)放在循环体中，只有在输入流cin有效的情况下才会执行到。而在输入流有效的情况下，表达式cin的值为非0，assert不做任何操作。

### 习题6.27

解释下面的循环：

```
string s;
while (cin >> s && s != sought) {} // empty body
assert(cin);
// process s
```

**【解答】**

在打开调试器的情况下，该循环从标准输入读入一系列单词，直到读入的单词与sought相等或遇到文件结束符。如果直到遇到文件结束符，读入的单词也与sought不相等，则assert输出信息并终止程序的执行，否则对读入的单词进行处理。

如果关闭调试器，则assert语句不做任何工作。

# 第7章

## 函 数

### 习题7.1

形参和实参有什么区别？

### 【解答】

形参在函数定义的形参表中进行定义，是一个变量，其作用域为整个函数。而实参出现在函数调用中，是一个表达式。进行函数调用时，用传递给函数的实参对形参进行初始化。

### 习题7.2

下列哪些函数是错误的？为什么？请给出修改意见。

- (a) int f() {  
    string s;  
    // ...  
    return s;  
}
- (b) f2(int i) { /\* ... \*/ }
- (c) int calc(int v1, int v1) /\* ... \*/
- (d) double square(double x) return x \* x;

### 【解答】

(a)是错误的。因为函数头中所定义的返回值类型为int，return语句实际返回的表达式的类型为string，两个类型不同，而string类型又不能隐式转换为int类型。可修改为：

```
string f() {  
    string s;  
    // ...  
    return s;  
}
```

(b)是错误的。因为该函数定义中没有指定返回类型，在标准C++中，定义函数时不指定返回类型是非法的。可修改为：

```
int f2(int i) { /* ... */ }
```

(c)是错误的。缺少括住函数体的左花括号，而且两个形参不应该同名。可修改为：

```
int calc(int v1, int v2) { /* ... */ }
```

(d)是错误的。缺少括住函数体的一对花括号。可修改为：

```
double square(double x) { return x * x; }
```

**习题7.3**

编写一个带有两个int型形参的函数，产生第一个参数的第二个参数次幂的值。编写程序传递两个int数值调用该函数，请检验其结果。

**【解答】**

程序如下：

```
// 7-3.cpp
// 函数power带有两个int型形参，产生第一个参数的第二个参数次幂的值。
// 主函数传递两个int型数值调用power函数
#include <iostream>
using namespace std;

int power(int x, int y) //该函数返回x的y次幂
{
    int result = 1;
    for (int loop = 1; loop <= y; ++loop)
        result *= x;
    return result;
}

int main()
{
    int xval, yval;
    cout << "Enter two integers(the second one should be equal to or bigger than 0):"
         << endl;
    cin >> xval >> yval;
    if (yval < 0){
        cout << "The second integer should be equal to or bigger than 0" << endl;
        return -1;
    }
    cout << "Result of raising " << xval
        << " to the power of " << yval
        << " is " << power(xval, yval) << endl;
    return 0;
}
```

注意，当输入的整数较大时，该power函数的计算结果容易溢出。

**习题7.4**

编写一个函数，返回其形参的绝对值。

**【解答】**

可编写如下abs函数，返回形参x的绝对值：

```
int abs(int x)
{
    return x >= 0 ? x : -x;
```

**习题7.5**

编写一个函数，该函数具有两个形参，分别为int型和指向int型的指针，并返回这两个int值中较大的数值。考虑应将其指针形参定义为什么类型？

**【解答】**

函数代码如下：

```
int getBigger(int x, const int* y)
{
    return x > *y ? x : *y;
}
```

该函数无需修改指针形参所指向的值，因此，为了保护指针形参所指向的值，将指针形参定义为指向const对象的指针。

**习题7.6**

编写函数交换两个int型指针所指向的值，调用并检验该函数，输出交换后的值。

**【解答】**

程序如下：

```
// 7-6.cpp
// 函数swap交换两个int型指针所指向的值。
// 主函数调用swap函数，输出交换后的值
#include <iostream>
using namespace std;

void swap(int *x, int *y) // 该函数交换x和y所指向的值
{
    int temp;
    temp = *x;
    *x = *y;
    *y = temp;
}

int main()
{
    int xval, yval;
    cout << "Enter two integers:" << endl;
    cin >> xval >> yval;
    cout << "Before swapped: "
        << "x = " << xval
        << " y = " << yval << endl;
    swap(&xval, &yval);
    cout << "After swapped: "
        << "x = " << xval
        << " y = " << yval << endl;

    return 0;
}
```

**习题7.7**

解释下面两个形参声明的不同之处：

```
void f(T);
void f(T&);
```

**【解答】**

前者声明的是T类型的形参。在f中修改形参的值不会影响调用f时所传递的实参的值。

后者声明的是T类型的引用形参。在f中修改形参的值实际上相当于修改调用f时所传递的实参的值。

**习题7.8**

举一个例子说明什么时候应该将形参定义为引用类型。再举一个例子说明什么时候不应该将形参定义为引用。

**【解答】**

如果希望通过函数调用修改实参的值，就应该将形参定义为引用类型。

例如，用swap函数交换两数的值。如果不将形参定义为指针类型，则需要直接修改实参的值，应该将形参定义为引用类型：

```
void swap(int &v1, int &v2)
{
    int tmp = v2;
    v2 = v1;
    v1 = tmp;
}
```

除了swap函数这种情况外，为了通过一次函数调用获得多个结果值，也可以使用引用形参。

另外，在向函数传递大型对象时，为了避免复制实参以提高效率，以及使用无法复制的类类型（其复制构造函数为private的类类型）作为形参类型时，也应该将形参定义为引用类型。但这时使用形参的目的是为了避免复制实参，所以应该将形参定义为const引用。

如果不需要通过函数调用修改实参的值，就不应该将形参定义为引用类型。例如，在求绝对值的函数abs（见习题7.4的解答）中，形参就不宜定义为引用类型。

**习题7.9**

将(7.2.2节定义的)find\_val函数的形参表中occurs的声明修改为非引用参数类型，并重新执行这个程序，该函数的行为发生了什么改变？

**【解答】**

调用该函数后，ctr的值将不变（保持调用该函数之前的原值，不再能反映42出现的次数）。因为在函数体中修改的是形参occurs（即ctr的局部副本），对实参ctr不产生影响。

**习题7.10**

下面的程序虽然是合法的，但可用性还不够好，指出并改正该程序的局限。

```
bool test(string& s) { return s.empty(); }
```

**【解答】**

其局限在于：此处使用引用形参的唯一目的是避免复制实参，但没有将形参定义为const引用，从而导致不能使用字符串字面值来调用该函数（因为非const引用形参只能与完全同类型的非const对象关联）。

可更正为：

```
bool test(const string& s) { return s.empty(); }
```

**习题7.11**

何时应将引用形参定义为const对象？如果在需要const引用时，将形参定义为普通引用，则会出

现什么问题？

### 【解答】

如果使用引用形参的唯一目的是避免复制实参，则应将引用形参定义为const对象。

如果在需要const引用时，将形参定义为普通引用，则会导致不能使用右值和const对象，以及需要进行类型转换的对象来调用该函数，从而不必要地限制了该函数的使用。

### 习题7.12

什么时候应使用指针形参？什么时候应使用引用形参？解释两者的优点和缺点。

### 【解答】

当函数需要处理数组且函数体不依赖于数组的长度时应使用指针形参，其他情况下应使用引用形参。

指针形参的优点是可以明确地表示函数所操纵的是指向数组元素的指针，而不是数组本身，而且可以使用任意长度的实参数组来调用函数；其缺点是函数体不能依赖于数组的长度，否则容易造成数组内存的越界访问，从而产生错误的结果或者导致程序崩溃。

引用形参的优点是在函数体中依赖数组的长度是安全的；其缺点是限制了可以传递的实参数组，只能使用长度匹配的实参数组来调用函数。

### 习题7.13

编写程序计算数组元素之和。要求编写函数三次，每次以不同的方法处理数组边界。

### 【解答】

```
// 7-13.cpp
// 计算数组元素之和。
// 三个求和函数以不同的方法处理数组边界
#include <iostream>
using namespace std;

// 传递指向数组第一个和最后一个元素的下一个位置的指针
int sum1(const int *begin, const int *end)
{
    int sum = 0;
    while (begin != end) {
        sum += *begin++;
    }
    return sum;
}

// 传递数组大小
int sum2(const int ia[], size_t size)
{
    int sum = 0;
    for (size_t ix = 0; ix != size; ++ix) {
        sum += ia[ix];
    }
    return sum;
}

// 传递指向数组第一个元素的指针和数组大小
int sum3(int *begin, size_t size)
{
    int sum = 0;
    int *p = begin;
```



```

        while (p != begin + size) {
            sum += *p++;
        }
        return sum;
    }

int main()
{
    int ia[] = { 1, 2, 3, 4 };
    cout << "Summation from sum1(): " << sum1(ia, ia+4) << endl;
    cout << "Summation from sum2(): " << sum2(ia, 4) << endl;
    cout << "Summation from sum3(): " << sum3(ia, 4) << endl;

    return 0;
}

```

**习题7.14**

编写程序求`vector<double>`对象中所有元素之和。

**【解答】**

```

// 7-14.cpp
// 计算vector<double>对象中所有元素之和
#include <iostream>
#include <vector>
using namespace std;

// 传递元素迭代器来处理元素
double vectorSum(vector<double>::iterator begin,
                  vector<double>::iterator end)
{
    double sum = 0.0;
    while (begin != end) {
        sum += *begin++;
    }
    return sum;
}

int main()
{
    vector<double> dvec;

    // 读入vector元素
    cout << "Enter double type elements for vector:(Ctrl+Z to end)" 
         << endl;
    double dval;
    while (cin >> dval)
        dvec.push_back(dval);

    // 求元素之和并输出结果
    cout << "Summation of elements: "
         << vectorSum(dvec.begin(), dvec.end()) << endl;

    return 0;
}

```

**习题7.15**

编写一个主函数main，使用两个值作为实参，并输出它们的和。

**【解答】**

```
// 7-15.cpp
// 主函数main使用两个值作为实参，并输出它们的和
#include <iostream>
using namespace std;

int main(int argc, char **argv)
{
    // 检查命令行参数
    if (argc != 3) {
        cout << "you should use three arguments!" << endl;
        return -1;
    }
    cout << "Summation of " << argv[1]
        << " and " << argv[2] << " is "
        // 使用标准库函数atof将C风格字符串转换为double型数据
        << (atof(argv[1]) + atof(argv[2])) << endl;

    return 0;
}
```

**习题7.16**

编写程序使之可以接受本节介绍的命令行选项，并输出传递给main的实参的值。

**【解答】**

```
// 7-16.cpp
// 接受命令行选项，并输出传递给main的实参的值
#include <iostream>
using namespace std;

int main(int argc, char **argv)
{
    cout << "arguments passed to main(): " << endl;
    for (int i = 0; i != argc; ++i)
        cout << argv[i] << endl;

    return 0;
}
```

**习题7.17**

什么时候返回引用是正确的？而什么时候返回const引用是正确的？

**【解答】**

返回指向在函数调用之前已存在的对象的引用是正确的。

当不希望返回的对象被修改时，返回const引用是正确的。

**习题7.18**

下面函数存在什么潜在的运行时问题？

```
string &processText() {
    string text;
    while (cin >> text) { /* ... */ }
    // ...
    return text;
}
```



**【解答】**

该函数返回了局部对象text的引用。当函数执行完毕时，将释放分配给局部对象的存储空间，这样，对局部对象的引用就会指向不确定的内存，因此该函数会在运行时出错。

**习题7.19**

判断下面程序是否合法；如果合法，解释其功能；如果不合法，更正它并解释原因。

```
int &get(int *arry, int index) { return arry[index]; }
int main() {
    int ia[10];
    for (int i = 0; i != 10; ++i)
        get(ia, i) = 0;
}
```

**【解答】**

该程序段合法。其功能为：将数组ia的各元素赋值为0。

**习题7.20**

将函数factorial重写为迭代函数（即非递归函数）。

**【解答】**

函数如下：

```
int factorial(int val)
{
    int result = 1;
    for (int i = 1; i <= val; ++i)
        result *= i;
    return result;
}
```

**习题7.21**

如果函数factorial的终止条件为：

```
if (val != 0)
```

会出现什么问题？

**【解答】**

会出现这样的问题：如果实参为负数x，则理论上该函数会求得 $x * (x-1) * \dots * (\text{int型能表示的最小负数}) * (\text{int型能表示的最大正数}) * (\text{int型能表示的最大正数}-1) * \dots * 1$ 这样的结果，但实际运行时会因递归函数调用次数过多而发生程序栈溢出，使得程序无法继续执行。

而当实参为负数时，原factorial函数求得的结果应该为1。

**习题7.22**

编写下面函数的原型：

- (a) 函数名为compare，有两个形参，都是名为matrix的类的引用，返回bool类型的值。
- (b) 函数名为change\_val，返回vector<int>类型的迭代器，有两个形参：一个是int型形参，另一个是vector<int>类型的迭代器。

提示：写函数原型时，函数名应当暗示函数的功能。考虑这个提示会如何影响你用的类型？

### 【解答】

函数原型如下：

```
bool compare(matrix&, matrix&);  
vector<int>::iterator change_val(int, vector<int>::iterator);
```

### 习题7.23

给出下面函数声明，判断哪些调用是合法的，哪些是不合法的。对于那些不合法的调用，解释原因。

```
double calc(double);  
int count(const string &, char);  
int sum(vector<int>::iterator, vector<int>::iterator, int);  
vector<int> vec(10);
```

- (a) calc(23.4, 55.1);
- (b) count("abcd", 'a');
- (c) calc(66);
- (d) sum(vec.begin(), vec.end(), 3.8);

### 【解答】

(b)、(c)、(d)合法。

(a)不合法。因为calc函数只有一个形参，调用该函数时却传递了两个实参。

### 习题7.24

如果有的话，指出下面哪些函数声明是错误的？为什么？

- (a) int ff(int a, int b = 0, int c = 0);
- (b) char \*init(int ht = 24, int wd, char bckgrnd);

### 【解答】

(b)是错误的。因为在形参表中，具有默认实参的形参应该出现在形参表的末尾（此处ht应出现在没有指定默认实参的形参wd和bckgrnd的后面）。

### 习题7.25

假设有如下函数声明和调用，指出哪些调用是不合法的？为什么？哪些是合法的但可能不符合程序员的原意？为什么？

```
// declarations  
char *init(int ht, int wd = 80, char bckgrnd = ' ');  
  
(a) init();  
(b) init(24, 10);  
(c) init(14, '**');
```

### 【解答】

(a)不合法。因为调用init函数时必须显式指定至少一个实参。

(c)合法，但可能不符合程序员的原意。因为这里是将char型实参'\*\*'转换为int型再传递给形参wd。

### 习题7.26

用字符's'作为默认实参重写函数make\_plural。利用这个版本的函数输出单词"success"和

"failure"的单数和复数形式。

### 【解答】

```
// 7-26.cpp
// 用字符's'作为默认实参实现函数make_plural。
// 利用这个版本的函数输出单词"success"和"failure" 的单数和复数形式
#include <iostream>
#include <string>
using namespace std;

// 如果ctr不为1则返回word的复数版本
string make_plural(size_t ctr, const string &word,
                     const string &ending = "s")
{
    return (ctr == 1) ? word : word + ending;
}

int main()
{
    cout << "Singular version: " << make_plural(1, "success", "es")
    << "\t\tplural version: "
    << make_plural(0, "success", "es") << endl
    << "Singular version: " << make_plural(1, "failure")
    << "\t\tplural version: "
    << make_plural(0, "failure") << endl;

    return 0;
}
```

### 习题7.27

解释形参、局部变量和静态局部变量的差别。并给出一个有效使用了这三种变量的程序例子。

### 【解答】

从本质上说，三者均属于局部作用域中的变量，其中，局部变量又可区分为普通（非静态）局部变量和静态局部变量。它们的差别在于：

(1) 形参的作用域为整个函数体，而普通（非静态）局部变量和静态局部变量的作用域为：从定义处到包含该变量定义的块的结束处。

(2) 形参由调用函数时所传递的实参初始化；而普通（非静态）局部变量和静态局部变量通常用初始化式进行初始化，且均在程序执行流程第一次经过该对象的定义语句时进行初始化。静态局部变量的初始化在整个程序执行过程中只进行一次。

(3) 形参和普通（非静态）局部变量均属自动变量，在每次调用函数时创建，并在函数结束时撤销；而静态局部变量的生命期却跨越了函数的多次调用，它在创建后直到程序结束时才撤销。

例如，如果需要连续输出 $1 \dots n$ 之间所有数的阶乘，可用如下程序：

```
// 7-27.cpp
// 读入上限值upLmt，输出[1..upLmt]之间所有整数的阶乘
#include <iostream>
using namespace std;

// 用于辅助求阶乘的函数
int fac(int x) // x为形参
{
    static result = 1; // result为静态局部变量
    result *= x;
```

```

        return result;
    }

int main()
{
    int upLmt;           // upLmt为普通(非静态)局部变量
    cout << "Enter value of upper limit:" << endl;
    cin >> upLmt;

    //依次输出[1..upLmt]之间所有整数的阶乘
    for (int i = 1; i <= upLmt; ++i)
        cout << i << "!" = " << fac(i) << endl;

    return 0;
}

```

**习题7.28**

编写函数，使其在第一次调用时返回0，然后再次调用时按顺序产生正整数（即返回其当前的调用次数）。

**【解答】**

```

// 7-28.cpp
// 编写函数，使其在第一次调用时返回0,
// 然后再次调用时按顺序产生正整数（即返回其当前的调用次数）
#include <iostream>
using namespace std;

size_t count_calls()
{
    static size_t ctr = -1; // ctr的生命期将跨越函数的多次调用
    return ++ctr;
}

int main()
{
    for (size_t i = 0; i != 10; ++i)
        cout << count_calls() << endl;

    return 0;
}

```

**习题7.29**

对于下面的声明和定义，你会将哪个放在头文件，哪个放在程序文本文件中呢？为什么？

- (a) inline bool eq(const BigInt&, const BigInt&) {...}
- (b) void putValues(int \*arr, int size);

**【解答】**

二者均应放在头文件中。

(b)是函数声明，适合于放在头文件中。

(a)虽然是一个函数定义，但这是一个内联函数的定义，也应该放在头文件中。因为：内联函数的定义对编译器而言必须是可见的，以便编译器能够在调用点内联展开该函数的代码，这样一来，仅有函数原型是不够的；而且内联函数有可能在程序中定义不止一次，这时必须保证在所有源文件中，其定义是完全相同的。把内联函数的定义放在头文件中，可以确保在调用函数时所使用的定义是相同的，

并且保证在调用点该函数的定义对编译器是可见的。

### 习题7.30

把7.2.2节的函数isShorter改写为内联函数。

#### 【解答】

将一个函数指定为内联函数，只需在函数的返回类型前加上关键字inline，因此函数isShorter可改写如下：

```
// 比较两个字符串的长度
inline bool isShorter(const string &s1, const string &s2)
{
    return s1.size() < s2.size();
}
```

### 习题7.31

编写你自己的Sales\_item类，添加两个公有（public）成员用于读和写Sales\_item对象。这两个成员函数的功能应类似于第1章介绍的输入输出操作符。交易也应类似于那一章所定义的。利用这个类读入并输出一组交易。

#### 【解答】

Sales\_item类的头文件如下：

```
// Sales_item.hpp
// 自定义的Sales_item类的头文件
// 定义Sales_item类,
// 添加两个public成员input和output用于读和写Sales_item对象
#ifndef SALESITEM_H
#define SALESITEM_H
#include <iostream>
#include <string>

class Sales_item {
public:
    // Sales_item对象的操作
    std::istream& input(std::istream& in);
    std::ostream& output(std::ostream& out) const;
    double avg_price() const;
    bool same_isbn(const Sales_item &rhs) const
    {
        return isbn == rhs.isbn;
    }

    // 默认构造函数需要初始化内置类型的数据成员
    Sales_item(): units_sold(0), revenue(0.0) { }

private:
    std::string isbn;
    unsigned units_sold;
    double revenue;
};

#endif
```

Sales\_item类的实现文件（源文件）如下：

```

// Sales_item.cpp
// 自定义的Sales_item类的实现文件(源文件)
#include "Sales_item.hpp"

std::istream& Sales_item::input(std::istream& in)
{
    double price;
    in >> isbn >> units_sold >> price;
    // 检验是否读入成功
    if (in)
        revenue = units_sold * price;
    else { // 读入失败: 将对象复位为默认状态
        units_sold = 0;
        revenue = 0.0;
    }
    return in;
}

std::ostream& Sales_item::output(std::ostream& out) const
{
    out << isbn << "\t" << units_sold << "\t"
       << revenue << "\t" << avg_price();
    return out;
}

double Sales_item::avg_price() const
{
    if (units_sold)
        return revenue / units_sold;
    else
        return 0;
}

```

主程序如下：

```

// 7-31.cpp
// 利用自定义的Sales_item类读入并输出一组交易
#include "Sales_item.hpp"
#include <iostream>
using namespace std;

int main()
{
    Sales_item item;

    // 读入并输出一组交易
    cout << "Enter some transactions(Ctrl+Z to end):"
        << endl;
    while (item.input(cin)) {
        cout << "The transaction readed is:" << endl;
        item.output(cout);
        cout << endl;
    }

    return 0;
}

```

### 习题7.32

编写一个头文件，包含你自己的Sales\_item类。使用通用的C++规则给这个头文件以及任何相关

的文件命名，相关文件用于存储在类外定义的非内联函数。

### 【解答】

头文件可命名为Sales\_item.hpp，相应的实现文件可命名为Sales\_item.cpp，具体代码已在上题解答中给出，此处不再赘述。

### 习题7.33

在Sales\_item类中加入一个成员，用于将两个Sales\_item对象相加。使用修改后的类重新求解第1章给出的平均价格问题。

### 【解答】

在Sales\_item类中增加成员函数add，用于将两个Sales\_item对象相加，为此对习题7.31解答中给出的Sales\_item类进行如下修改：

- 在Sales\_item类的定义体中public部分增加如下声明：

```
Sales_item add(Sales_item& other);
```

- 在Sales\_item类的实现文件中增加如下函数定义：

```
Sales_item Sales_item::add(Sales_item& other)
{
    revenue += other.revenue;
    units_sold += other.units_sold;
    return *this;
}
```

求解平均价格问题的主程序如下：

```
// 7-33.cpp
// 利用自定义的Sales_item类。
// 读入一组交易，输出每本书的销售册数、总收入及平均销售价格
#include "Sales_item.hpp"
#include <iostream>
using namespace std;

int main()
{
    Sales_item total, trans;// 保存总和以及下一交易记录

    cout << "Enter some transactions(Ctrl+Z to end):" << endl;
    if (total.input(cin)) { // 读入第一个交易记录有效
        while (trans.input(cin)) // 读入交易记录有效
            if (total.same_isbn(trans))
                // 新读入交易记录的ISBN与前面的相同则更新total
                total.add(trans);
            else {
                // 新读入交易记录的ISBN与前面的不同
                // 则输出并重置total
                total.output(cout) << endl;
                total = trans;
            }
        // 输出最后一个total
        total.output(cout) << endl;
    }
    else {
        // 无输入数据则提示用户
        cout << "No data?!" << endl;
        return -1;
    }
}
```

```

    }
    return 0;
}

```

**习题7.34**

定义一组名为error的重载函数，使之与下面的调用匹配：

```

int index, upperBound;
char selectVal;
// ...
error("Subscript out of bounds: ", index, upperBound);
error("Division by zero");
error("Invalid selection", selectVal);

```

**【解答】**

这组重载函数可定义如下：

```

void error(const string &s, int index, int upperBound)
{
    cout << s << "index is " << index
        << " and upperBound is " << upperBound << endl;
}

void error(const string &s)
{
    cout << s << endl;
}

void error(const string &s, char selectVal)
{
    cout << s << ":" << selectVal << endl;
}

```

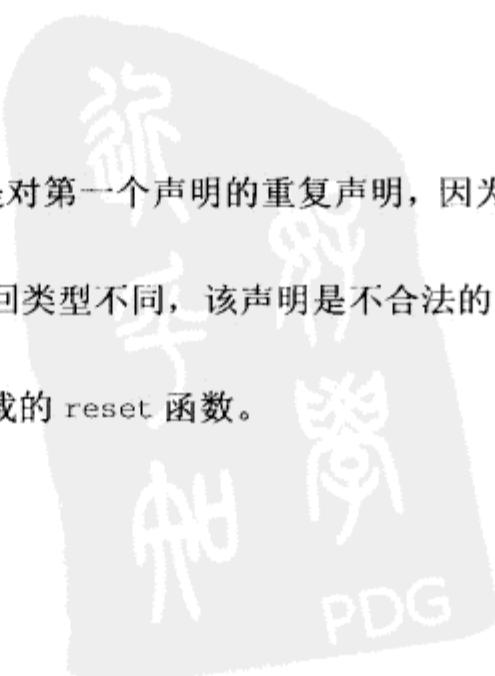
**习题7.35**

下面提供了三组函数声明，解释每组中第二个声明的效果，并指出哪些（如果有的话）是不合法的。

- (a) int calc(int, int);  
int calc(const int, const int);
- (b) int get();  
double get();
- (c) int \*reset(int \*);  
double \*reset(double \*);

**【解答】**

- (a) 第二个声明将形参定义为const，这是对第一个声明的重复声明，因为对于非引用形参而言，是否定义为const没有本质区别。
- (b) 第二个声明与第一个的区别仅在于返回类型不同，该声明是不合法的，因为不能仅仅基于不同的返回类型而实现函数重载。
- (c) 第二个声明的效果是：声明了一个重载的reset函数。



**习题7.36**

什么是候选函数？什么是可行函数？

**【解答】**

候选函数是指在调用点上其声明是可见的且与被调函数同名的函数。

可行函数是指从候选函数中选出的函数，必须满足下列条件：函数的形参数目与该函数调用的实参数目相同；每个实参的类型必须与对应形参的类型匹配，或者可以隐式转换为对应形参的类型。

**习题7.37**

已知本节所列出的 f 函数的声明，判断下面哪些函数调用是合法的。如果有的话，列出每个函数调用的可行函数。如果调用非法，指出是没有函数匹配还是该调用存在二义性。如果调用合法，指出哪个函数是最佳匹配。

- (a) f(2.56, 42);
- (b) f(42);
- (c) f(42, 0);
- (d) f(2.56, 3.14);

**【解答】**

(a) 可行函数是 void f(int, int) 和 void f(double, double = 3.14)。该调用不合法，存在二义性：既可将 2.56 转换为 int 型而调用前者，亦可将 42 转换为 double 型而调用后者。

(b) 可行函数是 void f(int) 和 void f(double, double = 3.14)。该调用合法，最佳匹配函数是 void f(int)。

(c) 可行函数是 void f(int, int) 和 void f(double, double = 3.14)。该调用合法，最佳匹配函数是 void f(int, int)。

(d) 可行函数是 void f(int, int) 和 void f(double, double = 3.14)。该调用合法，最佳匹配函数是 void f(double, double = 3.14)。

**习题7.38**

给出如下声明：

```
void manip(int, int);
double dobj;
```

对于下面两组函数调用，请指出实参上每个转换的优先级等级（7.8.4节）。

- (a) manip('a', 'z');
- (b) manip(55.4, dobj);

**【解答】**

(a) 中转换的优先级为 2，即通过类型提升（实参由 char 型转换为 int 型）实现匹配。

(b) 中转换的优先级为 3，即通过标准转换（实参由 double 型转换为 int 型）实现匹配。

**习题7.39**

解释以下每组声明中的第二个函数声明所造成的影响，并指出哪些不合法（如果有的话）。

- (a) int calc(int, int);
 int calc(const int&, const int&);

- (b) int calc(char\*, char\*);  
int calc(const char\*, const char\*);
- (c) int calc(char\*, char\*);  
int calc(char\* const, char\* const);

**【解答】**

(a)、(b)中第二个声明的效果是：声明了一个重载的calc函数。

(c)中第二个声明是对第一个声明的重复声明。因为当形参以副本传递（即按值传递）时，不能基于形参是否为const来实现函数重载。

**习题7.40**

下面的函数调用是否合法？如果不合法，请解释原因。

```
enum Stat { Fail, Pass };  
void test(Stat);  
test(0);
```

**【解答】**

该函数调用不合法。因为函数的形参为枚举类型Stat，函数调用的实参为int类型。枚举类型对象只能用同一枚举类型的另一对象或一个枚举成员进行初始化，因此不能将int类型的实参值传递给枚举类型的形参。



## 第 8 章

# 标准 IO 库

### 习题8.1

假设os是一个ofstream对象，下面程序做了什么？

```
os << "Goodbye!" << endl;
```

如果os是ostringstream对象呢？或者，os是ifstream对象呢？

#### 【解答】

如果os是一个ofstream对象，则os << "Goodbye!" << endl;将字符串“Goodbye!”及换行符写到os所关联的磁盘文件中。

如果os是一个ostringstream对象，则os << "Goodbye!" << endl;将字符串“Goodbye!”及换行符写到os所关联的字符串流中。

如果os是ifstream对象，则会出现一个编译错误，因为ifstream类中没有定义操作符“<<”。

### 习题8.2

下面的声明是错误的，指出其错误并改正之：

```
ostream print(ostream os);
```

#### 【解答】

错误在于：流类型不能作为函数的形参或返回类型，必须使用流类型的指针或引用。

更正为：

```
ostream& print(ostream& os);
```

### 习题8.3

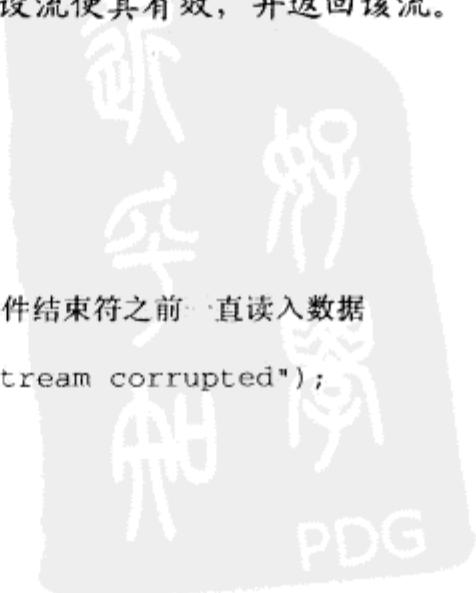
编写一个函数，其唯一的形参和返回值都是istream&类型。该函数应一直读取流直到文件结束符为止，还应将读取的内容输出到标准输出中。最后，重设流使其有效，并返回该流。

#### 【解答】

可编写如下get函数：

```
std::istream& get(std::istream& in)
{
    int ival;

    while (in >> ival, !in.eof()) // 遇到文件结束符之前一直读入数据
        if (in.bad()) // 出现系统级故障
            throw std::runtime_error("IO stream corrupted");
```



```

if (in.fail()) { // 出现可恢复错误
    std::cerr << "bad data, try again"; // 提示用户
    in.clear(); // 恢复流
    in.ignore(200, ' '); // 跳过类型非法的输入项
    continue; // 继续读入数据
}
// 读入正常
std::cout << ival << " ";
in.clear();
return in;
}

```

注意，笔者所使用的编译器不能正确支持`in.clear(iostream::failbit);`语句，所以改用了`in.clear();`语句。此外，恢复流之后需要跳过类型非法的输入项（否则会出现死循环），采用`ignore`函数跳过若干字符（跳过200个字符或遇到空格或文件结束为止），所以调用该函数时输入数据必须以空格作为间隔。

#### 习题8.4

通过以`cin`为实参实现调用来测试上题编写的函数。

#### 【解答】

将习题8.3编写的`get`函数的声明放在头文件`get.hpp`中，其定义放在实现文件`get.cpp`中，则可编写如下程序来调用该函数：

```

// 8-4.cpp
// 主函数以cin为实参调用get函数
#include "get.hpp" // 引入上题定义的get函数
#include <iostream>
using namespace std;

int main()
{
    double dval;

    get(cin);
    cin >> dval; // 重新使用恢复后的流
    cout << dval << endl;

    return 0;
}

```

#### 习题8.5

导致下面的`while`循环终止的原因是什么？

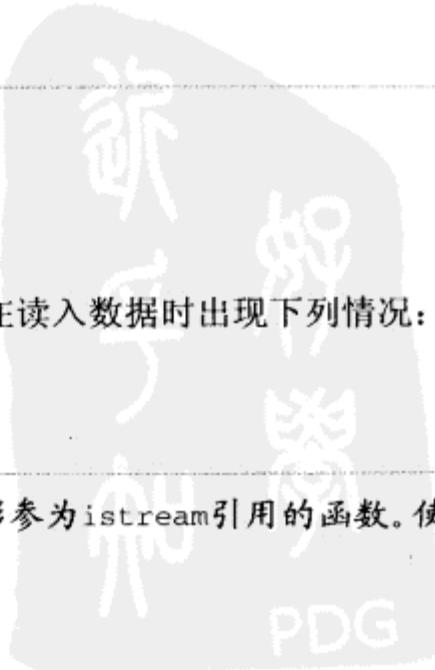
```
while (cin >> i) /* ... */
```

#### 【解答】

导致该循环终止的原因是流对象`cin`进入错误状态，具体包括在读入数据时出现下列情况：系统级故障；读入了无效数据；遇到文件结束符。

#### 习题8.6

由于`ifstream`继承了`istream`，因此可将`ifstream`对象传递给形参为`istream`引用的函数。使用8.2



节第一个习题编写的函数读取已命名的文件。

### 【解答】

程序如下：

```
// 8-6.cpp
// 使用习题8.3中编写的函数读取已命名的文件
#include "get.hpp"           // 引入习题8.3中定义的get函数
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main()
{
    string fileName;
    // 读入文件名
    cout << "Enter file name: " << endl;
    cin >> fileName;
    // 创建ifstream对象并绑定到名为fileName的文件
    ifstream inFile(fileName.c_str());
    if (!inFile) {          // 打开指定文件失败
        cerr << "error: can not open input file: "
            << fileName << endl;
        return -1;
    }
    get(inFile);           // 以inFile为实参调用函数get
    return 0;
}
```

### 习题8.7

本节编写的两个程序，在打开vector容器中存放的任何文件失败时，使用break跳出while循环。重写这两个循环，如果文件无法打开，则输出警告信息，然后从vector中获取下一个文件名继续处理。

### 【解答】

在循环中，如果文件无法打开，则输出警告信息，清除文件流的状态，然后从vector中获取下一个文件名，再将break语句改成continue语句即可。如第二个循环可修改如下：

```
while (it != files.end()) {
    input.open(it->c_str());      // 打开文件
    if (!input) {                  // 打开文件失败
        cerr << "error: can not open file: "
            << *it << endl;
        input.clear();             // 清除文件流的状态
        ++it;                      // 获取下-一个文件
        continue;                  // 继续处理下-一个文件
    }
    // 若打开成功，则读入并处理文件流input
    while (input >> s)           // 处理文件
        process(s);
    input.close();                // 关闭文件
    input.clear();                // 清除文件流的状态
    ++it;                        // 获取下-一个文件
}
```



**习题8.8**

习题8.7的程序可以不用continue语句实现。分别使用或不使用continue语句编写该程序。

**【解答】**

使用continue语句编写的程序见习题8.7的解答。

如果不使用continue语句编写该程序，只需将正常处理放在if语句的else部分即可，完整代码如下：

```
// 8-8.cpp
// 不使用continue语句改写8.4.1节第二个循环：
// 如果文件无法打开，则输出警告信息，
// 然后从vector中获取下一个文件名继续处理
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
using namespace std;

void process(string);

int main()
{
    vector<string> files;
    string fileName, s;

    // 读入vector对象的元素（即欲处理的各个文件的名字）
    cout << "Enter filenames :(Ctrl+Z to end)" << endl;
    while (cin >> fileName)
        files.push_back(fileName);
    ifstream input;
    vector<string>::const_iterator it = files.begin();

    // 处理各个文件
    while (it != files.end()) {
        input.open(it->c_str()); // 打开文件
        if (!input) // 打开文件失败
            cerr << "error: can not open file: "
                << *it << endl;
        input.clear(); // 消除文件流的状态
        ++it;
    }
    else {
        // 若打开成功，则读入并处理文件流input
        while (input >> s) // 处理文件
            process(s);
        input.close(); // 关闭文件
        input.clear(); // 消除文件流的状态
        ++it; // 获取下-一个文件
    }
}
return 0;
}

void process(string s)
{
    cout << s;
}
```

**习题8.9**

编写函数打开文件用于输入，将文件内容读入string类型的vector容器，每一行存储为该容器对象的一个元素。

**【解答】**

可采用全局函数getline读入一行。程序如下：

```
// 8-9.cpp
// 函数fileToVector打开文件用于输入,
// 将文件内容读入string类型的vector容器,
// 每一行存储为该容器对象的一个元素。
// 主函数例示fileToVector函数的使用
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
using namespace std;

int fileToVector(string fileName, vector<string>& svec)
{
    // 创建ifstream对象inFile并绑定到由形参fileName指定的文件
    ifstream inFile(fileName.c_str());
    if (!inFile) // 打开文件失败
        return 1;

    // 将文件内容读入到string类型的vector容器
    // 每一行存储为该容器对象的一个元素
    string s;
    while (getline(inFile, s))
        svec.push_back(s);
    inFile.close(); // 关闭文件
    if (inFile.eof()) // 遇到文件结束符
        return 4;
    if (inFile.bad()) // 发生系统级故障
        return 2;
    if (inFile.fail()) // 读入数据失败
        return 3;
}

int main()
{
    vector<string> svec;
    string fileName, s;

    // 读入文件名
    cout << "Enter filename :" << endl;
    cin >> fileName;

    // 处理文件
    switch (fileToVector(fileName, svec)) {
        case 1:
            cout << "error: can not open file: "
                << fileName << endl;
            return -1;
        case 2:
            cout << "error: system failure " << endl;
            return -1;
        case 3:
            cout << "error: read failure " << endl;
    }
}
```

```

        return -1;
    }

    // 输出vector对象进行检验
    cout << "Vector:" << endl;
    for (vector<string>::iterator iter = svec.begin();
         iter != svec.end(); ++iter)
        cout << *iter << endl;

    return 0;
}

```

**习题8.10**

重写上面的程序，把文件中的每个单词存储为容器的一个元素。

**【解答】**

只需修改读文件的语句，使用文件流对象的>>操作符进行读即可。程序如下：

```

// 8-10.cpp
// 函数fileToVector打开文件用于输入,
// 将文件内容读入string类型的vector容器,
// 每个单词存储为该容器对象的一个元素。
// 主函数例示fileToVector函数的使用
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
using namespace std;

int fileToVector(string fileName, vector<string>& svec)
{
    // 创建ifstream对象 inFile并绑定到由形参fileName指定的文件
    ifstream inFile(fileName.c_str());
    if (!inFile) // 打开文件失败
        return 1;

    // 将文件内容读入到string类型的vector容器
    // 每个单词存储为该容器对象的一个元素
    string s;
    while (inFile >> s) // 读入单词
        svec.push_back(s);
    inFile.close(); // 关闭文件
    if (inFile.eof()) // 遇到文件结束符
        return 4;
    if (inFile.bad()) // 发生系统级故障
        return 2;
    if (inFile.fail()) // 读入数据失败
        return 3;
}

int main()
{
    vector<string> svec;
    string fileName, s;

    // 读入文件名
    cout << "Enter filename :" << endl;
    cin >> fileName;

```

```

// 处理文件
switch (fileToVector(fileName, svec)) {
    case 1:
        cout << "error: can not open file: "
            << fileName << endl;
        return -1;
    case 2:
        cout << "error: system failure " << endl;
        return -1;
    case 3:
        cout << "error: read failure " << endl;
        return -1;
}

// 输出vector对象进行检验
cout << "Vector:" << endl;
for (vector<string>::iterator iter = svec.begin();
     iter != svec.end(); ++iter)
    cout << *iter << endl;

return 0;
}

```

**习题8.11**

对于open\_file函数，请解释为什么在调用open前先调用clear函数。如果忽略这个函数调用，会出现什么问题？如果在open后面调用clear函数，又会怎样？

**【解答】**

在open\_file函数中，因为此时不清楚通过形参in传递进来的流对象的当前状态，所以在调用open前先调用clear函数，将in置为有效状态。

如果忽略这个clear函数调用，则流in的原有状态不会清除，当用它打开另一文件后，有可能导致对另一文件的操作出现问题（例如，如果对in的上一次读操作遇到了文件结束符，则无法对新打开的文件进行有效的读）。

如果在open后面调用clear函数，则即使打开文件失败，in也会被置为有效状态，从而为后续的文件操作带来问题。

**习题8.12**

对于open\_file函数，请解释如果程序执行close函数失败，会产生什么结果？

**【解答】**

如果程序执行close函数失败，则不能用文件流in打开给定的文件。因为open函数会检查流是否已经打开；如果已经打开，则设置内部状态指示发生了错误，接下来使用in的任何尝试都会失败。

**习题8.13**

编写类似open\_file的程序打开文件用于输出。

**【解答】**

可编写如下类似函数：

```
// 打开out绑定到给定文件
ofstream& open_file(ofstream &out, const string &file)
```

```

    {
        out.close(); // 关闭以防它已经是打开的
        out.clear(); // 清除内部状态
        out.open(file.c_str()); // 打开给定文件
        return out;
    }
}

```

**习题8.14**

使用open\_file函数以及8.2节第一个习题编写的程序，打开给定的文件并读取其内容。

**【解答】**

下面的程序调用open\_file函数打开文件，调用get函数读取文件内容：

```

// 8-14.cpp
// 使用open_file函数以及习题8.3中编写的函数get,
// 打开给定的文件并读取其内容
#include "get.hpp" // 引入习题8.3中编写的函数get
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

ifstream& open_file(ifstream &in, const string &file)
{
    in.close(); // 关闭以防它已经是打开的
    in.clear(); // 清除内部状态
    in.open(file.c_str()); // 打开给定文件
    return in;
}

int main()
{
    string fileName, s;
    // 读入文件名
    cout << "Enter filename :" << endl;
    cin >> fileName;

    // 调用open_file函数打开文件
    ifstream inFile;
    if (!open_file(inFile, fileName.c_str())) { // 打开文件失败
        cout << "error: can not open file: "
            << fileName << endl;
        return -1;
    }

    // 调用get函数读取文件
    get(inFile);

    // 关闭文件
    inFile.close();

    return 0;
}

```

**习题8.15**

使用8.2节第一个习题编写的函数输出istringstream对象的内容。

**【解答】**

程序如下：

```
// 8-15.cpp
// 使用习题8.3中编写的get函数，输出istringstream对象的内容
#include "get.hpp" // 引入习题8.3中编写的get函数
#include <iostream>
#include <sstream>
#include <string>
using namespace std;
int main()
{
    // 读入一行文本
    string line;
    cout << "Enter a line of text:" << endl;
    getline(cin, line);

    // 在行末加上一个空格，避免最后一个数据无法输出
    line += " ";

    // 用读入的文本行创建istringstream对象
    istringstream isstr(line);

    // 调用get函数读取istringstream对象并输出其内容
    get(isstr);

    return 0;
}
```

注意，对于从标准输入读入、用于创建`istringstream`对象的文本行，需在末尾加上一个空格。因为`get`函数以文件结束符进行循环控制，加上空格，是为了将最后一个有效数据与文件结束符分隔开，以免最后一个数据无法输出。

**习题8.16**

编写程序将文件中的每一行存储在`vector<string>`容器对象中，然后使用`istringstream`从`vector`中以每次读一个单词的形式读取所存储的行。

**【解答】**

程序如下：

```
// 8-16.cpp
// 将文件中的每一行存储在vector<string>容器对象中，
// 然后使用istringstream
// 从vector里以每次读一个单词的形式读取所存储的行
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
#include <vector>
using namespace std;

int fileToVector(string fileName, vector<string>& svec)
{
    // 创建ifstream对象 inFile 并绑定到由形参fileName指定的文件
    ifstream inFile(fileName.c_str());
    if (!inFile) // 打开文件失败
        return 1;

    string line;
    while (getline(inFile, line))
    {
        istringstream isstr(line);
        string word;
        while (isstr.get(word))
            svec.push_back(word);
    }
}
```

```
// 将文件内容读入到string类型的vector容器
// 每一行存储为该容器对象的一个元素
string s;
while (getline(inFile, s))
    svec.push_back(s);
inFile.close();      // 关闭文件
if (inFile.eof())    // 遇到文件结束符
    return 4;
if (inFile.bad())    // 发生系统级故障
    return 2;
if (inFile.fail())   // 读入数据失败
    return 3;
}

int main()
{
    vector<string> svec;
    string fileName, s;

    // 读入文件名
    cout << "Enter filename :" << endl;
    cin >> fileName;

    // 处理文件
    switch (fileToVector(fileName, svec)) {
        case 1:
            cout << "error: can not to open file: "
                << fileName << endl;
            return -1;
        case 2:
            cout << "error: system failure " << endl;
            return -1;
        case 3:
            cout << "error: read failure " << endl;
            return -1;
    }

    // 使用istringstream从vector里以每次读一个单词的形式读取所存储的行
    string word;
    istringstream isstream;
    for (vector<string>::iterator iter = svec.begin();
        iter != svec.end(); ++iter) {
        // 将vector对象的当前元素复制给istringstream对象
        isstream.str(*iter);
        // 从istringstream对象中读取单词并输出
        while (isstream >> word) {
            cout << word << endl;
        }
        isstream.clear(); // 将istringstream流置为有效状态
    }

    return 0;
}
```

# 第 9 章

## 顺序容器

### 习题9.1

解释下列初始化，指出哪些是错误的，为什么？

```
int ia[7] = { 0, 1, 1, 2, 3, 5, 8 };
string sa[6] = {
    "Fort Sumter", "Manassas", "Perryville",
    "Vicksburg", "Meridian", "Chancellorsville" };
```

- (a) `vector<string> svec(sa, sa+6);`
- (b) `list<int> ilist(ia+4, ia+6);`
- (c) `vector<int> ivec(ia, ia+8);`
- (d) `list<string> slist (sa+6, sa);`

### 【解答】

(c)和(d)是错误的。前者错在用于初始化ivec的一对指针中，ia+8超出了数组的上界，因此会导致运行时出现数组访问越界错误，正确的指针应为ia+7；后者错在用于初始化slist的一对指针的顺序错了，应该是sa在前，sa+6在后，因为它们分别用于标记要复制的第一个元素和停止复制的条件（第二个指针标记要复制的最后一个元素的下一个位置）。

### 习题9.2

创建和初始化一个vector对象有4种方式，为每种方式提供一个例子，并解释每个例子生成的vector对象包含什么值。

### 【解答】

- 分配指定数目的元素，并对这些元素进行值初始化：

```
vector<int> ivec(10); // ivec包含10个0值元素
```

- 分配指定数目的元素，并将这些元素初始化为指定值：

```
vector<int> ivec(10, 1); // ivec包含10个值为1的元素
```

- 将vector对象初始化为一段元素的副本：

```
int ia[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
vector<int> ivec(ia, ia+10); // ivec包含10个元素，值分别为0~9
```

- 将一个vector对象初始化为另一vector对象的副本：

```
vector<int> ivec1(10, 1)
vector<int> ivec2(ivec1);
// ivec2包含10个值为1的元素（与ivec1相同）
```

**习题9.3**

解释复制容器对象的构造函数和使用两个迭代器的构造函数之间的差别。

**【解答】**

差别在于：复制容器对象的构造函数只能将一个容器初始化为另一容器的副本（即复制另一容器的全部元素），这种构造函数要求两个容器是同类型的；使用两个迭代器的构造函数可以将一个容器初始化为另一容器的子序列（即复制另一容器的一个子序列），而且采用这种构造函数不要求两个容器是同类型的。

**习题9.4**

定义一个list对象来存储deque对象，该deque对象存放int型元素。

**【解答】**

```
list< deque<int> > lst;
```

注意，必须用空格隔开两个相邻的>符号，以示这是两个分开的符号，否则，系统会认为>>是单个符号，为右移操作符，从而导致编译时错误。

**习题9.5**

为什么我们不可以使用容器来存储iostream对象？

**【解答】**

因为容器元素类型必须支持赋值操作及复制，而iostream类型不支持赋值和复制。

**习题9.6**

假设有一个名为Foo的类，这个类没有定义默认构造函数，但提供了需要一个int型参数的构造函数。定义一个存放Foo的list对象，该对象有10个元素。

**【解答】**

```
list<Foo> fooList(10, 1); // 各元素均初始化为1
```

**习题9.7**

下面的程序错在哪里？如何改正？

```
list<int> lst1;
list<int>::iterator iter1 = lst1.begin(),
                  iter2 = lst1.end();
while (iter1 < iter2) /* ... */
```

**【解答】**

错误在于while循环的条件表达式iter1 < iter2中使用了< 操作符，因为list容器的迭代器不支持关系操作。可更正为：

```
while (iter1 != iter2) /* ... */
```

**习题9.8**

假设`vec_iter`绑定到`vector`对象的一个元素，该`vector`对象存放`string`类型的元素，请问下面的语句实现什么功能？

```
if (vec_iter->empty()) /* ... */
```

**【解答】**

判断`vec_iter`所指向的那个`vector`元素（`string`对象）是否为空字符串。

**习题9.9**

编写一个循环将`list`容器的元素逆序输出。

**【解答】**

下面的循环将`list`容器的元素逆序输出：

```
list<int> lst1;
list<int>::iterator iter1 = lst1.begin(),
                  iter2 = lst1.end();
while (iter2 != iter1)
    cout << *(--iter2);
```

注意，`*(--iter2)`首先将`iter2`减1，再对减1之后的`iter2`进行解引用。

**习题9.10**

下列迭代器的用法哪些（如果有的话）是错误的？

```
const vector< int > ivec(10);
vector< string > svec(10);
list< int > ilist(10);

(a) vector<int>::iterator it = ivec.begin();
(b) list<int>::iterator it = ilist.begin() + 2;
(c) vector<string>::iterator it = &svec[0];
(d) for (vector<string>::iterator
         it = svec.begin(); it != 0; ++it)
         // ...
```

**【解答】**

(a)是错误的。因为返回的迭代器的类型为`const vector<int>`，不能用来对类型为`vector<int>`的迭代器`it`进行初始化。

(b)是错误的。因为`list`容器的迭代器不支持算术运算`(+)`。

(d)是错误的。因为循环条件中迭代器`it`与0值进行比较，导致运行时内存访问非法的错误，应该将`it!=0`改为`it!=svec.end()`。

**习题9.11**

要标记出有效的迭代器范围，迭代器需满足什么约束？

**【解答】**

如果迭代器`first`和`last`标记出有效的迭代器范围，则必须满足：`first`和`last`指向同一个容器中的元素或超出末端的下一位位置；如果`first`和`last`不相等，则对`first`反复做自增运算必须能够到达

`last`; 即在容器中`last`不能位于`first`之前。

### 习题9.12

编写一个函数，其形参是一对迭代器和一个int型数值，实现在迭代器标记的范围内寻找该int型数值的功能，并返回一个bool结果，以指明是否找到指定数据。

#### 【解答】

如下函数在vector对象中寻找给定的int型数值：

```
// 在迭代器标记的范围内寻找给定的int型数值，返回bool结果指明是否找到
bool findInt(vector<int>::iterator beg,
              vector<int>::iterator end, int ival)
{
    while (beg != end)
        if (*beg == ival) // 找到，则结束循环
            break;
        else
            ++beg;
    if (beg != end)      // 找到（循环提前结束）
        return true;
    else
        return false;
}
```

### 习题9.13

重写程序，查找元素的值，并返回指向找到的元素的迭代器。确保程序在要寻找的元素不存在时也能正确工作。

#### 【解答】

程序如下：

```
// 9-13.cpp
// findInt函数在形参迭代器标记的范围内寻找给定的int型数值，
// 返回指向所找到元素的迭代器。
// 主函数例示findInt函数的使用
#include <iostream>
#include <vector>
using namespace std;

// 在迭代器标记的范围内寻找给定的int型数值，返回指向所找到元素的迭代器
vector<int>::iterator findInt(vector<int>::iterator beg,
                               vector<int>::iterator end, int ival)
{
    while (beg != end)
        if (*beg == ival) // 找到，则结束循环
            break;
        else
            ++beg;

    return beg;          // 若找到，则beg指向所找到的元素；
                         // 否则，beg与end相等
}

int main()
{
    int ia[] = {0, 1, 2, 3, 4, 5, 6};
    vector<int> ivec(ia, ia+7);
```

```

// 读入要找的数据
cout << "Enter a integer:" << endl;
int ival;
cin >> ival;

// 调用findInt函数查找ival
vector<int>::iterator it;
it = findInt(ivec.begin(), ivec.end(), ival);
if (it != ivec.end()) // 找到 (函数返回的迭代器与调用函数的第二个实参不相等)
    cout << ival << " is a element of the vector." << endl;
else
    cout << ival << " isn't a element of the vector." << endl;

return 0;
}

```

注意,为了提高程序的通用性,使得对于不同类型的迭代器及元素类型,都能用同一查找函数进行查找,习题9.12和习题9.13中的findInt函数可以实现为函数模板(见第16章)。例如,可用如下函数模板findValue实现习题9.13中的查找函数findInt的功能:

```

// 函数模板findValue:
// 在迭代器beg和end标记的范围内查找给定的值,返回指向所找到元素的迭代器
template<typename T1, typename T2>
T1 findValue(T1 beg, T1 end, T2 val)
{
    while (beg != end)
        if (*beg == val) // 找到,则结束循环
            break;
        else
            ++beg;
    return beg; // 若找到,则beg指向所找到的元素;否则,beg与end相等
}

```

### 习题9.14

使用迭代器编写程序,从标准输入设备读入若干string对象,并将它们存储在一个vector对象中,然后输出该vector对象中的所有元素。

#### 【解答】

程序如下:

```

// 9-14.cpp
// 从标准输入设备读入若干string 对象,
// 并将它们存储在一个vector对象中,
// 然后输出该vector对象中的所有元素 (使用迭代器)
#include <iostream>
#include <vector>
#include <string>
using namespace std;

int main()
{
    vector<string> svec;
    string str;

    // 读入string对象并存储在vector对象中
    cout << "Enter some strings(Ctrl+Z to end):" << endl;

```

```

    while (cin >> str)
        svec.push_back(str);

    // 输出vector对象的元素
    for (vector<string>::iterator iter = svec.begin();
         iter != svec.end(); ++iter)
        cout << *iter << endl;

    return 0;
}

```

**习题9.15**

用list容器类型重写习题9.14得到的程序，列出改变了容器类型后要做的修改。

**【解答】**

程序如下：

```

// 9-15.cpp
// 从标准输入设备读入若干string对象,
// 并将它们存储在一个list对象中,
// 然后输出该list对象中的所有元素(使用迭代器)
#include <iostream>
#include <list>
#include <string>
using namespace std;

int main()
{
    list<string> slst;
    string str;

    // 读入string对象并存储在vector对象中
    cout << "Enter some strings(Ctrl+Z to end):" << endl;
    while (cin >> str)
        slst.push_back(str);

    // 输出vector对象的元素
    for (list<string>::iterator iter = slst.begin();
         iter != slst.end(); ++iter)
        cout << *iter << endl;

    return 0;
}

```

改变了容器类型后要做的修改有：(1) #include <vector>改为#include <list>; (2)所有的vector改为list; (3)所有的svec改为slst，这个修改不做程序也能正常运行，但从程序的可读性考虑，应该使用slst而不是svec作为list对象的名字。

**习题9.16**

int型的vector容器应该使用什么类型的索引？

**【解答】**

int型的vector容器应该使用的索引类型为vector<int>::size\_type。

**习题9.17**

读取存放string对象的list容器时，应该使用什么类型？

**【解答】**

读取存放string对象的list容器时，可以使用如下迭代器类型：

- `list<string>::iterator` 和 `list<string>::const_iterator` 实现顺序读取。
- `list<string>::reverse_iterator` 和 `list<string>::const_reverse_iterator` 实现逆序读取。

**习题9.18**

编写程序将int型的list容器的所有元素复制到两个deque容器中。list容器的元素如果为偶数，则复制到一个deque容器中；如果为奇数，则复制到另一个deque容器里。

**【解答】**

程序如下：

```
// 9-18.cpp
// 将int型的list容器的所有元素复制到两个deque容器中。
// list容器的元素如果为偶数，则复制到一个deque容器中；
// 如果为奇数，则复制到另一个deque容器里
#include <iostream>
#include <list>
#include <deque>
#include <string>
using namespace std;

int main()
{
    list<int> ilst;
    deque<int> evenDq, oddDq;
    int ival;

    // 读入int对象并存储在list对象中
    cout << "Enter some integers(Ctrl+Z to end):" << endl;
    while (cin >> ival)
        ilst.push_back(ival);

    // 复制list对象的元素至适当的deque对象
    for (list<int>::iterator iter = ilst.begin();
         iter != ilst.end(); ++iter) {
        if (*iter % 2 == 0)
            evenDq.push_back(*iter);
        else
            oddDq.push_back(*iter);
    }

    // 输出deque对象以进行检验
    deque<int>::iterator it;

    // 输出存放偶数的deque对象
    it = evenDq.begin();
    cout << "even deque:" << endl;
    while (it != evenDq.end()) {
        cout << *it << " ";
        ++it;
    }
    cout << endl;

    // 输出存放奇数的deque对象
    it = oddDq.begin();
    cout << "odd deque:" << endl;
    while (it != oddDq.end()) {
        cout << *it << " ";
        ++it;
    }
}
```

```

it = oddDq.begin();
cout << "odd deque:" << endl;
while (it != oddDq.end()) {
    cout << *it << " ";
    ++it;
}
cout << endl;

return 0;
}

```

**习题9.19**

假设iv是一个int型的vector容器，下列程序存在什么错误？如何改正之？

```

vector<int>::iterator mid = iv.begin() + iv.size()/2;
while (vector<int>::iterator iter != mid)
    if (*iter == some_val)
        iv.insert(iter, 2 * some_val);

```

**【解答】**

存在的错误包括：

- 对vector进行插入后会导致迭代器失效，而循环中使用了先前保存的迭代器mid。
- 循环中使用的迭代器iter没有初始化。
- if语句中的条件应该是比较iter所指向的元素（即\*iter）与some\_val是否相等。

可更正为：

```

vector<int>::iterator iter = iv.begin();
while (iter != iv.begin() + iv.size()/2) {
    if (*iter == some_val) {
        iter = iv.insert(iter, 2 * some_val);
        iter += 2; // 使iter指向下一个要处理的原始元素
    }
    else
        ++iter; // 使iter指向下一个要处理的原始元素
}

```

注意，insert函数在迭代器所指元素的前面插入新元素，返回值为指向新插入元素的迭代器，所以将iter加上2才能指向下一个要处理的原始元素。

**习题9.20**

编写程序判断一个vector<int>容器所包含的元素是否与一个list<int>容器的完全相同。

**【解答】**

程序如下：

```

// 9-20.cpp
// 判断一个vector<int>容器所包含的元素
// 是否与一个list<int>容器的完全相同
#include <iostream>
#include <vector>
#include <list>
using namespace std;

int main()

```

```

{
    vector<int> ivec;
    list<int> ilst;
    int ival;

    // 读入int对象并存储在vector对象中
    cout << "Enter some integers for vector(Ctrl+Z to end):"
        << endl;
    while (cin >> ival)
        ivec.push_back(ival);
    cin.clear(); // 使流对象重新置为有效状态

    // 读入int对象并存储在list对象中
    cout << "Enter some integers for list(Ctrl+Z to end):"
        << endl;
    while (cin >> ival)
        ilst.push_back(ival);

    // 比较vector对象和list对象中的对应元素
    vector<int>::iterator vit = ivec.begin();
    list<int>::iterator lit = ilst.begin();
    while (vit != ivec.end() && lit != ilst.end()) {
        if (*vit != *lit) // 对应元素不相等则结束循环
            break;
        ++vit;
        ++lit;
    }

    // 输出比较结果
    if (vit == ivec.end() && lit == ilst.end()) // 所有元素都相等
        cout << "The vector contains the same elements as the list."
            << endl;
    else
        cout << "List and vector contain different elements."
            << endl;
}

return 0;
}

```

注意，因为使用文件结束符作为输入vector元素的结束，所以，读入了所有的vector元素后，流cin处于无效状态，需要将流cin重新置为有效，否则，将无法使用cin继续读入list元素。

### 习题9.21

假设c1和c2都是容器，下列用法给c1和c2的元素类型带来什么约束？

if (c1 < c2)

(如果有的话)对c1和c2的约束又是什么？

#### 【解答】

对c1和c2的元素类型的约束为：类型必须相同且都支持<操作。

对c1和c2的约束为：类型必须相同且都支持<操作。

### 习题9.22

已知容器vec存放了25个元素，那么vec.resize(100)操作实现了什么功能？若再做操作vec.resize(10)，实现的又是什么功能？

**【解答】**

`vec.resize(100)`操作使容器`vec`中包含100个元素：前25个元素保持原值，后75个元素采用值初始化（见3.3.1节）。

若再做操作`vec.resize(10)`，则使容器`vec`中包含10个元素，只保留原来的前10个元素，后面的元素被删除。

**习题9.23**

使用只带有一个长度参数的`resize`操作对元素类型有什么要求（如果有的话）？

**【解答】**

因为只带有一个长度参数的`resize`操作对新添加的元素进行值初始化，所以元素类型如果是类类型，则该类必须显式提供默认构造函数，或者该类不显式提供任何构造函数以便使用编译器自动合成的默认构造函数。

**习题9.24**

编写程序获取`vector`容器的第一个元素。分别使用下标操作符、`front`函数以及`begin`函数实现该功能，并提供空的`vector`容器测试你的程序。

**【解答】**

程序如下：

```
// 9-24.cpp
// 分别使用下标操作符、front函数以及begin函数
// 获取vector容器的第一个元素
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    vector<int> ivec;
    int ival;

    // 读入int对象并存储在vector对象中
    cout << "Enter some integers for vector(Ctrl+Z to end):"
        << endl;
    while (cin >> ival)
        ivec.push_back(ival);

    // 输出vector对象的第一个元素
    if (ivec.empty())
        cout << "No element!" << endl;
    else {
        cout << "first element from subscript: " << ivec[0]
            << endl;
        cout << "first element from front(): " << ivec.front()
            << endl;
        cout << "first element from begin(): " << *ivec.begin()
            << endl;
    }

    return 0;
}
```

如果用空的vector容器测试该程序（即运行程序时不输入任何有效元素），则程序输出“No element!”。

### 习题9.25

需要删除一段元素时，如果val1与val2相等，那么程序会发生什么事情？如果val1和val2中的一个不存在，或两个都不存在，程序又会怎么样？

#### 【解答】

如果val1与val2相等，则不会删除任何元素。

如果val1和val2中的一个不存在，或两个都不存在，则会发生运行时错误。

### 习题9.26

假设有如下ia的定义，将ia复制到一个vector容器和一个list容器中。使用单个迭代器参数版本的erase函数将list容器中的奇数值元素删除，然后将vector容器中的偶数值元素删除。

```
int ia[] = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 55, 89 };
```

#### 【解答】

```
// 9-26.cpp
// 给定定义: int ia[] = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 55, 89 };
// 将ia复制到一个vector容器和一个list 容器中,
// 使用单个迭代器参数版本的erase函数将list容器中的奇数值元素删除掉,
// 然后将vector容器中的偶数值元素删除掉
#include <iostream>
#include <vector>
#include <list>
using namespace std;

int main()
{
    int ia[] = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 55, 89 };
    vector<int> ivec(ia, ia+11); // 将ia复制到vector容器
    list<int> ilst(ia, ia+11); // 将ia复制到list容器

    // 删除list容器中的奇数值元素
    for (list<int>::iterator lit = ilst.begin();
         lit != ilst.end(); ++lit) {
        if (*lit % 2 != 0) { // 迭代器所指向的元素为奇数值
            lit = ilst.erase(lit); // 删除元素
            --lit; // 迭代器回退, 指向被删除元素的前一元素
        }
    }

    // 删除vector容器中的偶数值元素
    for (vector<int>::iterator vit = ivec.begin();
         vit != ivec.end(); ++vit) {
        if (*vit % 2 == 0) { // 迭代器所指向的元素为偶数值
            vit = ivec.erase(vit); // 删除元素
            --vit; // 迭代器回退, 指向被删除元素的前一元素
        }
    }
    return 0;
}
```

注意，在删除元素后迭代器会失效，因此一定要对迭代器重新赋值。另外，`erase`函数返回一个迭代器，指向被删除元素的下一元素。因为在`for`语句头中要对迭代器加1，所以在`if`语句中将迭代器减1，以免遗漏需处理的元素。

### 习题9.27

编写程序处理一个`string`类型的`list`容器。在该容器中寻找一个特殊值，如果找到，则将它删除掉。用`deque`容器重写上述程序。

#### 【解答】

处理`list`容器的程序如下：

```
// 9-27(1).cpp
// 处理一个string类型的list容器。
// 在该容器中寻找一个特殊值，如果找到，则将它删除掉
#include <iostream>
#include <list>
#include <string>
using namespace std;

int main()
{
    list<string> slst;
    string str;

    // 读入list对象
    cout << "Enter some strings(Ctrl+Z to end): " << endl;
    while (cin >> str)
        slst.push_back(str);
    cin.clear(); // 使流重新处于有效状态

    // 读入要寻找的值
    cout << "Enter a string that you want to search: " << endl;
    cin >> str;

    // 处理list对象：删除list对象中与str相同的元素
    for (list<string>::iterator iter = slst.begin();
         iter != slst.end(); ++iter) {
        if (*iter == str) {
            iter = slst.erase(iter); // 删除元素并更新迭代器
            --iter; // 使迭代器指向被删除元素的前一元素
        }
    }

    return 0;
}
```

处理`deque`容器的程序如下：

```
// 9-27(2).cpp
// 处理一个string类型的deque容器。
// 在该容器中寻找一个特殊值，如果找到，则将它删除掉
#include <iostream>
#include <deque>
#include <string>
using namespace std;

int main()
```

```

{
    deque<string> sdq;
    string str;

    // 读入deque对象
    cout << "Enter some strings(Ctrl+Z to end): " << endl;
    while (cin >> str)
        sdq.push_back(str);
    cin.clear(); // 使流重新处于有效状态

    // 读入要寻找的值
    cout << "Enter a string that you want to search: " << endl;
    cin >> str;

    // 处理deque对象：删除deque对象中与str相同的元素
    for (deque<string>::iterator iter = sdq.begin();
         iter != sdq.end(); ++iter) {
        if (*iter == str) {
            iter = sdq.erase(iter); // 删除元素并更新迭代器
            --iter; // 使迭代器指向被删除元素的前一元素
        }
    }

    return 0;
}

```

**习题9.28**

编写程序将一个list容器的所有元素赋值给一个vector容器，其中list容器中存储的是指向C风格字符串的char\*指针，而vector容器的元素则是string类型。

**【解答】**

程序如下：

```

// 9-28.cpp
// 将一个list容器的所有元素赋值给一个vector容器,
// 其中list容器中存储的是指向C风格字符串的char*指针,
// 而vector容器的元素则是string类型
#include <iostream>
#include <list>
#include <vector>
#include <string>
using namespace std;

int main()
{
    char* sa[] = {"Mary", "Tom", "Bob", "Alice"};
    list<char*> slst(sa, sa+4);
    vector<string> svec;
    string str;

    // 将list对象的所有元素赋值给vector对象
    svec.assign(slst.begin(), slst.end());

    // 输出list对象中的元素
    for (list<char*>::iterator lit = slst.begin();
         lit != slst.end(); ++lit) {
        cout << *lit << " ";
    }
    cout << endl;
}

```

```

// 输出vector对象中的元素
for (vector<string>::iterator vit = svec.begin();
     vit != svec.end(); ++vit) {
    cout << *vit << " ";
}
cout << endl;

return 0;
}

```

**习题9.29**

解释vector的容量和长度之间的区别。为什么在连续存储元素的容器中需要支持“容量”的概念？而非连续的容器，如list，则不需要。

**【解答】**

vector的容量(capacity)是指容器在必须分配新存储空间之前可以存储的元素总数，而长度(size)是指容器当前拥有的元素个数。

对于连续存储元素的容器而言，容器中的元素是连续存储的。当在容器内添加一个元素时，如果容器中已经没有空间容纳新的元素，则为了保持元素的连续存储必须重新分配存储空间，用来存放原来的元素以及新添加的元素：首先将存放在旧存储空间中的元素复制到新存储空间里，接着插入新元素，最后撤销旧的存储空间。如果在每次添加新元素时，都要这样分配和撤销内存空间，其性能将会慢得让人无法接受。为了提高性能，连续存储元素的容器实际分配的容量要比当前所需的空间多一些，预留了一些额外的存储区，用于存放新添加的元素，使得不必为每个新元素重新分配容器。所以，在连续存储元素的容器中需要支持“容量”的概念。

而对于不连续存储元素的容器，不存在这样的内存分配问题。例如，在list容器中添加一个元素，标准库只需创建一个新元素，然后将该新元素连接到已存在的链表中，不需要重新分配存储空间，也不必复制任何已存在的元素。所以，这类容器不需要支持“容量”的概念。

**习题9.30**

编写程序研究标准库为vector对象提供的内存分配策略。

**【解答】**

程序如下：

```

// 9-30.cpp
// 研究标准库为vector对象提供的内存分配策略
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    vector<int> ivec;

    // 长度应该为0，容量由实现而定
    cout << "ivec: size: " << ivec.size()
        << " capacity: " << ivec.capacity() << endl;

    // 添加10个元素

```

```

for (int ix = 1; ix != 11; ++ix) {
    ivec.push_back(ix);
    // 长度应该为ix, 容量>=ix, 具体由实现而定
    cout << "ivec: size: " << ivec.size()
        << " capacity: " << ivec.capacity() << endl;
}

// 将现有容量用完
while (ivec.size() != ivec.capacity())
    ivec.push_back(0);

// 添加1个元素
ivec.push_back(0);

// 长度应该为原容量+1, 容量>=原容量+1, 具体由实现而定
cout << "ivec: size: " << ivec.size()
    << " capacity: " << ivec.capacity() << endl;

// 将容量设为至少100
ivec.reserve(100);

// 长度保持不变, 容量>=100, 具体由实现而定
cout << "ivec: size: " << ivec.size()
    << " capacity: " << ivec.capacity() << endl;

// 将当前容量用完
while (ivec.size() != ivec.capacity())
    ivec.push_back(0);

// 容量不变, 长度应该与容量相同
cout << "ivec: size: " << ivec.size()
    << " capacity: " << ivec.capacity() << endl;

// 再增加一个元素
ivec.push_back(42);

// 长度应该加1, 容量>=原容量加1, 具体由实现而定
cout << "ivec: size: " << ivec.size()
    << " capacity: " << ivec.capacity() << endl;
return 0;
}

```

运行用 Microsoft Visual C++ .NET 2003 编译的该程序, 输出为:

```

ivec: size: 0 capacity: 0
ivec: size: 1 capacity: 1
ivec: size: 2 capacity: 2
ivec: size: 3 capacity: 3
ivec: size: 4 capacity: 4
ivec: size: 5 capacity: 6
ivec: size: 6 capacity: 6
ivec: size: 7 capacity: 9
ivec: size: 8 capacity: 9
ivec: size: 9 capacity: 9
ivec: size: 10 capacity: 13
ivec: size: 14 capacity: 19
ivec: size: 14 capacity: 100
ivec: size: 100 capacity: 100
ivec: size: 101 capacity: 150

```

从输出结果可以看出所使用的系统中标准库为vector对象提供的内存分配策略为: 创建空vector

对象时容量为0；当现有容量用完后，重新分配空间时容量的增幅为原容量的1/2（整除2的结果），如果增幅小于1，则取1；`reserve`操作将容量设为指定大小。

### 习题9.31

容器的容量可以比其长度小吗？在初始时或插入元素后，容量是否恰好等于所需要的长度？为什么？

#### 【解答】

容器的容量不能比其长度小。

一般而言，在初始时或插入元素后，容量不会恰好等于所需要的长度（往往是容量大于长度）。因为系统会根据一定的分配策略预留一些额外的存储空间以备容器的增长，从而避免额外的重新分配内存、复制元素、释放内存等操作，提高性能。

### 习题9.32

解释下面程序实现的功能：

```
vector<string> svec;
svec.reserve(1024);
string text_word;
while (cin >> text_word)
    svec.push_back(text_word);
svec.resize(svec.size() + svec.size() / 2);
```

如果该程序读入了256个单词，在调整大小后，该容器的容量可能是多少？如果读入512、1000或1048个单词呢？

#### 【解答】

该程序段的功能为：创建空的`vector`对象后，将其容量设定为1024个元素，然后从标准输入设备读入一系列单词，最后将该`vector`对象的大小调整为所读入单词个数的3/2。

如果该程序读入的单词个数分别为256和512，那么在调整大小后，该容器的容量仍保持为1024，因为此时容器的大小没有超出已分配的内存容量，从而调整大小只改变容器中有效元素的个数，不会改变容量。

如果读入的单词个数为1000，那么在调整大小后，需1500个元素的存储空间，超过了已分配的容量（1024个元素），所以该容器的容量应大于1024，可能为1536（如果标准库采取增幅为当前容量的1/2的重分配策略）。

如果读入的单词个数为1048，则在读入第1025个单词时，因为用完已分配的容量（1024），该容器的容量可能会增长为1536（如果标准库采取增幅为当前容量的1/2的重分配策略）；而在调整大小后，需1572个元素的存储空间，超过了已分配的容量1536，因此会再次重分配，容量可能会再增长768，调整为2304。

### 习题9.33

对于下列程序任务，采用哪种容器（`vector`、`deque`还是`list`）实现最合适？解释选择的理由。如果无法说明采用某种容器比另一种容器更好的原因，请解释为什么无法说明？

- (a) 从一个文件中读入未知数目的单词，以生成英文句子。

(b) 读入固定数目的单词，在输入时将它们按字母顺序插入到容器中。下一章将介绍更适合处理此类问题的关联容器。

(c) 读入未知数目的单词。总是在容器尾部插入新单词，从容器首部删除下一个值。

(d) 从一个文件中读入未知数目的整数。对这些整数排序，然后把它们输出到标准输出设备。

### 【解答】

对于任务(a)，因为单词数量未知，且需要以非确定的顺序处理这些单词，所以采用vector实现最合适，因为vector支持随机访问。

对于任务(b)，采用list实现最合适，因为需要在容器的任意位置插入元素。

任务(c)采用deque实现最合适，因为总是在容器尾部插入新单词，从容器首部删除下一个值。

对于任务(d)，如果一边输入一边排序，则采用list实现最合适。因为在读入时需要在容器的任意位置插入元素（从而实现排序）；如果先读入所有整数，再进行排序，则采用vector最合适，因为进行排序最好有随机访问能力。

### 习题9.34

使用迭代器将string对象中的字符都改为大写字母。

### 【解答】

程序如下：

```
// 9-34.cpp
// 使用迭代器将string对象中的字符都改为大写字母
#include <iostream>
#include <string>
#include <cctype>
using namespace std;

int main()
{
    string str = "This is a example";
    for (string::iterator iter = str.begin();
         iter != str.end(); ++iter)
        *iter = toupper(*iter); // 将字符转换为对应的大写字母
    return 0;
}
```

### 习题9.35

使用迭代器寻找和删除string对象中所有的大写字符。

### 【解答】

程序如下：

```
// 9-35.cpp
// 使用迭代器寻找和删除string对象中所有的人写字符
#include <iostream>
#include <string>
#include <cctype>
using namespace std;

int main()
{
```



```

string str = "This IS A example";
for (string::iterator iter = str.begin();
     iter != str.end(); ++iter) {
    if (isupper(*iter)) { // 元素为大写字母
        str.erase(iter); // 删除该元素
        --iter; // 使迭代器指向被删除元素的前一元素
    }
}
return 0;
}

```

**习题9.36**

编写程序用vector<char>容器初始化string对象。

**【解答】**

```

// 9-36.cpp
// 用vector<char>容器初始化string对象
#include <string>
#include <vector>
using namespace std;

int main()
{
    vector<char> cvec(10, 'a');
    string str(cvec.begin(), cvec.end());
    return 0;
}

```

**习题9.37**

假设希望一次读取一个字符并写入string对象，而且已知需要读入至少100个字符，考虑应该如何提高程序的性能？

**【解答】**

因为string对象中的字符是连续存储的，所以为了提高性能应事先将对象的容量指定为至少100个字符大小，以避免多次进行内存的重新分配。可使用reserve函数达到这一目的（见9.4节）。

**习题9.38**

已知有如下string对象：

"ab2c3d7R4E6"

编写程序寻找该字符串中所有的数字字符，然后再寻找所有的字母字符。以两种版本编写该程序：第一个版本使用find\_first\_of函数，而第二个版本则使用find\_first\_not\_of函数。

**【解答】**

使用find\_first\_of函数的程序：

```

// 9-38(1).cpp
// 使用find_first_of函数，寻找给定字符串中所有的数字字符
#include <iostream>
#include <string>
using namespace std;

int main()

```

```

{
    string str = "ab2c3d7R4E6";
    string numerics("0123456789");
    string letters("abcdefghijklmnopqrstuvwxyz ABCDEFGHIJKLMNOPQRSTUVWXYZ");
    string::size_type pos = 0;

    // 调用find_first_of函数将pos定位到下一个出现数字的位置
    while ((pos = str.find_first_of(numerics, pos))
           != string::npos) {
        cout << "found number at index: " << pos
            << " element is " << str[pos] << endl;
        ++pos; // pos定位到下一字符位置
    }

    pos = 0;
    // 调用find_first_of函数将pos定位到下一个出现字母的位置
    while ((pos = str.find_first_of(letters, pos))
           != string::npos) {
        cout << "found letter at index: " << pos
            << " element is " << str[pos] << endl;
        ++pos; // pos定位到下一字符位置
    }

    return 0;
}

```

使用find\_first\_not\_of函数的程序：

```

// 9-38(2).cpp
// 使用find_first_not_of函数,
// 寻找该字符串中所有的数字字符, 然后再寻找所有的字母字符
#include <iostream>
#include <string>
using namespace std;

int main()
{
    string str = "ab2c3d7R4E6";
    string numerics("0123456789");
    string letters("abcdefghijklmnopqrstuvwxyz ABCDEFGHIJKLMNOPQRSTUVWXYZ");
    string::size_type pos = 0;

    // 调用find_first_not_of函数将pos定位到下一个出现数字的位置
    while ((pos = str.find_first_not_of(letters, pos))
           != string::npos) {
        cout << "found number at index: " << pos
            << " element is " << str[pos] << endl;
        ++pos; // pos定位到下一字符位置
    }

    pos = 0;
    // 调用find_first_not_of函数将pos定位到下一个出现字母的位置
    while ((pos = str.find_first_not_of(numerics, pos))
           != string::npos) {
        cout << "found letter at index: " << pos
            << " element is " << str[pos] << endl;
        ++pos; // pos定位到下一字符位置
    }

    return 0;
}

```

**习题9.39**

已知有如下string对象：

```
string line1 = "We were her pride of 10 she named us:";  
string line2 = "Benjamin, Phoenix, the Prodigal";  
string line3 = "and perspicacious pacific Suzanne";  
string sentence = line1 + ' ' + line2 + ' ' + line3;
```

编写程序计算sentence中有多少个单词，并指出其中最长和最短的单词。如果有多个最长或最短的单词，则将它们全部输出。

**【解答】**

定义代表分隔符集合的string对象separators，使用find\_first\_not\_of和find\_first\_of函数来获取单词的起始位置和结束位置；使用两个vector对象分别存放最长和最短的单词。

程序如下：

```
// 9-39.cpp  
// 对于如下string对象：  
// string line1 = "We were her pride of 10 she named us:";  
// string line2 = "Benjamin, Phoenix, the Prodigal";  
// string line3 = "and perspicacious pacific Suzanne";  
// string sentence = line1 + ' ' + line2 + ' ' + line3;  
// 计算sentence中有多少个单词，并指出其中最长和最短的单词。  
// 如果有多个最长或最短的单词，则将它们全部输出  
#include <iostream>  
#include <string>  
#include <vector>  
using namespace std;  
  
int main()  
{  
    string line1 = "We were her pride of 10 she named us:";  
    string line2 = "Benjamin, Phoenix, the Prodigal";  
    string line3 = "and perspicacious pacific Suzanne";  
    string sentence = line1 + ' ' + line2 + ' ' + line3;  
    string separators("\t,\r\n\f"); //用作分隔符的字符  
    string word;  
    // sentence中最长、最短单词以及下一单词的长度，单词的数目  
    string::size_type maxLen, minLen, wordLen, count = 0;  
    // 存放最长及最短单词的vector容器  
    vector<string> longestWords, shortestWords;  
    // 单词的起始位置和结束位置  
    string::size_type startPos = 0, endPos = 0;  
  
    // 每次循环处理sentence中的一个单词  
    while ((startPos = sentence.find_first_not_of(separators, endPos))  
        != string::npos) {  
        // 找到下一个单词的起始位置  
        ++count;  
  
        // 找下一个单词的结束位置  
        endPos = sentence.find_first_of(separators, startPos);  
  
        if (endPos == string::npos) {  
            // 找不到下一个出现分隔符的位置，即该单词是最后一个单词  
            wordLen = sentence.size() - startPos;  
        }  
        else { // 找到了下一个出现分隔符的位置  
            wordLen = endPos - startPos;  
        }  
        // 将单词存入容器  
        if (wordLen > maxLen) {  
            maxLen = wordLen;  
            longestWords.push_back(sentence.substr(startPos, wordLen));  
        }  
        if (wordLen < minLen) {  
            minLen = wordLen;  
            shortestWords.push_back(sentence.substr(startPos, wordLen));  
        }  
    }  
}
```

```

        wordLen = endPos - startPos;
    }

    word.assign(sentence.begin() + startPos,
                sentence.begin() + startPos + wordLen); // 获取单词

    // 设置下次查找的起始位置
    startPos = sentence.find_first_not_of(separators, endPos);

    if (count == 1) ( // 找到的是第一个单词
        maxLen = minLen = wordLen;
        longestWords.push_back(word);
        shortestWords.push_back(word);
    }
    else {
        if (wordLen > maxLen) ( // 当前单词比目前的最长单词更长
            maxLen = wordLen;
            longestWords.clear(); // 清空存放最长单词的容器
            longestWords.push_back(word);
        }
        else if (wordLen == maxLen) // 当前单词与目前的最长单词等长
            longestWords.push_back(word);

        if (wordLen < minLen) ( // 当前单词比目前的最长单词更短
            minLen = wordLen;
            shortestWords.clear(); // 清空存放最短单词的容器
            shortestWords.push_back(word);
        }
        else if (wordLen == minLen) // 当前单词与目前的最短单词等长
            shortestWords.push_back(word);
    }
}

// 输出单词数目
cout << "word amount: " << count << endl;
vector<string>::iterator iter;

// 输出最长单词
cout << "longest word(s):" << endl;
iter = longestWords.begin();
while (iter != longestWords.end())
    cout << *iter++ << endl;

// 输出最短单词
cout << "shortest word(s):" << endl;
iter = shortestWords.begin();
while (iter != shortestWords.end())
    cout << *iter++ << endl;

return 0;
}

```

**习题9.40**

编写程序接收下列两个string对象：

```
string q1("When lilacs last in the dooryard bloom'd");
string q2("The child is father of the man");
```

然后使用assign和append操作，创建string对象：

```
string sentence("The child is in the dooryard");
```

**【解答】**

程序如下：

```
// 9-40.cpp
// 接收下列两个string对象：
// string q1("When lilacs last in the dooryard bloom'd");
// string q2("The child is father of the man");
// 然后使用assign和append操作，创建string对象：
// string sentence("The child is in the dooryard");
#include <iostream>
#include <string>
using namespace std;

int main()
{
    string q1("When lilacs last in the dooryard bloom'd");
    string q2("The child is father of the man");
    string sentence;

    // 将sentence赋值为"The child is "
    sentence.assign(q2.begin(), q2.begin() + 13);

    // 在sentence末尾添加"in the dooryard"
    sentence.append(q1.substr(q1.find("in"), 15));

    // 输出sentence
    cout << sentence << endl;

    return 0;
}
```

**习题9.41**

已知有如下string对象：

```
string generic1("Dear Ms Daisy:");
string generic2("MrsMsMissPeople");
```

编写程序实现下面函数：

```
string greet(string form, string lastname, string title,
string::size_type pos, int length);
```

该函数使用replace操作实现以下功能：对于字符串form，将其中的Daisy替换为lastname，将其中的Ms替换为字符串title中从pos下标开始的length个字符。例如，语句

```
string lastName("AnnaP");
string salute = greet(generic1, lastName, generic2, 5, 4);
```

将返回字符串

```
Dear Miss AnnaP:
```

**【解答】**

greet函数的代码如下：

```
string greet(string form, string lastname, string title,
```

```

        string::size_type pos, int length)
{
    string::iterator beg, end;
    beg = form.begin() + form.find("Daisy");
    end = beg + 5;
    form.replace(beg, end, lastname);
    beg = form.begin() + form.find("Ms");
    end = beg + 2;
    form.replace(beg, end, title.substr(pos, length));
    return form;
}

```

**习题9.42**

编写程序读入一系列单词，并将它们存储在stack对象中。

**【解答】**

程序如下：

```

// 9-42.cpp
// 读入一系列单词，并将它们存储在stack对象中
#include <iostream>
#include <string>
#include <stack>
using namespace std;

int main()
{
    stack<string> sstack;
    string str;

    // 读入单词并存储在stack对象中
    cout << "Enter some words(Ctrl+Z to end):" << endl;
    while (cin >> str)
        sstack.push(str);

    return 0;
}

```

**习题9.43**

使用stack对象处理带圆括号的表达式。遇到左圆括号时，将其标记下来。然后在遇到右圆括号时，弹出stack对象中这两边括号之间的相关元素（包括左圆括号）。接着在stack对象中压入一个值，用以表明这个用一对圆括号括起来的表达式已经被替换。

**【解答】**

程序如下：

```

// 9-43.cpp
// 使用stack对象处理带圆括号的表达式：
// 遇到左圆括号时，将其标记下来；然后在遇到右圆括号时，
// 弹出stack对象中这两边括号之间的相关元素（包括左圆括号），
// 接着在stack对象中压入一个值'@'，
// 用以表明这个用一对圆括号括起来的表达式已经被替换
#include <iostream>
#include <stack>
#include <string>
using namespace std;

```

```
int main()
{
    stack<char> SEXP; // 处理表达式的stack对象
    string exp;       // 存储表达式的string对象

    // 读入表达式
    cout << "Enter a expression:" << endl;
    cin >> exp;

    // 处理表达式
    string::iterator iter = exp.begin();
    while (iter != exp.end()) {
        if (*iter != ')') // 读到的字符不是右圆括号
            SEXP.push(*iter);
        else {
            // 读到的字符是右圆括号，弹出元素，直到栈顶为左圆括号或栈为空
            while (SEXP.top() != '(' && !SEXP.empty()) {
                SEXP.pop();
            }

            if (SEXP.empty()) // 栈为空
                cout << "parentheses are not matched" << endl;
            else {           // 栈顶为左圆括号
                SEXP.pop();
                SEXP.push('@');
            }
        }
        ++iter;
    }

    return 0;
}
```



# 第 10 章

## 关联容器

### 习题10.1

编写程序读入一系列string和int型数据，将每一组存储在一个pair对象中，然后将这些pair对象存储在vector容器里。

#### 【解答】

```
// 10-1.cpp
// 读入一系列string和int型数据，将每一组存储在一个pair对象中，
// 然后将这些pair对象存储在vector容器里
#include <iostream>
#include <utility> // 使用其中的pair类型
#include <vector>
#include <string>
using namespace std;

int main()
{
    pair<string, int> sипр;
    string str;
    int ival;
    vector< pair<string, int> > pvec; // 存储pair对象的vector对象

    // 读入一系列string和int型数据
    cout << "Enter a string and an integer(Ctrl+Z to end):"
        << endl;
    while (cin >> str >> ival) {
        sипр = make_pair(str, ival); // 生成pair对象
        pvec.push_back(sипр); // 将pair对象存储在vector容器
    }

    return 0;
}
```

注意，存储pair对象的vector对象，其类型为vector< pair<string,int> >，两个>符号之间必须有空格，否则，系统会将其误认为提取操作符>>，出现编译错误。

### 习题10.2

在习题10.1中，至少可使用三种方法创建pair对象。编写三个版本的程序，分别采用不同的方法来创建pair对象。你认为哪一种方法更易于编写和理解，为什么？

#### 【解答】

方法一：在定义pair对象时提供初始化式从而创建pair对象，程序如下：

```
// 10-2(1).cpp
```

```

// 读入一系列string和int型数据，将每一组存储在一个pair对象中，
// 然后将这些pair对象存储在vecotr容器里。
// 采用提供初始化式的方法创建pair对象
#include <iostream>
#include <utility> // 使用其中的pair类型
#include <vector>
#include <string>
using namespace std;

int main()
{
    string str;
    int ival;
    vector< pair<string, int> > pvec;

    // 读入一系列string和int型数据
    cout << "Enter a string and an integer(Ctrl+Z to end):"
        << endl;
    while (cin >> str >> ival) {
        pair<string, int> sipr(str, ival); // 创建pair对象
        pvec.push_back(sipr); // 将pair对象存储在vector容器
    }

    return 0;
}

```

方法二：直接访问pair对象的数据成员从而生成pair对象，程序如下：

```

// 10-2(2).cpp
// 读入一系列string和int型数据，将每一组存储在一个pair对象中，
// 然后将这些pair对象存储在vecotr容器里。
// 采用直接访问数据成员的方法生成pair对象。
#include <iostream>
#include <utility> // 使用其中的pair类型
#include <vector>
#include <string>
using namespace std;

int main()
{
    pair<string, int> sipr;
    string str;
    int ival;
    vector< pair<string, int> > pvec;

    // 读入一系列string和int型数据
    cout << "Enter a string and an integer(Ctrl+Z to end):"
        << endl;
    while (cin >> str >> ival) {
        // 生成pair对象
        sipr.first = str;
        sipr.second = ival;
        pvec.push_back(sipr); // 将pair对象存储在vector容器
    }

    return 0;
}

```

方法三：使用make\_pair函数生成pair对象，见习题10.1解答。

这三种方法在程序编写和理解上没有太大差别。不过笔者个人更喜欢第三种方法，因为使用make\_pair函数似乎可以更明确地表明生成pair对象这一行为。

**习题10.3**

描述关联容器和顺序容器的差别。

**【解答】**

关联容器和顺序容器的本质差别在于：关联容器通过键（key）存储和读取元素，而顺序容器则通过元素在容器中的位置顺序存储和访问元素。

**习题10.4**

举例说明list、vector、deque、map以及set类型分别适用的情况。

**【解答】**

list类型适用于需要在容器的中间位置插入和删除元素的情况。例如，以无序方式读入一系列学生数据，并按学号顺序存储。

vector类型适用于需要随机访问元素的情况。例如，在序号为1...n的一系列人员当中，访问第x个人的信息。

deque类型适用于需要在容器的尾部或头部插入和删除元素的情况。例如，对服务窗口进行管理，先来的顾客先得到服务。

map类型适用于需要键-值对的集合的情况。例如，字典、电话簿的建立和使用。

set类型适用于需要使用键集合的情况。例如，黑名单的建立和使用。

**习题10.5**

定义一个map对象，将单词与一个list对象关联起来，该list对象存储对应的单词可能出现的行号。

**【解答】**

可定义如下的map对象wordLines：

```
map < string, list<int> > wordLines;
```

**习题10.6**

可否定义一个map对象以vector<int>::iterator为键关联int型对象？如果以list<int>::iterator关联int型对象呢？或者，以pair<int, string>关联int？对于每种情况，如果不允许，请解释其原因。

**【解答】**

可以定义map对象以vector<int>::iterator和pair<int, string>为键关联int型对象。

不能定义map对象以list<int>::iterator为键关联int型对象。因为键类型必须支持<操作，而list容器的迭代器类型不支持<操作。

**习题10.7**

对于以int型对象为索引关联vector<int>型对象的map容器，它的mapped\_type、key\_type和value\_type分别是什么？

**【解答】**

mapped\_type、key\_type 和 value\_type 分别是：vector<int>、int 和 pair<const int, vector<int>>。

**习题10.8**

编写一个表达式，使用map的迭代器给其元素赋值。

**【解答】**

假设map的迭代器为iter，要赋给元素的值为val，则可以用`iter->second = val;`语句给map的元素赋值。（注意，键是不能修改的，所以只能给值成员赋值。）

**习题10.9**

编写程序统计并输出所读入的单词出现的次数。

**【解答】**

可以建立一个map对象，保存所读入的单词及其出现次数（以单词为键，对应的值为单词的出现次数）。

对于map容器，如果下标所表示的键在容器中不存在，则添加新元素，利用这一特性可编写程序如下：

```
// 10-9.cpp
// 通过建立map对象保存所读入的单词及其出现次数,
// 统计并输出所读入的单词出现的次数
#include <iostream>
#include <map>
#include <string>
using namespace std;

int main()
{
    map<string, int> wordCount;
    string word;

    // 读入单词并统计其出现次数
    cout << "Enter some words(Ctrl+Z to end):"
        << endl;
    while (cin >> word)
        ++wordCount[word]; // word的出现次数加1

    // 输出结果
    cout << "word\t\t" << "times" << endl;
    for (map<string, int>::iterator iter = wordCount.begin();
         iter != wordCount.end(); ++iter)
        cout << (*iter).first << "\t\t"
            << (*iter).second << endl;
    return 0;
}
```

**习题10.10**

解释下面程序的功能：

```
map<int, int> m;
m[0] = 1;
```

比较上一程序和下面程序的行为：

```
vector<int> v;
```



```
v[0] = 1;
```

**【解答】**

程序段

```
map<int, int> m;
m[0] = 1;
```

的功能是：首先创建一个空的map容器m，然后在m中增加一个键为0的元素，并将其赋值为1。而程序段

```
vector<int> v;
v[0] = 1;
```

将出现运行时错误。因为vector容器v为空，其中下标为0的元素不存在。对于vector容器而言，不能对尚不存在的元素直接赋值，只能使用push\_back、insert等函数增加元素。

**习题10.11**

哪些类型可用作map容器对象的下标？下标操作符返回的又是什么类型？给出一个具体例子说明，即定义一个map对象，指出哪些类型可用作其下标，以及下标操作符返回的类型。

**【解答】**

可用作map容器对象的下标的类型必须是支持<操作的类型。

下标操作符返回的类型为map容器中定义的mapped\_type类型。例如，对于如下定义的对象：

```
map<string, int> wordCount;
```

可用作其下标的类型为string以及C风格字符串类型（包括字面值、数组名和指针），下标操作符返回的类型为int。

**习题10.12**

重写10.3.4节习题的单词统计程序，要求使用insert函数代替下标运算。你认为哪个程序更容易编写和阅读？请解释原因。

**【解答】**

使用insert函数对map对象进行插入操作时，如果试图插入的元素所对应的键已经在容器中，则insert将不做任何操作。而且，带一个键-值pair形参的insert函数将返回一个pair对象，该对象包含一个迭代器和一个bool值，其中迭代器指向map中具有相应键的元素，而bool值则表示是否插入了该元素。

利用上述特点，可编写程序如下：

```
// 10-12.cpp
// 通过建立map对象保存所读入的单词及其出现次数,
// 统计并输出所读入的单词出现的次数。
// 其中使用insert函数代替下标操作
#include <iostream>
#include <map>
#include <utility>
#include <string>
using namespace std;

int main()
{
```



```
map<string, int> wordCount;
string word;

// 读入单词并统计其出现次数
cout << "Enter some words(Ctrl+Z to end):"
    << endl;
while (cin >> word) {
    // 插入元素<word, 1>
    pair<map<string, int>::iterator, bool> ret =
        wordCount.insert(make_pair(word, 1));
    if (!ret.second) // 该单词已在容器中存在
        ++ret.first->second; // 将该单词的出现次数加1
}

// 输出结果
cout << "word\t\t" << "times" << endl;
for (map<string, int>::iterator iter = wordCount.begin();
iter != wordCount.end(); ++iter)
    cout << (*iter).first << "\t\t"
        << (*iter).second << endl;

return 0;
}
```

使用下标操作的程序更简洁，更容易编写和阅读，而insert函数的返回值的使用比较复杂。但使用insert函数可以避免使用下标操作所带来的副作用，即避免对新插入元素的不必要的值初始化，而且可以显式表示元素的插入（下标操作是隐式表示元素的插入），有其优点。

### 习题10.13

假设有`map<string, vector<int>>`类型，指出在该容器中插入一个元素的insert函数应具有的参数类型和返回值类型。

#### 【解答】

参数类型为`pair<const string, vector<int> >`。

返回值类型为`pair<map<string, vector<int> >::iterator, bool>`。

### 习题10.14

map容器的count和find运算有何区别？

#### 【解答】

前者返回map容器中给定键k的出现次数，其返回值只能是0或1；后者在map容器中存在按给定键k索引的元素的情况下，返回指向该元素的迭代器，否则返回超出末端迭代器。

### 习题10.15

你认为count适合用于解决哪一类问题？而find呢？

#### 【解答】

count适合用于解决判断map容器中某键是否存在的问题，而find适合用于解决在map容器中查找指定键对应的元素的问题。

**习题10.16**

定义并初始化一个变量，用来存储调用键为string、值为vector<int>的map对象的find函数的返回结果。

**【解答】**

假设map对象为xmap，要查找的键为k，则可以定义并初始化如下变量iter：

```
map<string, vector<int> >::iterator iter = xmap.find(k);
```

**习题10.17**

上述转换程序（10.3.9节中给出的转换程序）使用了find函数来查找单词：

```
map<string, string>::const_iterator map_it = trans_map.find(word);
```

你认为这个程序为什么要使用find函数？如果使用下标操作符又会怎么样？

**【解答】**

该程序中使用find函数，是为了当文本中出现的单词word是要转换的单词（即以该单词为键的元素存在于map容器trans\_map中）时获取该元素的引用，以便获取对应的转换后的单词。

如果使用下标操作符，则必须先通过使用count函数来判断元素是否存在，否则，当元素不存在时，会创建新的元素并插入到容器中，从而导致单词转换结果出错。

**习题10.18**

定义一个map对象，其元素的键是家族姓氏，而值则是存储该家族孩子名字的vector对象。为这个map容器输入至少6个条目。通过基于家族姓氏的查询检测你的程序，查询应输出该家族所有孩子的名字。

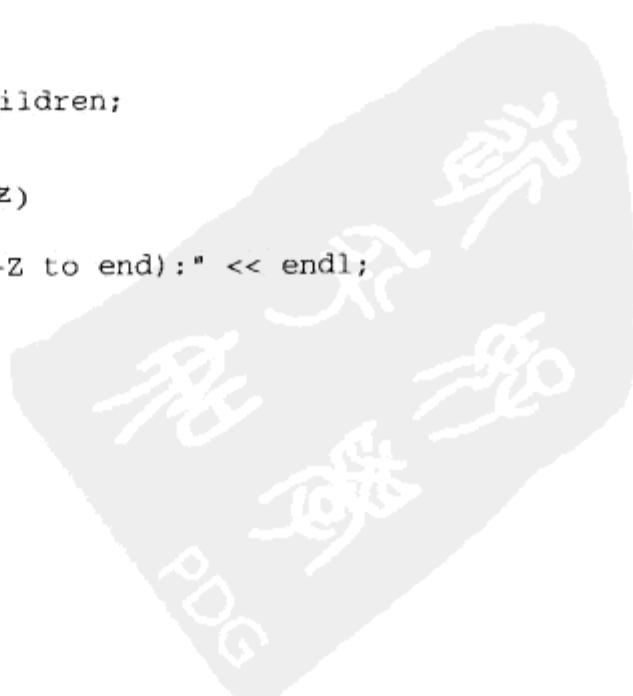
**【解答】**

程序如下：

```
// 10-18.cpp
// 定义一个map对象，其元素的键是家族姓氏，
// 而值则是存储该家族孩子名字的vector对象。
// 进行基于家族姓氏的查询，输出该家族所有孩子的名字
#include <iostream>
#include <map>
#include <vector>
#include <string>
using namespace std;

int main()
{
    map< string, vector<string> > children;
    string surname, childName;

    // 读入条目（家族姓氏及其所有孩子的名字）
    do {
        cout << "Enter surname(Ctrl+Z to end):" << endl;
        cin >> surname;
        if (!cin) // 读入结束
            break;
        vector<string> &childrenForSurname = children[surname];
        do {
            cout << "Enter child name:" << endl;
            cin >> childName;
            childrenForSurname.push_back(childName);
        } while (childName != "quit");
    } while (surname != "quit");
}
```



```

// 插入新条目
vector<string> chd;
pair<map<string, vector<string> >::iterator, bool> ret =
    children.insert(make_pair(surname, chd));

if (!ret.second) // 该家族姓氏已在map容器中存在
    cout << "repeated surname: " << surname << endl;
    continue;
}

cout << "Enter children's name(Ctrl+Z to end):" << endl;
while (cin >> childName) // 读入该家族所有孩子的名字
    ret.first->second.push_back(childName);
    cin.clear(); // 使输入流重新有效
} while (cin);

cin.clear(); // 使输入流重新有效

// 读入要查询的家族
cout << "Enter a surname to search:" << endl;
cin >> surname;

// 根据读入的家族姓氏进行查找
map< string, vector<string> >::iterator iter =
    children.find(surname);

// 输出查询结果
if (iter == children.end()) // 找不到该家族姓氏
    cout << "no this surname: " << surname << endl;
else { // 找到了该家族姓氏
    cout << "children: " << endl;
    // 输出该家族中所有孩子的名字
    vector<string>::iterator it = iter->second.begin();
    while (it != iter->second.end())
        cout << *it++ << endl;
}

return 0;
}

```

### 习题10.19

把习题10.18的map对象再扩展一下，使其vector对象存储pair类型的对象，记录每个孩子的名字和生日。相应地修改程序，测试修改后的程序以检查所编写的map是否正确。

#### 【解答】

map对象的类型修改为`map< string, vector< pair<string, string> > >`。

程序如下：

```

// 10-19.cpp
// 定义一个map对象，其元素的键是家族姓氏，
// 而值则是vector对象，该vector对象存储pair类型的对象，
// pair对象记录每个孩子的名字和生日。
// 进行基于家族姓氏的查询，输出该家族所有孩子的名字和生日
#include <iostream>
#include <utility>
#include <map>
#include <vector>
#include <string>

```

```

using namespace std;

int main()
{
    map< string, vector< pair<string, string> > > children;
    string surname, childName, birthday;

    // 读入条目(家族姓氏及其所有孩子的名字和生日)
    do {
        cout << "Enter surname(Ctrl+Z to end):" << endl;
        cin >> surname;

        if (!cin) // 读入结束
            break;

        // 插入新条目
        vector< pair<string, string> > chd;
        pair<map<string, vector< pair<string, string> > >::iterator, bool> ret =
            children.insert(make_pair(surname, chd));
        if (!ret.second) // 该家族姓氏已在map容器中存在
            cout << "repeated surname: " << surname << endl;
            continue;
    }

    cout << "Enter children's name and birthday(Ctrl+Z to end):"
        << endl;
    // 读入该家族所有孩子的名字和生日
    while (cin >> childName >> birthday) {
        ret.first->second.push_back(make_pair(childName, birthday));
    }
    cin.clear(); // 使输入流重新有效
} while (cin);

cin.clear(); // 使输入流重新有效

// 读入要查询的家族
cout << "Enter a surname to search:" << endl;
cin >> surname;

// 根据读入的家族姓氏进行查找
map< string, vector< pair<string, string> > >::iterator iter;
iter = children.find(surname);

// 输出查询结果
if (iter == children.end()) // 找不到该家族姓氏
    cout << "no this surname: " << surname << endl;
else // 找到了该家族姓氏
    cout << "children\t\tbirthday" << endl;
    // 输出该家族中所有孩子的名字和生日
    vector< pair<string, string> >::iterator it =
        iter->second.begin();
    while (it != iter->second.end()) {
        cout << it->first << "\t\t" << it->second << endl;
        it++;
    }
}

return 0;
}

```

**习题10.20**

列出至少三种可以使用map类型的应用。为每种应用定义map对象，并指出如何插入和读取元素。

**【解答】**

可以使用map类型的应用如下：

- 字典：map对象定义为`map<string, string> dictionary`。
- 电话簿：map对象定义为`map<string, string> telBook`。
- 商品价目表：map对象定义为`map<string, float> priceList`。

可以使用下标操作符或insert函数插入元素，使用find函数读取元素。

**习题10.21**

解释map和set容器的差别，以及它们各自适用的情况。

**【解答】**

map容器和set容器的差别在于：map容器是键-值对的集合，set容器只是键的集合；map类型适用于需要了解键与值的对应的情况，例如，字典（需要了解单词（键）与其解释（值）的对应情况），而set类型适用于只需判断某值是否存在的情况，例如，判断某人的名字是否在黑名单中。

**习题10.22**

解释set和list容器的差别，以及它们各自适用的情况。

**【解答】**

set容器和list容器的主要差别在于：set容器中的元素不能修改，而list容器中的元素无此限制；set容器适用于保存元素值不变的集合，而list容器适用于保存会发生变化的元素。

**习题10.23**

编写程序将被排除的单词存储在vector对象中，而不是存储在set对象中。请指出使用set的好处。

**【解答】**

程序如下：

```
// 10-23.cpp
// 函数restricted_wc：
// 根据形参所指定文件建立单词排除集，
// 将被排除的单词存储在vector对象中，
// 并从标准输入设备读入文本，对不在排除集中的单词进行出现次数统计。
// 主函数示例restricted_wc函数的使用
#include <iostream>
#include <fstream>
#include <map>
#include <vector>
#include <string>
using namespace std;

void restricted_wc(ifstream &removeFile,
                    map<string, int> &wordCount)
{
    vector<string> excluded; // 保存被排除的单词
    string removeWord;
    while (removeFile >> removeWord)
```

```

        excluded.push_back(removeWord);

// 读入文本并对不在排除集中的单词进行出现次数统计
cout << "Enter text(Ctrl+Z to end):" << endl;
string word;
while (cin >> word) {
    bool find = false;

    // 确定该单词是否在排除集中
    vector<string>::iterator iter = excluded.begin();
    while (iter != excluded.end()) {
        if (*iter == word) {
            find = true;
            break;
        }
        ++iter;
    }

    // 如果单词不在排除集中，则进行计数
    if (!find)
        ++wordCount[word];
}

int main()
{
    map<string, int> wordCount;
    string fileName;

    // 读入包含单词排除集的文件的名字并打开相应文件
    cout << "Enter filename:" << endl;
    cin >> fileName;
    ifstream exFile(fileName.c_str());
    if (!exFile) // 打开文件失败
        cout << "error: can not open file:" << fileName << endl;
        return -1;
}

// 调用restricted_wc函数,
// 对输入文本中不在排除集中的单词进行出现次数统计
restricted_wc(exFile, wordCount);

// 输出统计结果
cout << "word\t\t\ttimes" << endl;
map<string, int>::iterator iter = wordCount.begin();
while (iter != wordCount.end()) {
    cout << iter->first << "\t\t" << iter->second << endl;
    iter++;
}

return 0;
}

```

使用set的好处是：可以简单地使用count函数来检查单词是否出现在排除集中（而使用vector则需用循环比较来完成）。

注意，也可以使用标准库中提供的泛型算法find来检查单词是否出现在排除集中，将restricted\_wc函数中的第二个while循环修改如下：

```
while (cin >> word) {
```

```

if (find(excluded.begin(), excluded.end(), word) == excluded.end())
    // 该单词不在排除集中
    ++wordCount[word]; // 进行计数
}

```

**习题10.24**

编写程序通过删除单词尾部的's'生成该单词的非复数版本。同时，建立一个单词排除集，用于识别以's'结尾、但这个结尾的's'又不能删除的单词。例如，放在该排除集中的单词可能有success和class。使用这个排除集编写程序，删除输入单词的复数后缀，而如果输入的是排除集中的单词，则保持该单词不变。

**【解答】**

```

// 10-24.cpp
// 建立一个单词排除集,
// 用于识别以's'结尾、但这个结尾的's'又不能删除的单词。
// 使用这个排除集删除输入单词尾部的's'生成该单词的非复数版本。
// 如果输入的是排除集中的单词，则保持该单词不变
#include <iostream>
#include <set>
#include <string>
using namespace std;

int main()
{
    set<string> excluded;

    // 建立单词排除集
    excluded.insert("success");
    excluded.insert("class");
    //.... 可以在此处继续插入其他以s结尾的单词

    string word;
    cout << "Enter a word(Ctrl+Z to end)" << endl;
    // 读入单词并根据排除集生成该单词的非复数版本
    while (cin >> word) {
        if (!excluded.count(word)) // 该单词未在排除集合中出现
            word.resize(word.size()-1); // 去掉单词末尾的's'
        cout << "non-plural version: " << word << endl;
        cout << "Enter a word(Ctrl+Z to end)" << endl;
    }

    return 0;
}

```

**习题10.25**

定义一个vector容器，存储你在未来6个月里要阅读的书，再定义一个set，用于记录你已经看过的书名。编写程序从vector中为你选择一本没有读过而现在要读的书。当它为你返回选中的书名后，应该将该书名放入记录已读书目的set中。如果实际上你把这本书放在一边没有看，则本程序应该支持从已读书目的set中删除该书的记录。在虚拟的6个月后，输出已读书目和还没有读的书目。

**【解答】**

可通过用户的输入数据来模拟读书及时间的流逝：如果用户希望选择一本书来阅读，则从vector容器中随机选择一本书提供给用户，从vector容器中删除该书并将该书放入set中；如果用户的输入

表明最后没有读这本书，则从set中删除该书并将该书重新放入到vector容器中。

程序如下：

```
// 10-25.cpp
// 定义一个vector容器，存储在未来6个月里要阅读的书的名字。
// 定义一个set，用于记录已经看过的书名。
// 本程序支持从vector中选择一本没有读过而现在要读的书，
// 并将该书名放入记录已读书目的set中，
// 并且支持从已读书目的set中删除该书的记录。
// 在虚拟的6个月后，输出已读书目和还没有读的书目
#include <iostream>
#include <set>
#include <vector>
#include <string>
#include <cstdlib>
#include <ctime>
using namespace std;

int main()
{
    vector<string> books;
    set<string> readedBooks;
    string name;

    // 建立保存未来6个月要阅读的书名的vector对象
    cout << "Enter names for books you'd like to read\\(Ctrl+Z to end):"
        << endl;
    while (cin >> name)
        books.push_back(name);

    cin.clear(); // 使流对象重新有效

    bool timeOver = false;
    string answer, bookName;
    // 用当前系统时间设置随机数发生器种子
    srand((unsigned)time(NULL));

    // 模拟随着时间的流逝而改变读书记录
    while (!timeOver && !books.empty()) {
        // 时间未到6个月且还有书没有读过
        cout << "Would you like to read a book?(Yes/No)" << endl;
        cin >> answer;

        if (answer[0] == 'y' || answer[0] == 'Y') {
            // 在vector中随机选择一本书
            int i = rand() % books.size(); // 产生一个伪随机数
            bookName = books[i];
            cout << "You can read this book: "
                << bookName << endl;
            readedBooks.insert(bookName); // 将该书放入已读集合
            books.erase(books.begin() + i); // 从vector对象中删除该书

            cout << "Did you read it?(Yes/No)" << endl;
            cin >> answer;
            if (answer[0] == 'n' || answer[0] == 'N') {
                // 没有读这本书
                readedBooks.erase(bookName); // 从已读集合中删除该书
                books.push_back(bookName); // 将该书重新放入vector中
            }
        }
    }
}
```

```

        cout << "Time over?(Yes/No)" << endl;
        cin >> answer;
        if (answer[0] == 'y' || answer[0] == 'Y') {
            // 虚拟的6个月结束了
            timeOver = true;
        }
    }
    if (timeOver) { // 虚拟的6个月结束了
        // 输出已读书目
        cout << "books read:" << endl;
        for (set<string>::iterator sit = readedBooks.begin();
             sit != readedBooks.end(); ++sit)
            cout << *sit << endl;
        // 输出还没有读的书目
        cout << "books not read:" << endl;
        for (vector<string>::iterator vit = books.begin();
             vit != books.end(); ++vit)
            cout << *vit << endl;
    }
    else
        cout << "Congratulations! You have read all these books."
            << endl;
}
return 0;
}

```

**习题10.26**

编写程序建立作者及其作品的multimap容器。使用find函数在multimap中查找元素，并调用erase将其删除。当所寻找的元素不存在时，确保你的程序依然能正确执行。

**【解答】**

程序如下：

```

// 10-26.cpp
// 建立作者及其作品的multimap容器。
// 使用find函数在multimap中查找元素，并调用erase将其删除。
// 当所寻找的元素不存在时，程序依然能正确执行
#include <iostream>
#include <map>
#include <string>
using namespace std;

int main()
{
    multimap<string, string> authors;
    string author, work, searchItem;

    // 建立作者及其作品的multimap容器
    do {
        cout << "Enter author name(Ctrl+Z to end):" << endl;
        cin >> author;

        if (!cin)
            break;

        cout << "Enter author's works(Ctrl+Z to end):" << endl;
        while (cin >> work)
            authors.insert(make_pair(author, work));
    }
}

```

```

        cin.clear(); // 读入了一位作者的所有作品后使流对象重新有效
    }while (cin);
    cin.clear(); // 使流对象重新有效

    // 读入要找的作者
    cout << "Who is the author that you want erase:" << endl;
    cin >> searchItem;

    // 找到该作者对应的第一个元素
    multimap<string, string>::iterator iter =
        authors.find(searchItem);
    if (iter != authors.end())
        // 删除该作者的所有作品
        authors.erase(searchItem);
    else
        cout << "Can not find this author!" << endl;

    // 输出multimap对象
    cout << "author\t\twork:" << endl;
    for (iter = authors.begin(); iter != authors.end(); ++iter)
        cout << iter->first << "\t\t" << iter->second << endl;

    return 0;
}

```

对find函数所返回的迭代器进行判断，当该迭代器指向authors中的有效元素时才进行erase操作，从而保证当所寻找的元素不存在时，程序依然能正确执行。

### 习题10.27

重复习题10.26所编写的程序，但这一次要求使用equal\_range函数获取迭代器，然后删除一段范围内的元素。

#### 【解答】

程序如下：

```

// 10-27.cpp
// 建立作者及其作品的multimap容器。
// 使用equal_range函数获取迭代器，然后删除一段范围内的元素。
// 当所寻找的元素不存在时，程序依然能正确执行
#include <iostream>
#include <map>
#include <string>
using namespace std;

int main()
{
    multimap<string, string> authors;
    string author, work, searchItem;

    // 建立作者及其作品的multimap容器
    do {
        cout << "Enter author name(Ctrl+Z to end):" << endl;
        cin >> author;

        if (!cin)
            break;

        cout << "Enter author's works(Ctrl+Z to end):" << endl;
        cin >> work;
        authors.insert({author, work});
    } while (cin);
}

```

```

        while (cin >> work)
            authors.insert(make_pair(author, work));

        cin.clear(); // 读入了一位作者的所有作品后使流对象重新有效
    }while (cin);

    cin.clear(); // 使流对象重新有效

    // 读入要找的作者
    cout << "Who is the author that you want erase:" << endl;
    cin >> searchItem;

    // 确定该作者对应的multimap元素的范围
    typedef multimap<string, string>::iterator itType;
    pair<itType, itType> pos = authors.equal_range(searchItem);

    if (pos.first != pos.second)
        // 删除该作者的所有作品
        authors.erase(pos.first, pos.second);
    else
        cout << "Can not find this author!" << endl;

    // 输出multimap对象
    cout << "author\t\twork:" << endl;
    for (itType iter = authors.begin();
         iter != authors.end(); ++iter)
        cout << iter->first << "\t\t" << iter->second << endl;

    return 0;
}

```

注意，equal\_range函数返回存储一对迭代器的pair对象。如果multimap中存在与参数匹配的元素，则pair对象中的两个迭代器分别指向该键关联的第一个实例和最后一个实例的下一位置；如果找不到与参数匹配的元素，则对象中的两个迭代器都指向此键应插入的位置。

### 习题10.28

沿用习题10.27中的multimap容器，编写程序以下面的格式按姓名首字母的顺序输出作者及其作品：

```

Author Names Beginning with 'A':
Author, book, book, ...
...
Author Names Beginning with 'B':
...

```

### 【解答】

```

// 10-28.cpp
// 建立作者及其作品的multimap容器。
// 以下面的格式按姓名首字母的顺序输出作者及其作品：
// Author Names Beginning with 'A':
// Author, book, book, ...
// ...
// Author Names Beginning with 'B':
// ...
#include <iostream>
#include <map>
#include <string>
using namespace std;

```



```

int main()
{
    multimap<string, string> authors;
    string author, work, searchItem;

    // 建立作者及其作品的multimap容器
    do {
        cout << "Enter author name(Ctrl+Z to end):" << endl;
        cin >> author;

        if (!cin)
            break;

        cout << "Enter author's works(Ctrl+Z to end):" << endl;
        while (cin >> work)
            authors.insert(make_pair(author, work));

        cin.clear(); // 读入了一位作者所有作品后使流对象重新有效
    }while (cin);

    // 输出multimap对象
    typedef multimap<string, string>::iterator itType;
    itType iter = authors.begin();
    if (iter == authors.end()) { // multimap容器为空
        cout << "empty multimap!" << endl;
        return 0;
    }
    string currAuthor, preAuthor; // 记录当前作者及其前一作者
    do {
        currAuthor = iter->first;

        if (preAuthor.empty() || currAuthor[0] != preAuthor[0])
            // 如果出现了首字母不同的作者，则输出该首字母
            cout << "Author Names Beginning with "
                << iter->first[0] << ":" << endl;

        // 输出作者
        cout << currAuthor;

        // 输出该作者所有作品
        pair<itType, itType> pos = authors.equal_range(iter->first);
        while (pos.first != pos.second) {
            cout << ", " << pos.first->second;
            ++pos.first;
        }
        cout << endl; // 输出了一个作者所有作品后，换行

        iter = pos.second; // iter指向下一作者
        preAuthor = currAuthor; // 将当前作者设为前一作者
    }while (iter != authors.end());

    return 0;
}

```

**习题10.29**

解释本节最后一个程序的输出表达式使用的操作数`pos.first->second`的含义。

**【解答】**

其含义为：迭代器对`pos`当中第一个迭代器所指向的`multimap`元素的值。此处的`pos`是一个`pair`

对象，`pos`中存储由函数`equal_range`返回的一对迭代器，其中第一个迭代器指向键`search_item`所关联的第一个实例，第二个迭代器指向键`search_item`所关联的最后一个实例的下一位置。`pos.first`访问这对迭代器中的第一个迭代器，而`pos.first->second`则访问该迭代器所指向的`multimap`元素的值（即键`search_item`所关联的值，程序中`search_item`的值为“Alain de Botton”，所以`pos.first->second`就是作者“Alain de Botton”写的某本书的书名）。

### 习题10.30

`TextQuery`类的成员函数仅使用了前面介绍过的内容。先别查看后面章节，请自己编写这些成员函数。提示：唯一棘手的是`run_query`函数在行号集合`set`为空时应返回什么值？解决方法是构造并返回一个新的（临时）`set`对象。

#### 【解答】

`TextQuery`类的成员函数已在10.6.4节中给出，此处不再赘述。

### 习题10.31

如果没有找到要查询的单词，`main`函数输出什么？

#### 【解答】

（假设要查询的单词为`xstr`）如果没有找到该单词，则`main`函数输出：

```
xstr occurs 0 times
```

### 习题10.32

重新实现文本查询程序，使用`vector`容器代替`set`对象来存储行号。注意，由于行以升序出现，因此只有在当前行号不是`vector`容器对象中的最后一个元素时，才能将新行号添加到`vector`中。这两种实现方法的性能特点和设计特点分别是什么？你觉得哪一种解决方法更好？为什么？

#### 【解答】

主程序如下：

```
// 10-32.cpp
// 文本查询主程序
// 使用以vector容器存储行号的TextQuery类
#include "TextQuery.hpp"

string make_plural(size_t, const string&, const string&);

string open_file(ifstream&, const string&);

void print_results(const vector<TextQuery::line_no>& locs,
                   const string& sought, const TextQuery &file)
{
    // 如果找到单词sought，则输出该单词出现的行数
    typedef vector<TextQuery::line_no> line_nums;
    line_nums::size_type size = locs.size();
    cout << "\n" << sought << " occurs "
        << size << " "
        << make_plural(size, "time", "s") << endl;

    // 输出出现该单词的每一行
    line_nums::const_iterator it = locs.begin();
    for ( ; it != locs.end(); ++it) {
        cout << "\t(line "
            << (*it) << ")"
            << endl;
    }
}
```

```

        // 行号从1开始
        << (*it) + 1 << " ) "
        << file.text_line(*it) << endl;
    }
}

// main函数接受文件名为参数
int main(int argc, char **argv)
{
    // open the file from which user will query words
    ifstream infile;
    if (argc < 2 || !open_file(infile, argv[1])) {
        cerr << "No input file!" << endl;
        return EXIT_FAILURE;
    }

    TextQuery tq;
    tq.read_file(infile); // 建立map容器

    // 循环接受用户的查询要求并输出结果
    while (true) {
        cout << "enter word to look for, or q to quit: ";
        string s;
        cin >> s;
        // 将s变为小写
        string ret;
        for (string::const_iterator it = s.begin();
                                         it != s.end(); ++it) {
            ret += tolower(*it);
        }
        s = ret;

        // 如果用户输入文件结束符或字符'q'及'Q'，则结束循环
        if (!cin || s == "q" || s == "Q") break;

        // 获取出现所查询单词所有行的行号
        vector<TextQuery::line_no> locs = tq.run_query(s);

        // 输出出现次数及所有相关文本行
        print_results(locs, s, tq);
    }
    return 0;
}

```

TextQuery 类的头文件如下：

```

// TextQuery.hpp(for 10-32)
// TextQuery类的头文件
// 使用vector容器存储行号
#ifndef TEXTQUERY_H
#define TEXTQUERY_H
#include <string>
#include <vector>
#include <map>
#include <cctype>
#include <iostream>
#include <fstream>
#include <cstring>
using namespace std;

class TextQuery {
public:

```

```

// 类型别名
typedef string::size_type str_size;
typedef vector<string>::size_type line_no;

// 接口:
// read_file建立给定文件的内部数据结构
void read_file(ifstream &is)
    ( store_file(is); build_map(); )

// run_query查询给定单词并返回该单词所在行的行号集合
vector<line_no> run_query(const string&) const;

// text_line返回输入文件中指定行号对应的行
string text_line(line_no) const;

private:
    //read_file所用的辅助函数
    void store_file(ifstream&); // 存储输入文件
    void build_map(); // 将每个单词与一个行号集合相关联

    //保存输入文件
    vector<string> lines_of_text;

    //将单词与出现该单词的行的行号集合相关联
    map< string, vector<line_no> > word_map;

    //去掉标点并将字母变成小写
    static std::string cleanup_str(const std::string&);

};

#endif

```

TextQuery类的实现文件（源文件）如下：

```

// TextQuery.cpp(for 10-32)
// TextQuery类的实现文件（源文件）
// 使用vector容器存储行号
#include "TextQuery.hpp"
#include <iostream>

string TextQuery::text_line(line_no line) const
{
    if (line < lines_of_text.size())
        return lines_of_text[line];
    throw out_of_range("line number out of range");
}

// 读输入文件，将每行存储为lines_of_text的一个元素
void TextQuery::store_file(ifstream &is)
{
    string textline;
    while (getline(is, textline))
        lines_of_text.push_back(textline);
}

// 在输入vector中找以空白为间隔的单词
// 将单词以及出现该单词的行的行号一起放入word_map
void TextQuery::build_map()
{
    // 处理输入vector中的每一行
    for (line_no line_num = 0;
         line_num != lines_of_text.size();

```

```

        ++line_num)
    {
        // 一次读一个单词
        istringstream line(lines_of_text[line_num]);
        string word;
        while (line >> word) {
            // 去掉标点
            word = cleanup_str(word);
            // 将行号加入到vector容器中
            if (word_map.count(word) == 0) // 单词不在map容器中
                // 下标操作将加入该单词
                word_map[word].push_back(line_num);
            else // 单词已在map容器中
                if (line_num != word_map[word].back())
                    // 行号与vector容器中最后一个元素不相等
                    word_map[word].push_back(line_num);
        }
    }
}

vector<TextQuery::line_no>
TextQuery::run_query(const string &query_word) const
{
    // 注意，为了避免在word_map中加入单词，使用find函数而不用下标操作
    map<string, vector<line_no>>::const_iterator
        loc = word_map.find(query_word);
    if (loc == word_map.end())
        return vector<line_no>(); // 找不到，返回空的vector对象
    else
        // 获取并返回与该单词关联的行号vector对象
        return loc->second;
}

// 去掉标点并将字母变成小写
string TextQuery::cleanup_str(const string &word)
{
    string ret;
    for (string::const_iterator it = word.begin();
                     it != word.end(); ++it) {
        if (!ispunct(*it))
            ret += tolower(*it);
    }
    return ret;
}

```

定义函数make\_plural和open\_file的源文件如下：

```

// functions.cpp
// 定义函数make_plural和open_file
#include <iostream>
#include <string>
using namespace std;

// 如果ctr不为1，返回word的复数版本
string make_plural(size_t ctr, const string &word,
                    const string &ending)
{
    return (ctr == 1) ? word : word + ending;
}

```

```
// 打开输入文件流in并绑定到给定的文件
ifstream& open_file(ifstream &in, const string &file)
{
    in.close();      // close in case it was already open
    in.clear();      // clear any existing errors
    // if the open fails, the stream will be in an invalid state
    in.open(file.c_str()); // open the file we were given
    return in; // condition state is good if open succeeded
}
```

使用vector容器存储行号，在进行行号的插入时，为了不存储重复的行号，需先判断该行号是否已在容器中，再决定是否插入；而使用set容器存储行号，则可以利用set容器所提供的insert函数的特点（如果元素已经存在，insert函数不进行操作），无需判断行号是否已存在，直接利用insert函数插入元素即可，性能相对较高。

从设计上看，vector容器的特点是特别适用于需随机访问元素的情况。而此处对于存储的行号，不需要进行随机访问（只需顺序访问），因而未能充分体现vector容器的优点。而set容器因其插入操作的简单，使得设计更为简洁。因此，此处使用set容器更好。

### 习题10.33

TextQuery::text\_line函数为什么不检查它的参数是否为负数？

#### 【解答】

因为print\_results函数调用text\_line函数时，传递由run\_query函数所获取的set对象中的元素为实参，而由build\_map函数生成的map容器里，set对象中出现的元素（即行号）不可能为负数。



# 第 11 章

## 泛型算法

### 习题11.1

algorithm头文件定义了一个名为count的函数，其功能类似于find。这个函数使用一对迭代器和一个值做参数，返回这个值出现次数的统计结果。编写程序读取一系列int型数据，并将它们存储到vector对象中，然后统计某个指定的值出现了多少次。

#### 【解答】

```
// 11-1.cpp
// 读取一系列int型数据，并将它们存储到vector对象中，
// 然后使用algorithm头文件中定义的名为count的函数，
// 统计某个指定的值出现了多少次
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
    int ival, searchValue;
    vector<int> ivec;

    // 读入int型数据并存储到vector对象中，直至遇到文件结束符
    cout << "Enter some integers(Ctrl+Z to end):" << endl;
    while (cin >> ival)
        ivec.push_back(ival);

    cin.clear(); // 使输入流重新有效

    // 读入欲统计其出现次数的int值
    cout << "Enter an integer you want to search:" << endl;
    cin >> searchValue;

    // 使用count函数统计该值出现的次数并输出结果
    cout << count(ivec.begin(), ivec.end(), searchValue)
        << " elements in the vector have value "
        << searchValue << endl;

    return 0;
}
```

### 习题11.2

重复前面的程序，但是，将读入的值存储到一个string类型的list对象中。

#### 【解答】

只需在上题的程序中将所有的int类型用string类型代替，vector类型用list类型代替，对象名

也做相应修改以提高程序的可读性。

程序如下：

```
// 11-2.cpp
// 读取一系列string型数据，并将它们存储到list对象中，
// 然后使用algorithm头文件中定义的名为count的函数，
// 统计某个指定的字符串出现了多少次
#include <iostream>
#include <list>
#include <string>
#include <algorithm>
using namespace std;

int main()
{
    string sval, searchValue;
    list<string> slst;

    // 读入string型数据并存储到list对象中，直至遇到文件结束符
    cout << "Enter some strings(Ctrl+Z to end):" << endl;
    while (cin >> sval)
        slst.push_back(sval);

    cin.clear(); // 使输入流重新有效

    // 读入欲统计其出现次数的string值
    cout << "Enter a string you want to search:" << endl;
    cin >> searchValue;

    // 使用count函数统计该值出现的次数并输出结果
    cout << count(slst.begin(), slst.end(), searchValue)
        << " elements in the list are \""
        << searchValue << "\"" << endl;

    return 0;
}
```

### 习题11.3

用accumulate统计vector<int>容器对象中的元素之和。

#### 【解答】

程序如下：

```
// 11-3.cpp
// 读取一系列int型数据，并将它们存储到vector对象中，
// 然后使用algorithm头文件中定义的名为accumulate的函数，
// 统计vector对象中的元素之和
#include <iostream>
#include <vector>
#include <numeric>
using namespace std;

int main()
{
    int ival;
    vector<int> ivec;

    // 读入int型数据并存储到vector对象中，直至遇到文件结束符
```

```

cout << "Enter some integers(Ctrl+Z to end):" << endl;
while (cin >> ival)
    ivec.push_back(ival);

// 使用accumulate函数统计vector对象中的元素之和并输出结果
cout << "summation of elements in the vector: "
<< accumulate(ivec.begin(), ivec.end(), 0)
<< endl;

return 0;
}

```

**习题11.4**

假定v是vector<double>类型的对象，则调用accumulate(v.begin(), v.end(), 0)是否有错？如果有的话，错在哪里？

**【解答】**

从函数调用上看没有错误。

调用accumulate函数必须满足的条件包括：容器内的元素类型必须与第三个实参的类型匹配，或者可转换为第三个实参的类型。上述调用中的第三个实参为int类型，而vector对象中的元素的类型为double类型，可以转换为int类型。

但计算的结果不准确。因为将double类型转换为int类型会截去小数部分，得到的求和结果是各元素的整数部分的和，是一个int类型的值，与实际的元素值总和相比会有比较大的误差。

**习题11.5**

对于本节调用find\_first\_of的例程，如果不给it加1，将会如何？

**【解答】**

如果不给it加1，有两种情况：

- 如果不存在同时出现在两个列表中的名字，则循环终止，统计结果正确；
- 如果存在同时出现在两个列表中的名字，则会出现死循环。

**习题11.6**

使用fill\_n编写程序，将一系列int型值设为0。

**【解答】**

如果用vector存放一系列int型值，则可以使用如下程序段将这些int型值都设为0：

```

vector<int> ivec;
ivec.resize(10); // 使该vector容器包含10个元素
fill_n(ivec.begin(), ivec.size(), 0);

```

注意，不能在没有元素的空容器上调用fill\_n函数。

如果用其他类型的容器（如list）存放一系列int型值，也可以编写类似的程序。

**习题11.7**

判断下面的程序是否有错，如果有，请改正之：

(a) vector<int> vec; list<int> lst; int i;

```

while (cin >> i)
    lst.push_back(i);
copy(lst.begin(), lst.end(), vec.begin());
(b) vector<int> vec;
vec.reserve(10);
fill_n(vec.begin(), 10, 0);

```

**【解答】**

(a) 有错。错误在于 vec 是一个空的 vector 容器，而 copy 函数试图将 list 容器 lst 中的元素复制到 vec。更正为：

```
copy(lst.begin(), lst.end(), back_inserter(vec));
```

(b) 有错。错误在于虽然为 vector 对象 vec 分配了内存，但该对象仍是一个空的 vector 对象，而在空的容器上调用 fill\_n 函数是错误的。更正为：

```

vector<int> vec;
vec.resize(10);
fill_n(vec.begin(), 10, 0);

```

**习题11.8**

前面说过，算法不改变它所操纵的容器的大小，为什么使用 back\_inserter 也没有突破这个限制？

**【解答】**

在使用 back\_inserter 时，不是算法直接改变它所操纵的容器的大小，而是算法操纵 back\_inserter 迭代器，迭代器的操作导致容器大小的改变，所以，即使使用 back\_inserter 也没有突破“算法不改变它所操纵的容器的大小”这一限制。

**习题11.9**

编写程序统计长度不小于4的单词，并输出输入序列中不重复的单词。在程序源文件上运行和测试你自己编写的程序。

**【解答】**

因为需要在程序源文件上运行和测试所编写的程序，所以应该从文件读入文本。

程序如下：

```

// 11-9.cpp
// 读入文本文件,
// 统计长度不小于4的单词，并输出输入序列中不重复的单词
#include <iostream>
#include <fstream>
#include <vector>
#include <algorithm>
#include <string>
using namespace std;

// 用于将单词按长度排序的比较函数
bool isShorter(const string &s1, const string &s2)
{
    return s1.size() < s2.size();
}

// 确定给定单词的长度是否不小于4
bool GT4(const string &s)

```



```

{
    return s.size() >= 4;
}

//如果ctr不为1, 返回word的复数版本
string make_plural(size_t ctr, const string &word,
                     const string &ending)
{
    return (ctr == 1) ? word : word + ending;
}

// main函数接受文件名为参数
int main(int argc, char **argv)
{
    // 检查命令行参数个数
    if (argc < 2) {
        cerr << "No input file!" << endl;
        return EXIT_FAILURE;
    }

    // 打开输入文件
    ifstream inFile;
    inFile.open(argv[1]);
    if (!inFile) {
        cerr << "Can not open input file!" << endl;
        return EXIT_FAILURE;
    }

    vector<string> words;
    string word;
    // 读入要分析的输入序列，并存放在vector容器中
    while (inFile >> word)
        words.push_back(word);

    // 对输入排序以便去除重复的单词
    sort(words.begin(), words.end());

    // 使用算法unique对元素重新排序并返回一个迭代器,
    // 表示无重复的单词范围的结束,
    // erase操作使用该迭代器删除输入序列中重复的单词
    words.erase(unique(words.begin(), words.end()), words.end());

    // 将单词按长度排序, 等长的单词按字典顺序排列
    stable_sort(words.begin(), words.end(), isShorter);

    // 计算并输出长度不小于4的单词的数目
    vector<string>::size_type wc =
        count_if (words.begin(), words.end(), GT4);
    cout << wc << " " << make_plural(wc, "word", "s")
        << " 4 characters or longer" << endl;

    // 输出输入序列中不重复的单词
    cout << "unique words: " << endl;
    for (vector<string>::iterator iter = words.begin();
         iter != words.end(); ++iter)
        cout << *iter << " ";
    cout << endl;

    return 0;
}

```

**习题11.10**

标准库定义了一个`find_if`函数。与`find`一样，`find_if`函数带有一对迭代器形参，指定其操作的范围。与`count_if`一样，该函数还带有第三个形参，表明用于检查范围内每个元素的谓词函数。`find_if`返回一个迭代器，指向第一个使谓词函数返回非零值的元素。如果这样的元素不存在，则返回第二个迭代器实参。使用`find_if`函数重写上述例题（11.2.3节中的例题）中统计长度大于6的单词个数的程序部分。

**【解答】**

该程序段可改写如下：

```
// 使用find_if函数统计长度大6的单词的数目
vector<string>::iterator iter = words.begin();
vector<string>::size_type wc = 0;
while ((iter = find_if(iter, words.end(), GT6)) != words.end()) {
    ++wc; // 单词计数加1
    // 找到一个匹配的单词后，iter加1，继续在输入序列的剩余部分中查找
    ++iter;
}
```

**习题11.11**

你认为为什么算法不改变容器的大小？

**【解答】**

为了使得算法能够独立于容器，从而普适性更好，真正成为“泛型”算法，而且算法的概念更为清晰，设计也更为简单。

**习题11.12**

为什么必须使用`erase`，而不是定义一个泛型算法来删除容器中的元素？

**【解答】**

因为算法是不直接修改容器的大小的（见上题解答）。

**习题11.13**

解释三种插入迭代器的区别。

**【解答】**

三种插入迭代器的区别在于插入元素的位置不同：

- `back_inserter`，使用`push_back`实现在容器末端插入。
- `front_inserter`，使用`push_front`实现在容器前端插入。
- `inserter`，使用`insert`实现在容器中指定位置插入。

因此，除了所关联的容器外，`inserter`还带有第二个实参——指向插入起始位置的迭代器。

**习题11.14**

编写程序使用`replace_copy`将一个容器中的序列复制给另一个容器，并将前一个序列中给定的值替换为指定的新值。分别使用`inserter`、`back_inserter`和`front_inserter`实现这个程序。讨论在不同情况下输出序列如何变化。

**【解答】**

假设vector容器ivec中的序列为1 2 3 4 100 5 100，将ivec复制给list容器ilst，并将ivec中值为100的元素替换为0值。

使用inserter实现：

```
replace_copy (ivec.begin(), ivec.end(),
             inserter (ilst, ilst.begin()), 100, 0);
```

在输出序列中的固定位置实现插入，因此ilst中的序列为：

1 2 3 4 0 5 0

使用back\_inserter实现：

```
replace_copy (ivec.begin(), ivec.end(),
             back_inserter (ilst), 100, 0);
```

总是在输出序列的末端实现插入，因此ilst中的序列为：

1 2 3 4 0 5 0

使用front\_inserter实现：

```
replace_copy (ivec.begin(), ivec.end(),
             front_inserter (ilst), 100, 0);
```

总是在输出序列的前端实现插入，因此ilst中的序列为：

0 5 0 4 3 2 1

完整程序如下：

```
// 11-14.cpp
// 使用replace_copy将一个vector容器中的序列复制给一个list容器,
// 并将前一个序列中给定的值替换为指定的新值。
// 分别使用inserter、back_inserter 和front_inserter实现
#include <iostream>
#include <list>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
    int ia[] = {1, 2, 3, 4, 100, 5, 100};
    vector<int> ivec(ia, ia+7);
    list<int> ilst;

    // 将ivec复制给ilst，并将ivec中的值为100的元素替换为0值
    // 使用inserter实现
    replace_copy (ivec.begin(), ivec.end(),
                  inserter (ilst, ilst.begin()), 100, 0);

    // 使用back_inserter实现:
    // replace_copy (ivec.begin(), ivec.end(),
    //               back_inserter (ilst), 100, 0);

    // 使用front_inserter实现:
    // replace_copy (ivec.begin(), ivec.end(),
```

```

//           front_inserter(ilst), 100, 0);

// 输出list容器
cout << "list: " << endl;
for (list<int>::iterator iter = ilst.begin();
     iter != ilst.end(); ++iter)
    cout << *iter << " ";
cout << endl;

return 0;
}

```

注意，可分别去掉程序代码中相应部分的注释，运行程序以验证结果。

### 习题11.15

算法标准库定义了一个名为unique\_copy的函数，其操作与unique类似，唯一的区别在于：前者接受第三个迭代器实参，用于指定复制不重复元素的目标序列。编写程序使用unique\_copy将一个list对象中不重复的元素复制到一个空的vector对象中。

#### 【解答】

```

// 11-15.cpp
// 使用unique_copy算法
// 将一个list对象中不重复的元素复制到一个空的vector对象中
#include <iostream>
#include <list>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
    int ia[] = {1, 2, 3, 4, 100, 5, 100};
    list<int> ilst(ia, ia+7);
    vector<int> ivec;

    // 将list对象ilst中不重复的元素复制到空的vector对象ivec中
    unique_copy(ilst.begin(), ilst.end(), back_inserter(ivec));

    // 输出vector容器
    cout << "vector: " << endl;
    for (vector<int>::iterator iter = ivec.begin();
         iter != ivec.end(); ++iter)
        cout << *iter << " ";
    cout << endl;

    return 0;
}

```

### 习题11.16

重写（11.3.2节第3小节）的程序，使用copy算法将一个文件的内容写到标准输出中。

#### 【解答】

程序如下：

```

// 11-16.cpp
// 使用copy算法将一个文件的内容写到标准输出中
#include <iostream>

```

```

#include <cstdlib>
#include <fstream>
#include <string>
#include <iterator>
#include <algorithm>
using namespace std;

int main()
{
    string fileName;

    // 输入文件名
    cout << "Enter input file name: " << endl;
    cin >> fileName;

    // 打开文件
    ifstream inFile(fileName.c_str());
    if (!inFile) {
        cout << "Can not open file: " << fileName << endl;
        return EXIT_FAILURE;
    }

    // 使用copy算法将文件的内容写到标准输出中
    ostream_iterator<string> outIter(cout, " "); // 以空格分隔数据
    istream_iterator<string> inIter(inFile), eof;
    copy(inIter, eof, outIter);

    // 关闭文件
    inFile.close();

    return 0;
}

```

**习题11.17**

使用一对istream\_iterator对象初始化一个int型的vector对象。

**【解答】**

```

// 定义一对istream_iterator对象
istream_iterator<int> cin_it(cin);
istream_iterator<int> end_of_stream;
// 初始化一个int型的vector对象
vector<int> ivec(cin_it, end_of_stream);

```

**习题11.18**

编写程序使用istream\_iterator对象从标准输入读入一系列整数。使用ostream\_iterator对象将其中的奇数写到一个文件中，并在每个写入的值后面加一个空格。同样使用ostream\_iterator对象将偶数写到第二个文件，每个写入的值都存放在单独的行中。

**【解答】**

```

// 11-18.cpp
// 使用istream_iterator对象从标准输入读入一系列整数。
// 使用ostream_iterator对象将其中的奇数写到一个文件中,
// 并在每个写入的值后面加一个空格。
// 同样使用ostream_iterator 对象将偶数写到第二个文件,
// 每个写入的值都存放在单独的行中
#include <iostream>

```

```

#include <fstream>
#include <iostream>
#include <algorithm>
using namespace std;

int main()
{
    // 打开输出文件流并关联到相应文件
    ofstream oddFile("oddfile.dat");
    ofstream evenFile("evenfile.dat");

    // 打开文件失败
    if (!oddFile || !evenFile) {
        cerr << "Can not open output file!" << endl;
        return EXIT_FAILURE;
    }

    cout << "Enter some integers:(Ctrl+Z to end)" << endl;
    istream_iterator<int> inIter(cin), eof;
    ostream_iterator<int> outOddIter(oddFile, " ");
    ostream_iterator<int> outEvenIter(evenFile, "\n");
    // 读入整数直至遇到eof，将读入数据写入文件流中
    // 并将相应迭代器加1
    while (inIter != eof) {
        if (*inIter % 2 != 0) {// 读入的是奇数
            *outOddIter = *inIter;
            ++outOddIter;
        }
        else { // 读入的是偶数
            *outEvenIter = *inIter;
            ++outEvenIter;
        }
        ++inIter;
    }

    // 关闭文件
    oddFile.close();
    evenFile.close();

    return 0;
}

```

**习题11.19**

编写程序使用reverse\_iterator对象以逆序输出vector容器对象的内容。

**【解答】**

```

// 11-19.cpp
// 使用reverse_iterator对象以逆序输出vector容器对象的内容
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    int ia[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    vector<int> ivec(ia, ia+10);
    vector<int>::reverse_iterator r_iter; // 反向迭代器

    // 逆序输出vector容器对象的元素

```

```

for (r_iter = ivec.rbegin(); r_iter != ivec.rend(); ++r_iter)
    cout << *r_iter << endl;
return 0;
}

```

注意，`reverse_iterator`对象的自增操作使得迭代器指向容器中的前一元素。

### 习题11.20

现在，使用普通的迭代器逆序输出习题11.19中对象的元素。

#### 【解答】

程序如下：

```

// 11-20.cpp
// 使用普通迭代器以逆序输出vector容器对象的内容
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    int ia[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    vector<int> ivec(ia, ia+10);
    vector<int>::iterator iter; // 普通迭代器

    // 逆序输出vector容器对象的元素
    iter = ivec.end();
    --iter; // 使得iter指向最后一个元素
    while (iter >= ivec.begin()) {
        cout << *iter << endl;
        --iter;
    }

    return 0;
}

```

注意，容器的`end`函数返回的是超出末端迭代器，指向最后一个元素的下一位置，所以，在逆序输出元素时要将该迭代器减1，以指向容器中最后一个元素。

### 习题11.21

使用`find`在一个`int`型的`list`中寻找值为0的最后一个元素。

#### 【解答】

给`find`传递一对反向迭代器实参，即可找到与给定值相等的最后一个元素。

程序如下：

```

// 11-21.cpp
// 使用find在一个int型的list中寻找值为0的最后一个元素
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

int main()
{
    int ia[] = {0, 1, 2, 3, 4, 5, 6, 0, 8, 9};
    list<int> ilist(ia, ia+10);
    list<int>::reverse_iterator riter; // 反向迭代器

```

```

// 寻找值为0的最后一个元素
riter = find(ilst.rbegin(), ilst.rend(), 0);

// 输出结果
if (riter != ilst.rend()) // 找到，则riter指向该元素
    // 输出该元素的后一元素以进行检验
    cout << "element after the last 0 : "
        << *(--riter) << endl;
else // 找不到，则riter等于ilist.rend()
    cout << "no element 0" << endl;

return 0;
}

```

**习题11.22**

假设有一个存储了10个元素的vector对象，将其中第3~7个位置上的元素以逆序复制给list对象。

**【解答】**

给copy算法传递一对反向迭代器实参，即可对指定范围的元素进行逆序复制。需要注意的是第二个反向迭代器实参应指向要复制的第一个元素的前一元素（即指向第2个位置上的元素）。

程序如下：

```

// 11-22.cpp
// 对于一个存储了10个元素的vector对象,
// 将其中第3~7个位置上的元素以逆序复制给list对象
#include <iostream>
#include <vector>
#include <list>
#include <algorithm>
using namespace std;

int main()
{
    int ia[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    vector<int> ivec(ia, ia+10);
    list<int> ilst;
    vector<int>::reverse_iterator rstart, rend; // 反向迭代器

    rstart = ivec.rbegin(); // 获得指向尾元素的反向迭代器
    // 获得指向第7个元素的反向迭代器
    for (int cnt1 = 1; cnt1 != 4; ++cnt1)
        ++rstart;

    // 获得指向第2个元素的反向迭代器
    rend = rstart;
    for (int cnt2 = 1; cnt2 != 6; ++cnt2)
        ++rend;

    // 逆序复制元素
    copy(rstart, rend, inserter(ilst, ilst.begin()));

    // 输出list对象，以检验是否为...3
    for (list<int>::iterator iter = ilst.begin();
         iter != ilst.end(); ++iter)
        cout << *iter << endl;
}

return 0;

```

}

**习题11.23**

列出五种迭代器种类及其各自支持的操作。

**【解答】**

迭代器种类	支持的操作
输入迭代器	<code>== != ++ * (只用于读元素, 出现在赋值操作符右边) -&gt;</code>
输出迭代器	<code>++ * (只用于写元素, 出现在赋值操作符左边)</code>
前向迭代器	<code>== != ++ * -&gt;</code>
双向迭代器	<code>== != ++ * -&gt; -- &lt; &lt;= &gt; &gt;= + +=</code>
随机访问迭代器	<code>- (迭代器-整型值 n 迭代器-迭代器) -= []</code>

(见11.3.5节)

**习题11.24**

list容器拥有什么类型的迭代器? 而vector呢?

**【解答】**

list容器拥有双向迭代器, 而vector拥有随机访问迭代器。

**习题11.25**

你认为copy算法需要使用哪种迭代器? 而reverse和unique呢?

**【解答】**

copy算法至少需要使用输入迭代器和输出迭代器; reverse算法至少需要使用双向迭代器; unique算法至少需要使用前向迭代器。

**习题11.26**

解释下列代码错误的原因, 指出哪些错误可以在编译时捕获。

- (a) `string sa[10];`  
`const vector<string> file_names(sa, sa+6);`  
`vector<string>::iterator it = file_names.begin() + 2;`
- (b) `const vector<int> ivec;`  
`fill(ivec.begin(), ivec.end(), ival);`
- (c) `sort(ivec.begin(), ivec.rend());`
- (d) `sort(ivec1.begin(), ivec2.end());`

**【解答】**

(a) 错误在于: file\_names是一个const对象, 因此file\_names.begin()返回的是一个const迭代器, 不能用该迭代器对普通迭代器it进行初始化。

(b) 错误在于: 两个实参迭代器是const迭代器, 不能用来修改容器中的元素, 因此不能用于调用fill算法。

(c) 错误在于：用来指定范围的两个迭代器实参类型不同，前者是 iterator，后者是 reverse\_iterator。

(d) 错误在于：用来指定范围的两个迭代器实参是不同容器的迭代器。

(a)、(b)、(c)中的错误可以在编译时捕获，(d)中的错误不能在编译时捕获。

### 习题11.27

标准库定义了下面的算法：

```
replace(beg, end, old_val, new_val);
replace_if(beg, end, pred, new_val);
replace_copy(beg, end, dest, old_val, new_val);
replace_copy_if(beg, end, dest, pred, new_val);
```

只根据这些函数的名字和形参，描述这些算法的功能。

#### 【解答】

`replace(beg, end, old_val, new_val)`: 将由 beg 和 end 指定的输入范围内值为 old\_val 的元素的值用 new\_val 替换。

`replace_if(beg, end, pred, new_val)`: 将由 beg 和 end 指定的输入范围内使得谓词 pred 为真的元素的值用 new\_val 替换。

`replace_copy(beg, end, dest, old_val, new_val)`: 将由 beg 和 end 指定的输入范围内的元素复制到 dest，并将值为 old\_val 的元素的值用 new\_val 替换。

`replace_copy_if(beg, end, dest, pred, new_val)`: 将由 beg 和 end 指定的输入范围内的元素复制到 dest，并将使得谓词 pred 为真的元素的值用 new\_val 替换。

### 习题11.28

假设 `lst` 是存储了 100 个元素的容器。请解释下面的程序段，并修正你认为的错误。

```
vector<int> vec1;
reverse_copy(lst.begin(), lst.end(), vec1.begin());
```

#### 【解答】

该程序段试图将容器 `lst` 中的元素以逆序复制到 `vector` 容器 `vec1` 中。

其中存在错误：`vec1` 是一个空的 `vector` 容器，尚未分配存储空间，所以该程序段运行时会产生内存访问非法的错误。可将定义 `vec1` 的语句更正为：

```
vector<int> vec1(100);
```

创建包含 100 个元素的 `vector` 容器 `vec1`，其中的每个元素采用值初始化（初始化为 0）。这样一来，就不会存在运行时错误了。

### 习题11.29

用 `list` 容器取代 `vector` 重新实现 11.2.3 节编写的排除重复单词的程序。

#### 【解答】

11.2.3 节编写的程序使用 `vector` 容器存储输入序列，使用标准库提供的泛型算法排除重复的单词。如果用 `list` 容器取代 `vector` 容器解决同一问题，需要注意两点：

(1) list容器上的迭代器是双向迭代器而不是随机访问迭代器，因此，在list容器上不能使用需要随机访问迭代器的泛型算法，如sort算法。

(2) 泛型算法unique虽然可以用在list上，但其性能会降低。

因此，应该使用list容器特有的sort和unique操作来重新实现这一程序。另外，需注意到list容器特有的unique操作与泛型算法unique的差别：list容器的unique操作真正删除了重复的元素，从而无需再使用容器的erase操作来删除重复元素。

程序如下：

```
// 11-29.cpp
// 读入文本文件，存储在list容器中，
// 将其中重复的单词去掉，并输出输入序列中不重复的单词
#include <iostream>
#include <fstream>
#include <list>
#include <algorithm>
#include <string>
using namespace std;

// main函数接受文件名为参数
int main(int argc, char **argv)
{
    // 检查命令行参数个数
    if (argc < 2) {
        cerr << "No input file!" << endl;
        return EXIT_FAILURE;
    }

    // 打开输入文件
    ifstream inFile;
    inFile.open(argv[1]);
    if (!inFile) {
        cerr << "Can not open input file!" << endl;
        return EXIT_FAILURE;
    }
    list<string> words;
    string word;

    // 读入要分析的输入序列，并存放在list容器中
    while (inFile >> word)
        words.push_back(word);

    // 使用list容器的操作sort对输入排序以便去除重复的单词
    words.sort();

    // 使用list容器的操作unique删除输入序列中重复的单词
    words.unique();

    // 输出输入序列中不重复的单词
    cout << "unique words: " << endl;
    for (list<string>::iterator iter = words.begin();
         iter != words.end(); ++iter)
        cout << *iter << " ";
    cout << endl;

    return 0;
}
```

# 第 12 章

## 类

### 习题12.1

编写一个名为Person的类，表示人的名字和地址。使用string来保存每个元素。

#### 【解答】

```
class Person{  
private:  
    std::string name;  
    std::string address;  
};
```

### 习题12.2

为Person提供一个接受两个string参数的构造函数。

#### 【解答】

在习题12.1定义的Person类中增加一个构造函数，类定义如下：

```
class Person{  
public:  
    Person(const std::string &nm, const std::string &addr):  
        name(nm), address(addr)  
    {}  
private:  
    std::string name;  
    std::string address;  
};
```

### 习题12.3

提供返回名字和地址的操作。这些函数应为const吗？解释你的选择。

#### 【解答】

在习题12.2定义的Person类中增加返回名字和地址的成员函数getName和getAddress，类定义如下：

```
class Person{  
public:  
    Person(const std::string &nm, const std::string &addr):  
        name(nm), address(addr)  
    {}  
  
    std::string getName() const  
    {  
        return name;  
    }  
};
```

```

    }

    std::string getAddress() const
    {
        return address;
    }

private:
    std::string name;
    std::string address;
};

```

成员函数getName和getAddress不应该修改数据成员的值，因此应设计为const以保证这一点。

### 习题12.4

指明Person的那个成员应声明为public，哪个成员应声明为private。解释你的选择。

#### 【解答】

将Person类的数据成员name、address声明为private以实现信息隐藏，将成员函数getName、getAddress声明为public以提供外界使用的接口，外界通过该接口访问Person类的数据成员。而构造函数通常声明为public以便创建Person类的对象。

### 习题12.5

C++类支持哪些访问标号？在每个访问标号之后应定义哪种成员？如果有的话，在类的定义中，一个访问标号可以出现在何处以及可出现多少次？约束条件是什么？

#### 【解答】

C++类支持的访问标号包括public、private和protected。

在public之后定义的成员称为公有成员，可以由程序的所有部分访问；在private之后定义的成员称为私有成员，只能由本类（的成员函数）访问；在protected之后定义的成员称为受保护成员，只能由本类及本类的后代类访问（见15.2.2节）。

在类的定义中，一个访问标号可以出现在任意成员定义之前且出现的次数没有限制。

约束条件是：每个访问标号指定了随后的成员定义的访问级别，这个指定的访问级别持续有效，直至遇到下一个访问标号或看到类定义体的右花括号为止。

### 习题12.6

用class关键字定义的类和用struct定义的类有什么不同？

#### 【解答】

区别在于默认的访问标号不同：如果类中某成员的定义之前没有出现任何访问标号，则在用class关键字定义的类中，该成员默认为private成员；而在用struct关键字定义的类中，该成员默认为public成员。

### 习题12.7

什么是封装？为什么封装是有用的？

#### 【解答】

封装是一种将低层次的元素组合起来形成新的、高层次实体的技术。例如，函数是封装的一种形

式：函数所执行的细节行为被封装在函数本身这个更大的实体中；类也是一个封装的实体：它代表若干成员的聚集，大多数（良好设计的）类类型隐藏了实现该类型的成员（见12.1.2节）。

封装隐藏了内部元素的实现细节（例如，可以调用一个函数但不能访问它所执行的语句），其主要优点在于：避免类内部出现无意的、可能破坏对象状态的用户级错误；使得在修改类的实现时只要保持类的接口不变，就无需改变用户级代码。因此，封装是有用的。

### 习题12.8

将Sales\_item::avg\_price定义为内联函数。

#### 【解答】

将一个类成员函数（如Sales\_item::avg\_price）定义为内联函数，有三种方式。

方式一：将函数的定义写在类定义体内部。

```
class Sales_item {
public:
    double Sales_item::avg_price() const
    {
        if (units_sold)
            return revenue/units_sold;
        else
            return 0;
    }
    //...其他成员函数略
private:
    std::string isbn;
    unsigned units_sold;
    double revenue;
};
```

方式二：在类定义体内部的函数声明上用inline显式指定。

```
class Sales_item {
public:
    inline double Sales_item::avg_price() const;
    //...其他成员函数略
private:
    std::string isbn;
    unsigned units_sold;
    double revenue;
};
// ...avg_price的定义略
```

方式三：在类定义体外部的函数定义上用inline显式指定。

```
class Sales_item {
public:
    double Sales_item::avg_price() const;
    //...其他成员函数略
private:
    std::string isbn;
    unsigned units_sold;
    double revenue;
};

inline double Sales_item::avg_price() const
{
```

```

    if (units_sold)
        return revenue/units_sold;
    else
        return 0;
}

```

**习题12.9**

修改本节中给出的Screen类，给出一个构造函数，根据屏幕的高度、宽度和内容的值来创建Screen。

**【解答】**

在screen类的定义体内public部分增加如下构造函数的声明：

```
Screen(index hght, index wdth, const std::string &cntnts);
```

在该类的定义体之外增加如下构造函数的定义：

```
Screen::Screen(index hght, index wdth, const std::string &cntnts):
    contents(cntnts), cursor(0), height(hght), width(wdth)
{}
```

或者，也可以仅在screen类的定义体内public部分增加如下构造函数的定义：

```
Screen::Screen(index hght, index wdth, const std::string &cntnts):
    contents(cntnts), cursor(0), height(hght), width(wdth)
{}
```

注意，这两种方式有一点点区别，后者定义的构造函数是一个inline函数，前者不是；如果要将前者定义为inline函数，可在其声明或定义上显式指定inline（见习题12.8的解答）。

**习题12.10**

解释下述类中的每个成员：

```

class Record {
    typedef std::size_t size;
    Record(): byte_count(0) { }
    Record(size s): byte_count(s) { }
    Record(std::string s): name(s), byte_count(0) { }
    size byte_count;
    std::string name;
public:
    size get_count() const { return byte_count; }
    std::string get_name() const { return name; }
};

```

**【解答】**

size是Record类中定义的局部类型名字，三个Record函数是Record类的重载构造函数，分别接受不同的参数，byte\_count和name是Record类的数据成员，以上成员的访问标号为默认的private，只能在Record类中使用；get\_count和get\_name是Record类的成员函数，其访问标号为public，可以由程序的任意部分使用。

**习题12.11**

定义两个类X和Y，X中有一个指向Y的指针，Y中有一个X类型的对象。

**【解答】**

```
class Y;
class X{
    //...类中其他成员
private:
    Y *ptr;
};
class Y{
    //...类中其他成员
    X obj;
};
```

注意，类Y的定义出现在类X的定义之后，因此，当类X的定义中使用到类Y时，应在类X的定义之前给出类Y的前向声明。

**习题12.12**

解释类声明与类定义之间的差异。何时使用类声明？何时使用类定义？

**【解答】**

类声明由关键字 class 和类名字构成，类定义由关键字 class、类名字以及由一对花括号括住的类定义体构成。

类声明给出了一个不完全类型，只能用于定义指向该类型的指针及引用，或者用于声明使用该类型作为形参类型或返回值类型的函数。例如，在上题中，因为需要在X类中定义指向Y类的指针，而此时类Y尚未定义，所以在类X的定义之前给出了类Y的声明。

如果要创建类的对象以及使用引用或指针访问类的成员，必须事先给出类定义，否则，编译器无法确定应分配多少内存。

**习题12.13**

扩展Screen类以包含move、set和display操作。通过执行如下表达式来测试类：

```
// move cursor to given position, set that character and display the screen
myScreen.move(4,0).set('#').display(cout);
```

**【解答】**

最简单的一种解决方式是参照12.1.3节至12.2节的内容，给出如下程序：

```
// 12-13.cpp
// 扩展Screen类以包含move、set和display操作。
// 通过执行如下表达式来测试类：
// myScreen.move(4,0).set('#').display(cout)
#include <iostream>
#include <string>
using namespace std;

class Screen {
public:
    typedef string::size_type index;
    char get() const { return contents[cursor]; }
    inline char get(index ht, index wd) const;
```

```

index get_cursor() const;
Screen(index hght, index wdth, const string &cntnts);

// 增加三个成员函数
Screen& move(index r, index c);
Screen& set(char c);
Screen& display(ostream &os);

private:
    std::string contents;
    index cursor;
    index height, width;
};

Screen::Screen(index hght, index wdth, const string &cntnts):
    contents(cntnts), cursor(0), height(hght), width(wdth)
{
}

char Screen::get(index r, index c) const
{
    index row = r * width;
    return contents[row + c];
}

inline Screen::index Screen::get_cursor() const
{
    return cursor;
}

// 增加的三个成员函数的定义
Screen& Screen::set(char c)
{
    contents[cursor] = c;
    return *this;
}

Screen& Screen::move(index r, index c)
{
    index row = r * width;
    cursor = row + c;
    return *this;
}

Screen& Screen::display(ostream &os)
{
    os << contents;
    return *this;
}

int main()
{
    // 根据屏幕的高度、宽度和内容的值来创建Screen
    Screen myScreen(5, 6, "aaaaaa\naaaaaaaaaaaaaaaaaaaaaaaa\n");
    // 将光标移至指定位置，设置字符并显示屏幕内容
    myScreen.move(4, 0).set('#').display(cout);

    return 0;
}

```

这个解决方法已满足了题目提出的要求，但存在一些缺陷：

- (1) 创建Screen对象时必须给出表示整个屏幕内容的字符串，即使有些位置上没有内容。

(2) 显示的屏幕内容没有恰当地分行，而是连续显示，因此(4,0)位置上的'#'，在实际显示时不一定正好在屏幕的(4,0)位置，显示效果较差。

(3) 如果创建的Screen对象是一个const对象，则不能使用display函数进行显示（因为const对象只能使用const成员）。

(4) 如果move操作的目的位置超出了屏幕的边界，会出现运行时错误。

要解决第一个缺陷，可以如下修改构造函数：

```
Screen::Screen(index hght, index wdth, const string &cntnts = ""):  
    cursor(0), height(hght), width(wdth)  
{  
    // 将整个屏幕内容置为空格  
    contents.assign(hght*wdth, ' ');  
    // 用形参string对象的内容设置屏幕的相应字符  
    if (cntnts.size() != 0)  
        contents.replace(0, cntnts.size(), cntnts);  
}
```

要解决第二个缺陷，可以如下修改display函数：

```
Screen& Screen::display(ostream &os)  
{  
    string::size_type index = 0;  
    while (index != contents.size()) {  
        os << contents[index];  
        if ((index+1) % width == 0) {  
            os << '\n';  
        }  
        ++index;  
    }  
    return *this;  
}
```

要解决第三个缺陷，可以在Screen类定义体中增加如下函数声明：

```
const Screen& display(ostream &os) const;
```

声明display函数的一个重载版本，供const对象使用。并在Screen类定义体外增加该函数的定义如下：

```
const Screen& Screen::display(ostream &os) const  
{  
    string::size_type index = 0;  
    while (index != contents.size()) {  
        os << contents[index];  
        if ((index+1) % width == 0) {  
            os << '\n';  
        }  
        ++index;  
    }  
    return *this;  
}
```

注意，两个重载的display函数的函数体完全一样，因此，为了减少重复代码，可以定义一个do\_display函数来完成实际的显示工作，而两个重载的display函数都调用该函数。这个起辅助作用的do\_display函数可以定义在Screen类的private部分。读者可以自己完成这一工作。

要解决第四个缺陷，可以如下修改move函数：

```

Screen& Screen::move(index r, index c)
{
    // 行、列号均从0开始
    if (r >= height || c >= width) {
        cerr << "invalid row or column" << endl;
        throw EXIT_FAILURE;
    }
    index row = r * width;
    cursor = row + c;

    return *this;
}

```

修改后的move函数对目的行和列的标号进行检查以避免操作越界。

### 习题12.14

通过this指针引用成员虽然合法，但却是多余的。讨论显式使用this指针访问成员的优缺点。

#### 【解答】

优点：可以非常明确地指出访问的是调用该函数的对象的成员，且可以在成员函数中使用与数据成员同名的形参。

缺点：显得多余，使代码显得累赘。

### 习题12.15

列出在类作用域中的程序文本部分。

#### 【解答】

包括类的定义体，在类外定义的成员函数的定义（其中不包括函数的返回类型）。

### 习题12.16

如果如下定义get\_cursor，将会发生什么：

```

index Screen::get_cursor() const
{
    return cursor;
}

```

#### 【解答】

会出现编译错误。因为，成员函数返回类型index是在Screen类的作用域之外的，而在Screen类之外没有定义index类型。要使用Screen类中定义的index类型，必须在index之前加上Screen::。

### 习题12.17

如果将Screen类中的类型别名放到类中的最后一行，将会发生什么？

#### 【解答】

会出现编译错误。因为，此时对类型别名index的使用出现在其定义之前。

### 习题12.18

解释下述代码。指出每次使用Type或initVal时用到的是哪个名字定义。如果存在错误，说明如何改正。

```

typedef string Type;
Type initVal();

class Exercise {
public:
    // ...
    typedef double Type;
    Type setVal(Type);
    Type initVal();
private:
    int val;
};

Type Exercise::setVal(Type parm) {
    val = parm + initVal();
}

```

成员函数setVal的定义有错。进行必要的修改以便类Exercise使用全局的类型别名Type和全局函数initVal。

### 【解答】

代码解释见注释：

```

typedef string Type; // 定义全局的类型别名Type
Type initVal(); // 全局的initVal函数声明

// 从此处开始Exercise类的定义体
class Exercise {
public:
    // ...
    typedef double Type; // 定义Exercise类内部的类型别名Type
    Type setVal(Type); // 成员函数setVal的声明
    Type initVal(); // 成员函数initVal的声明
private:
    int val; // 定义数据成员
}; // Exercise类的定义体的结束

// Exercise类的成员函数setVal的定义
Type Exercise::setVal(Type parm) {
    val = parm + initVal();
}

```

在Exercise类的定义体内，成员函数setVal和initVal的声明中作为形参类型和返回类型的Type，所使用的都是Exercise内部定义的类型别名Type。

在Exercise类的定义体之外成员函数setVal的定义中，作为形参类型的Type，使用的是Exercise内部定义的类型别名Type；作为返回类型的Type，使用的是全局的类型别名Type。

在Exercise类的成员函数setVal的定义中使用的initVal，用到的是Exercise类的成员函数。

成员函数setVal的定义有错：编译器会认为Exercise类中存在两个相互重载的成员函数setVal，但这两个函数的区别仅在于返回类型不同，不能构成合法的函数重载，因此出错；而且该函数应返回一个值。可更正为：

```

Exercise::Type Exercise::setVal(Type parm) {
    val = parm + initVal();
    return val;
}

```

对上述代码进行如下修改，可使得类Exercise使用全局的类型别名Type和全局函数initVal：

```
typedef string Type;
Type initVal();

class Exercise {
public:
    // ...
    //typedef double Type; // 去掉Exercise类中类型别名Type的定义
    Type setVal(Type);
    //Type initVal(); // 去掉Exercise类中的成员函数initVal
private:
    Type val; // 将val的类型改为Type
};

Type Exercise::setVal(Type parm) {
    val = parm + initVal();
    return val; // 加上返回函数值的return语句
}
```

### 习题12.19

提供一个或多个构造函数，允许该类的用户不指定数据成员的初始值或指定所有数据成员的初始值：

```
class NoName {
public:
    // constructor(s) go here ...
private:
    std::string *pstring;
    int ival;
    double dval;
};
```

解释如何确定需要多少个构造函数以及它们应该接受什么样的形参。

#### 【解答】

允许类用户不指定数据成员的初始值的构造函数应该是不带参数的构造函数，而允许类用户指定所有数据成员的初始值的构造函数应该带有三个形参（类型分别与三个数据成员的类型匹配），因此需要提供两个构造函数。

方法一：在NoName类的定义体中public部分增加下面两个构造函数的定义。

```
NoName()
{
}
NoName(std::string *pstr, int iv, double dv)
{
    pstring = pstr;
    ival = iv;
    dval = dv;
}
```

方法二：在NoName类的定义体中public部分增加下面两个构造函数的声明。

```
NoName();
NoName(std::string *pstr, int iv, double dv);
```

然后在NoName类的定义体外给出它们的定义：

```
NoName::NoName()
```

```

{
}
NoName::NoName(std::string *pstr, int iv, double dv)
{
    pstring = pstr;
    ival = iv;
    dval = dv;
}

```

注意，方法一中定义的构造函数默认为inline函数，方法二中也可以将两个构造函数显式指定为inline函数。

### 习题12.20

从下述抽象中选择一个（或一个自己定义的抽象），确定类中需要什么数据，并提供适当的构造函数集。解释你的决定：

- |             |            |              |
|-------------|------------|--------------|
| (a) Book    | (b) Date   | (c) Employee |
| (d) Vehicle | (e) Object | (f) Tree     |

### 【解答】

选择(b)Date。因为确定一个日期需要有年、月、日三个部分，所以Date类应该有三个数据成员：year、month、day。

可以为Date类提供两个构造函数：一个默认构造函数，用于创建空的Date对象；一个带有三个形参的构造函数，用于创建指定日期的Date对象。

Date类可定义如下：

```

class Date{
public:
    Date() {}
    Date(int yy, int mm, int dd): year(yy), month(mm), day(dd)
    {
    }
    // ...其他成员函数
private:
    int year, month, day;
};

```

### 习题12.21

使用构造函数初始化列表编写类的默认构造函数，该类包含如下成员：一个const string、一个int、一个double\*和一个ifstream&。初始化string来保存类的名字。

### 【解答】

假设该类的名字为DemoClass，其中数据成员name的类型为const string，因为题目只要求对name进行初始化，所以可以如下编写其默认构造函数：

```
DemoClass(): name("DemoClass") {}
```

或者也可以如下编写该类：

```

ifstream file("text1");
class DemoClass {
public:

```

```
DemoClass(): name("DemoClass"), ival(0), pd(0), inFile(file) { }
private:
    const string name;
    int ival;
    double *pd;
    ifstream &inFile;
};
```

**习题12.22**

下面的初始化式有错误。找出并改正错误。

```
struct X {
    X (int i, int j): base(i), rem(base % j) { }
    int rem, base;
};
```

**【解答】**

根据类X中数据成员的定义次序，应该首先初始化rem，然后再初始化base，上述初始化列表的效果是用尚未初始化的base值来初始化rem，所以出错。

可更正为：

```
struct X {
    X (int i, int j): base(i), rem(base % j) { }
    int base, rem; // 先定义base，再定义rem
};
```

或者：

```
struct X {
    X (int i, int j): base(i), rem(i % j) { }
    int rem, base;
};
```

**习题12.23**

假定有个命名为NoDefault的类，该类有一个接受一个int的构造函数，但没有默认构造函数。定义有一个NoDefault类型成员的类C。为类C定义默认构造函数。

**【解答】**

NoDefault类有一个接受一个int的构造函数，但没有默认构造函数，因此类C的默认构造函数必须为调用NoDefault类的构造函数提供常量作为初始化式，可定义如下：

```
class C {
public:
    C() : noDeObj(10) { }
    // ...其他成员
private:
    NoDefault noDeObj;
    // ...其他成员
};
```

**习题12.24**

上面的Sales\_item定义了两个构造函数，其中之一有一个默认实参对应其单个string形参。使用

该Sales\_item版本，确定用哪个构造函数来初始化下述的每个变量，并列出每个对象中数据成员的值：

```
Sales_item first_item(cin);
int main() {
    Sales_item next;
    Sales_item last("9-999-99999-9");
}
```

### 【解答】

使用接受一个std::istream&参数的构造函数来初始化对象first\_item，用从标准输入设备读入的值对first\_item进行初始化。

使用接受一个std::string&参数的构造函数来初始化对象next和last。next中各数据成员的值为：isbn为空字符串，units\_sold为0，revenue为0.0；last中各数据成员的值为：isbn为“9-999-99999-9”，units\_sold为0，revenue为0.0。

### 习题12.25

逻辑上讲，我们可能希望将cin作为默认实参提供给接受一个istream&形参的构造函数。编写使用cin作为默认实参的构造函数声明。

### 【解答】

```
Sales_item(std::istream &is = std::cin);
```

### 习题12.26

接受一个string和接受一个istream&的构造函数都具有默认实参是合法的吗？如果不是，为什么？

### 【解答】

如果接受一个string和接受一个istream&的构造函数都具有默认实参，则不合法。

因为如果二者都具有默认实参，则造成了默认构造函数的重复定义。当定义Sales\_item对象而没有给出用于构造函数的实参时，将因无法确定使用哪个构造函数而出现编译错误（函数调用的二义性）。

### 习题12.27

下面的陈述中哪个是不正确的（如果有的话）？为什么？

- (a) 类必须提供至少一个构造函数。
- (b) 默认构造函数的形参列表中没有形参。
- (c) 如果一个类没有有意义的默认值，则该类不应该提供默认构造函数。
- (d) 如果一个类没有定义默认构造函数，则编译器会自动生成一个，同时将每个数据成员初始化为相关类型的默认值。

### 【解答】

(a) 不正确。因为类也可以不提供构造函数，这时使用由编译器合成的默认构造函数。

(b) 不正确。因为所有形参都提供了默认实参的构造函数也定义了默认构造函数，而这样的构造函数形参列表中是有形参的。

(c) 不正确。因为如果一个类没有默认构造函数（指的是该类提供了构造函数，但没有提供自己的默认构造函数），则在编译器需要隐式使用默认构造函数的环境中，该类就不能使用，所以，如果一

个类定义了其他构造函数，则通常也应该提供一个默认构造函数（见12.4.3节）。

(d)不正确。因为编译器合成的默认构造函数，不是将每个数据成员初始化为相关类型的默认值，而是使用与变量初始化相同的规则来初始化成员：类类型的成员执行各自的默认构造函数进行初始化；内置和复合类型的成员，只对定义在全局作用域中的对象才初始化（见12.4.3节）。

### 习题12.28

解释一下接受一个string的Sales\_item构造函数是否应该为explicit。将构造函数设置为explicit的好处是什么？缺点是什么？

#### 【解答】

接受一个string的Sales\_item构造函数应该为explicit。因为如果不将其声明为explicit，则编译器可以使用它进行隐式类型转换（将一个string对象转换为Sales\_item对象），而这种行为是容易发生语义错误的。

将构造函数设置为explicit的好处是可以避免因隐式类型转换而带来的错误；缺点是当用户的确需要进行相应的类型转换时，不能依靠隐式类型转换，必须显式地创建临时对象。

### 习题12.29

解释在下面的定义中所发生的操作。

```
string null_isbn = "9-999-99999-9";
Sales_item null1(null_isbn);
Sales_item null("9-999-99999-9");
```

#### 【解答】

```
string null_isbn = "9-999-99999-9";
```

首先调用接受一个C风格字符串形参的string构造函数，创建一个临时的string对象，然后，调用string类的复制构造函数（见13.1节）将null\_isbn初始化为该临时对象的副本。

```
Sales_item null1(null_isbn);
```

使用string对象null\_isbn为实参，调用Sales\_item类的构造函数创建Sales\_item对象null1。

```
Sales_item null("9-999-99999-9");
```

首先调用接受一个C风格字符串形参的string构造函数，创建一个临时的string对象，然后使用该临时对象为实参，调用Sales\_item类的构造函数创建Sales\_item对象null。

### 习题12.30

编译如下代码：

```
f(const vector<int>&);
int main() {
    vector<int> v2;
    f(v2); // should be ok
    f(42); // should be an error
    return 0;
}
```

基于对f的第二个调用中出现的错误，我们可以对vector构造函数作出什么推断？如果该调用成

功了，那么你能得出什么结论？

### 【解答】

可以用单个实参调用的构造函数定义从形参类型到该类类型的隐式转换，如果这样的构造函数被声明为`explicit`，则编译器不使用它作为转换操作符。函数f的形参为`const vector<int>&`类型，如果实参为`vector<int>`类型或能够隐式转换为`vector<int>`类型，则函数调用成功。因此，基于对f的第二个调用中出现的错误，我们可以对`vector`构造函数作出如下推断：`vector`中没有定义接受一个`int`型参数的构造函数，或者即使定义了接受一个`int`型参数的构造函数，该构造函数也被设置为`explicit`。

如果该调用成功了，则说明`vector`中定义了接受一个`int`型参数的非`explicit`构造函数。

### 习题12.31

`pair`的数据成员为`public`，然而下面这段代码却不能编译，为什么？

```
pair<int, int> p2 = {0, 42}; // doesn't compile, why?
```

### 【解答】

因为`pair`类定义了构造函数，所以尽管其数据成员为`public`，也不能采用这种显式初始化方式。只有没有定义构造函数且其全体数据成员均为`public`的类，才可以采用与初始化数组元素相同的方式初始化其成员。

可更正为：

```
pair<int, int> p2(0, 42);
```

### 习题12.32

什么是友元函数？什么是友元类？

### 【解答】

被指定为某类的友元的函数称为该类的友元函数。

被指定为某类的友元的类称为授予友元关系的那个类的友元类。

### 习题12.33

什么时候友元是有用的？讨论使用友元的优缺点。

### 【解答】

在需要允许某些特定的非成员函数访问一个类的私有成员（及受保护成员），而同时仍阻止一般的访问的情况下，友元是有用的。

使用友元的优点：可以灵活地实现需要访问若干类的私有或受保护成员才能完成的任务；便于与其他不支持类概念的语言（如C语言、汇编语言等）进行混合编程；通过使用友元函数重载可以更自然地使用C++语言的I/O流库。

使用友元的缺点：一个类将对其非公有成员的访问权授予其他的函数或类，会破坏该类的封装性，降低该类的可靠性和可维护性。

**习题12.34**

定义一个将两个Sales\_item对象相加的非成员函数。

**【解答】**

```
Sales_item add(const Sales_item &obj1, const Sales_item &obj2)
{
    if (!obj1.same_isbn(obj2))
        return obj1;
    Sales_item temp;
    temp.isbn = obj1.isbn;
    temp.units_sold = obj1.units_sold + obj2.units_sold;
    temp.revenue = obj1.revenue + obj2.revenue;
    return temp;
}
```

注意，需要将该函数指定为Sales\_item类的友元。

相应的Sales\_item类可定义如下：

```
class Sales_item {
public:
    // 将add函数指定为Sales_item类的友元
    friend Sales_item add(const Sales_item&, const Sales_item&);
    bool same_isbn(const Sales_item &rhs) const
    { return isbn == rhs.isbn; }

    // 构造函数
    Sales_item(const std::string &book = "") :
        isbn(book), units_sold(0), revenue(0.0) {}
    Sales_item(std::istream &is)
    {
        cin >> isbn >> units_sold >> revenue;
    }
private:
    std::string isbn;
    unsigned units_sold;
    double revenue;
};
```

**习题12.35**

定义一个非成员函数，读取一个istream并将读入的内容存储到一个Sales\_item中。

**【解答】**

可定义如下read函数：

```
std::istream& read(std::istream &is, Sales_item& obj)
{
    double price;
    is >> obj.isbn >> obj.units_sold >> price;
    if (is)      // 读入成功
        obj.revenue = obj.units_sold * price;
    else        // 读入不成功
        obj = Sales_item(); // 将obj重置为默认状态
    return is;
}
```

注意，需要将该函数指定为Sales\_item类的友元。对用户而言，读入售出价格比读入销售收入更

方便，因此read函数读入price并计算revenue。

相应的Sales\_item类可定义如下：

```
class Sales_item {
public:
    // 将read函数指定为Sales_item类的友元
    friend istream& read(istream&, Sales_item&);
    bool same_isbn(const Sales_item &rhs) const
    { return isbn == rhs.isbn; }

    // 构造函数
    Sales_item(const std::string &book = "") :
        isbn(book), units_sold(0), revenue(0.0) {}

private:
    std::string isbn;
    unsigned units_sold;
    double revenue;
};
```

### 习题12.36

什么是static类成员？static成员的优点是什么？它们与普通成员有什么不同？

#### 【解答】

所谓static类成员，指的是其声明前带有关键字static的类成员。static类成员不是任意对象的组成部分，但由该类的全体对象所共享。

static成员的优点是：

- static成员的名字是在类的作用域中，因此可以避免与其他类的成员或全局对象名字冲突；
- 可以实施封装。static成员可以是私有成员，而全局对象不可以；
- 通过阅读程序容易看出static成员是与特定类关联的。这种可见性可清晰地显示程序员的意图。

它们与普通成员的不同在于：普通成员总是与对象相关联，是某个对象的组成部分；而static成员与类相关联，由该类的全体对象所共享，不是任意对象的组成部分。

### 习题12.37

编写自己的Account类版本。

#### 【解答】

可如下定义Account类：

```
class Account {
public:
    // 构造函数
    Account(std::string own, double amnt)
    {
        owner = own;
        amount = amnt;
    }
    // 计算余额
    void applyint()
    {
        amount += amount * interestRate;
    }
    // 返回当前利率
```

```

static double rate()
{
    return interestRate;
}
// 设置新的利率
static void rate(double newRate)
{
    interestRate = newRate;
}
// 存款
double deposit(double amnt)
{
    amount += amnt;
    return amount;
}
// 取款
bool withdraw(double amnt)
{
    if (amount < amnt) // 余额不足
        return false;
    else {
        amount -= amnt;
        return true;
    }
}
// 查询当前余额
double getBalance()
{
    return amount;
}
private:
    std::string owner;
    double amount;
    static double interestRate;
};

double Account::interestRate = 2.5;

```

注意，`static`数据成员必须在类定义体外部定义，且只定义一次。一般可将它放在类的实现文件（源文件）中。

### 习题12.38

定义一个命名为`Foo`的类，具有单个`int`型数据成员。为该类定义一个构造函数，接受一个`int`值并用该值初始化数据成员。为该类定义一个函数，返回其数据成员的值。

#### 【解答】

`Foo`类定义如下：

```

class Foo {
public:
    Foo(int x)
    {
        value = x;
    }
    int get()
    {
        return value;
    }
private:

```

```
    int value;
};
```

**习题12.39**

给定上题中定义的Foo类，定义另一个Bar类。Bar类具有两个static数据成员：一个为int型，另一个为Foo类型。

**【解答】**

Bar类可定义如下：

```
class Bar {
private:
    static int ival;
    static Foo fval;
};
```

**习题12.40**

使用上面两题中定义的类，给Bar类增加一对成员<sup>1</sup>：第一个成员命名为FooVal，返回Bar类的Foo类型static成员的值；第二个成员命名为callsFooVal，保存FooVal<sup>2</sup>被调用的次数。

**【解答】**

Bar类可定义如下：

```
class Bar {
public:
    Foo FooVal()
    {
        callsFooVal++;
        return fval;
    }
private:
    static int ival;
    static Foo fval;
    static int callsFooVal;
};
```

注意，应在Bar类的定义体外部对其static数据成员进行初始化：

```
int Bar::ival = 20;
Foo Bar::fval(0);
int Bar::callsFooVal = 0;
```

**习题12.41**

利用12.6.1节的习题中编写的类Foo和Bar，初始化Bar<sup>3</sup>的static成员。将int成员初始化为20，并将Foo成员初始化为0。

**【解答】**

在Bar类的定义体外部进行如下初始化即可：

1. 此处英文原文有误。
2. 此处英文原文有误。
3. 英文原文误为 Foo。

```
int Bar::ival = 20;
Foo Bar::fval(0);
```

**习题12.42**

下面的static数据成员声明和定义中哪些是错误的（如果有的话）？解释为什么。

```
// example.h
class Example {
public:
    static double rate = 6.5;
    static const int vecSize = 20;
    static vector<double> vec(vecSize);
};

// example.C
#include "example.h"
double Example::rate;
vector<double> Example::vec;
```

**【解答】**

类的定义体中对static成员rate和vec的初始化是错误的。因为非const static成员的初始化必须放在类定义体的外部。example.C文件中对static成员rate和vec的定义也是错误的，因为此处必须给出初始值。

可更正为：

```
// example.h
class Example {
public:
    static double rate;
    static const int vecSize = 20;
    static vector<double> vec;
};

// example.C
#include "example.h"
double Example::rate = 6.5;
//const int Example::vecSize;
vector<double> Example::vec(vecSize);
```

注意，12.6.2节指出：在类的定义体中初始化的const static数据成员，仍必须在类的定义体外进行定义，但定义时无需再指定初始值。但在Microsoft Visual C++ .NET 2003中，这样做会导致连接错误。



## 复 制 控 制

### 习题13.1

什么是复制构造函数？何时使用它？

#### 【解答】

复制构造函数是具有如下特点的构造函数：只有单个形参，且形参是对本类类型对象的引用（常用const修饰）。

复制构造函数在下列情况下使用：

- 根据另一个同类型的对象显式或隐式初始化一个对象。
- 复制一个对象，将它作为实参传给一个函数。
- 从函数返回时复制一个对象。
- 初始化顺序容器中的元素。
- 根据元素初始化式列表初始化数组元素。

### 习题13.2

下面的第二个初始化不能编译。可以从vector的定义得出什么推断？

```
vector<int> v1(42); // ok: 42 elements, each 0
vector<int> v2 = 42;
// error: what does this error tell us about vector?
```

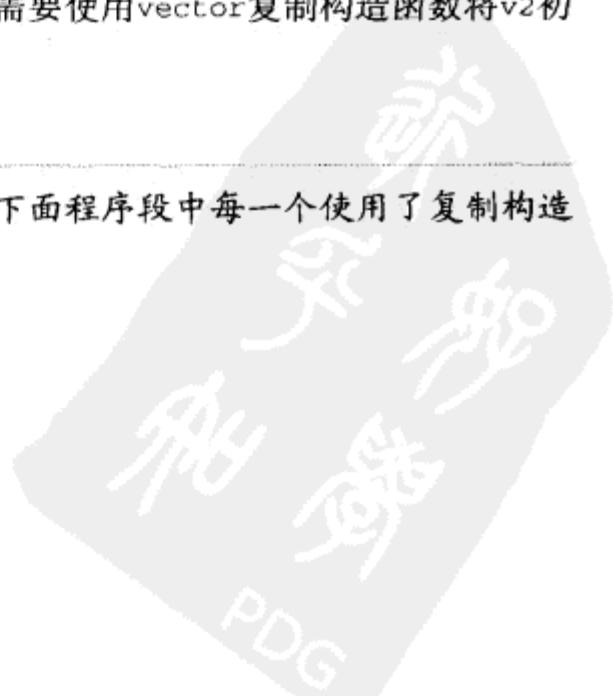
#### 【解答】

从上述第二个初始化不能编译的情况可以推断出：vector容器类没有提供公有的复制构造函数。因为第二个初始化是复制初始化（copy-initialization），创建v2时，编译器首先调用接受一个int型形参的vector构造函数，创建一个临时vector对象，然后，编译器需要使用vector复制构造函数将v2初始化为该临时vector对象的副本。

### 习题13.3

假定Point为类类型，该类类型有一个复制构造函数，指出下面程序段中每一个使用了复制构造函数的地方：

```
Point global;
Point foo_bar(Point arg)
{
    Point local = arg;
    Point *heap = new Point(global);
```



```

    *heap = local;
    Point pa[ 4 ] = { local, *heap };
    return *heap;
}

```

**【解答】**

上述程序段中使用了复制构造函数的地方包括（见下面程序段中的注释）：

```

Point foo_bar(Point arg) // 调用foo_bar函数时将实参Point对象的副本
                        // 传递给形参Point对象arg
{
    Point local = arg; // 对局部Point对象local进行复制初始化
    Point *heap = new Point(global); // 根据全局Point对象global创建新的Point对象

    *heap = local;
    Point pa[ 4 ] = { local, *heap }; // 根据元素初始化式列表初始化数组pa的元素
    return *heap; // 从函数返回Point对象*heap的副本
}

```

**习题13.4**

对于如下的类的简单定义，编写一个复制构造函数来复制所有成员。复制pstring指向的对象而不是复制指针。

```

struct NoName {
    NoName(): pstring(new std::string), i(0), d(0) { }
private:
    std::string *pstring;
    int i;
    double d;
};

```

**【解答】**

复制构造函数可编写如下：

```

NoName::NoName(const NoName& other)
{
    pstring = new std::string;
    *pstring = *(other.pstring);
    i = other.i;
    d = other.d;
}

```

也可以采用初始化列表的形式编写如下：

```

NoName::NoName(const NoName& other):
pstring(new std::string(*(other.pstring))), i(other.i), d(other.d) {}

```

**习题13.5**

哪个类定义可能需要一个复制构造函数？

- (a) 包含4个float成员的Point3w类。
- (b) Matrix类，其中，实际矩阵在构造函数中动态分配，在析构函数中删除。
- (c) Payroll类，在这个类中为每个对象提供唯一ID。
- (d) Word类，包含一个string和一个以行列位置对为元素的vector。

**【解答】**

一般而言，如果一个类拥有指针成员，或者在复制对象时有一些特定工作要做，则该类需要复制构造函数。

(a) Point3w类不需要复制构造函数。因为Point3w类中的数据成员都是内置类型的，没有指针成员，使用编译器提供的复制构造函数即可。

(b) Matrix类需要复制构造函数。因为需要涉及指针及内存的动态分配。

(c) Payroll类需要复制构造函数。因为在根据已存在的Payroll对象创建其副本时，需要提供唯一的ID。

(d) Word类不需要复制构造函数。因为编译器会自动为其数据成员调用string和vector的复制构造函数。

**习题13.6**

复制构造函数的形参并不限制为const，但必须是一个引用。解释这个限制的基本原理，例如，解释为什么下面的定义不能工作。

```
Sales_item::Sales_item(const Sales_item rhs);
```

**【解答】**

上述定义之所以不能工作，是因为它试图以传值（pass-by-value）方式将实参传递给一个复制构造函数。但是，每当以传值方式传递参数时，会导致调用复制构造函数，因此，如果要使用以传值方式传递参数的复制构造函数，必须使用一个“不以传值方式传递参数”的复制构造函数，否则，就会导致复制构造函数的无穷递归调用。所以，复制构造函数的形参必须是一个引用，即以传址（pass-by-reference）方式传递参数。

**习题13.7**

类何时需要定义赋值操作符？

**【解答】**

一般而言，如果一个类需要定义复制构造函数，则该类也需要定义赋值操作符。具体而言，如果一个类中包含指针型数据成员，或者在进行赋值操作时有一些特定工作要做，则该类通常需要定义赋值操作符。

**习题13.8**

对于习题13.5中列出的每个类型，指出类是否需要赋值操作符。

**【解答】**

一般而言，如果一个类拥有指针成员，或者在进行赋值操作时有一些特定工作要做，则该类需要赋值操作符。

(a) Point3w类不需要赋值操作符。因为Point3w类中的数据成员都是内置类型的，没有指针成员，使用编译器提供的赋值操作符即可。

(b) Matrix类需要赋值操作符。因为需要涉及指针及内存的动态分配。

(c) Payroll类需要赋值操作符。因为在用已存在的Payroll对象给另一个Payroll对象赋值时，需

要提供唯一的ID。

(d) word类不需要赋值操作符。因为编译器会自动为其数据成员调用string和vector的赋值操作符。

### 习题13.9

习题13.4中包括NoName类的简单定义。确定这个类是否需要赋值操作符。如果需要，实现它。

#### 【解答】

NoName类中包含指针型数据成员，所以该类需要赋值操作符。可定义如下：

```
NoName& NoName::operator=(const NoName &rhs)
{
    pstring = new std::string;
    *pstring = *(rhs.pstring);
    i = rhs.i;
    d = rhs.d;
    return *this;
}
```

### 习题13.10

定义一个Employee类，包含雇员名字和一个唯一的雇员标识。为该类定义默认构造函数和参数为表示雇员名字的string的构造函数。如果该类需要复制构造函数或赋值操作符，实现这些函数。

#### 【解答】

可以定义一个static数据成员作为计数器，用于为每个雇员产生唯一的雇员标识。因为每个雇员的雇员标识是唯一的，所以该类需要复制构造函数及赋值操作符，以便在“根据已有雇员创建新的雇员”，或者“将一个雇员赋值给另一个雇员”时提供唯一的雇员标识。

Employee类可定义如下：

```
class Employee {
public:
    // 构造函数
    Employee(): name("NoName"), id(counter)
    {
        ++counter;
    }

    Employee(std::string nm): name(nm), id(counter)
    {
        ++counter;
    }

    Employee(const Employee& other): name(other.name), id(counter)
    {
        ++counter;
    }

    // 赋值操作符
    Employee& operator = (const Employee& rhe)
    {
        name = rhe.name;
        return *this;
    }
private:
    std::string name;
```

```

    int id;
    static int counter;
};

```

假设雇员标识为从1开始的整数，在Employee类定义体外部对其static成员进行初始化：

```
int Employee::counter = 1;
```

此处给出的代码中，雇员标识为简单的整数，也可以利用setId成员生成string型的雇员标识，只需进行相应修改即可。例如，如果雇员标识为形如“Empl-0001”的字符串，则可以将Employee类定义如下：

```

class Employee {
public:
    // 构造函数
    Employee(): name("NoName")
    {
        setId();
    }

    Employee(std::string nm): name(nm)
    {
        setId();
    }

    Employee(const Employee& other): name(other.name)
    {
        setId();
    }

    // 赋值操作符
    Employee& operator = (const Employee& rhe)
    {
        name = rhe.name;
        return *this;
    }
private:
    std::string name;
    std::string id;
    static int counter;
    // 设置雇员ID
    void setId()
    {
        id = "Empl-";
        if (counter < 10)
            id += "000";
        else if(counter < 100)
            id += "00";
        else if(counter < 1000)
            id += "0";
        else {
            std::cerr << "no valid employee id!" << std::endl;
        }

        char buffer[5];
        _itoa(counter, buffer, 10); // 使用库函数将整数转换为字符串
        id += buffer;
        ++counter; // 计数器加1
    }
};

```

同样，需在Employee类定义体外部对其static成员进行初始化：

```
int Employee::counter = 1;
```

**习题13.11**

什么是析构函数？合成析构函数有什么用？什么时候会合成析构函数？什么时候一个类必须定义自己的析构函数？

**【解答】**

析构函数是特殊的成员函数，其名字是在类名字前加上一个代字号(~)。该函数没有返回值和形参，用于对象超出作用域或需要删除对象时来清除对象。

合成析构函数的作用：按对象创建时的逆序（即成员在类中声明次序的逆序）撤销每个非static成员。对于类类型的成员，合成析构函数调用该成员的析构函数来撤销对象。

编译器会为每个类合成析构函数。如果有些工作（如释放资源、执行特定操作等）需要析构函数完成，一个类就必须定义自己的析构函数。

**习题13.12**

确定在习题13.4中概略定义的NoName类是否需要析构函数，如果需要，实现它。

**【解答】**

因为该类在构造函数中动态创建了一个string对象，应在析构函数中撤销该对象，所以该类需要析构函数。

其析构函数可定义如下：

```
NoName::~NoName()
{
    delete pstring; // 撤销pstring所指向的string对象
}
```

**习题13.13**

确定在习题13.10中定义的Employee类是否需要析构函数，如果需要，实现它。

**【解答】**

因为该Employee类不需要在析构函数中释放资源，也没有特定的操作要做，所以该类不需要显式定义析构函数，使用编译器合成的析构函数即可满足需要。

**习题13.14**

理解复制控制成员和构造函数的一个良好方式是定义一个简单类，该类具有这些成员，每个成员打印自己的名字：

```
struct Exmpl {
    Exmpl() { std::cout << "Exmpl()" << std::endl; }
    Exmpl(const Exmpl&)
    { std::cout << "Exmpl(const Exmpl&)" << std::endl; }
    // ...
};
```

编写一个像Exmpl这样的类，给出复制控制成员和其他构造函数。然后写一个程序，用不同方式

使用Exmpl类型的对象：作为非引用形参和引用形参传递，动态分配，放在容器中，等等。研究何时执行哪个构造函数和复制控制成员，可以帮助你融会贯通地理解这些概念。

### 【解答】

定义Exmpl类，该类给出复制控制成员和一个默认构造函数，各成员函数输出自己的名字。主程序中以不同方式使用Exmpl类型的对象：作为非引用形参和引用形参传递，动态分配；作为函数返回值，进行赋值操作；作为元素放在vector容器中，以此研究构造函数和复制控制成员的执行情况。

程序代码如下：

```
// 13-14.cpp
// 定义Exmpl类，该类给出复制控制成员和其他构造函数。
// 用不同方式使用Exmpl 类型的对象：
// 作为非引用形参和引用形参传递，动态分配，
// 作为函数返回值，进行赋值操作，作为元素放在vector容器中。
// 研究何时执行哪个构造函数和复制控制成员
#include <vector>
#include <iostream>

struct Exmpl {
    // 默认构造函数
    Exmpl() { std::cout << "Exmpl()" << std::endl; }

    // 复制构造函数
    Exmpl(const Exmpl&)
    { std::cout << "Exmpl(const Exmpl&)" << std::endl; }

    // 赋值操作符
    Exmpl& operator = (const Exmpl &rhe)
    {
        std::cout << "operator = (const Exmpl&)" << std::endl;
        return *this;
    }

    // 析构函数
    ~Exmpl()
    { std::cout << "~Exmpl()" << std::endl; }
};

void func1(Exmpl obj)      // 形参为Exmpl对象
{
}

void func2(Exmpl& obj)     // 形参为Exmpl对象的引用
{
}

Exmpl func3()
{
    Exmpl obj;
    return obj; // 返回Exmpl对象
}

int main()
{
    Exmpl eobj; // 调用默认构造函数创建Exmpl对象eobj
    func1(eobj); // 调用复制构造函数
    // 将形参Exmpl对象创建为实参Exmpl对象的副本
    // 函数执行完毕后调用析构函数撤销形参Exmpl对象
```

```

func2(eobj);      // 形参为Exmpl对象的引用，无需调用复制构造函数传递实参
eobj = func3();   // 调用默认构造函数创建局部Exmpl对象
                  // 函数返回时调用复制构造函数创建作为返回值副本的Exmpl对象
                  // 然后调用析构函数撤销局部Exmpl对象
                  // 然后调用赋值操作符
                  // 执行完赋值操作后,
                  // 调用析构函数撤销作为返回值副本的Exmpl对象
Exmpl *p = new Exmpl; // 调用默认构造函数动态创建Exmpl对象
std::vector<Exmpl> evec(3); // 调用默认构造函数
                            // 创建一个临时值Exmpl对象
                            // 然后3次调用复制构造函数,
                            // 将临时值Exmpl对象复制到
                            // vector容器evec的每个元素
                            // 然后调用析构函数撤销临时值Exmpl对象
· delete p;           // 调用析构函数撤销动态创建的Exmpl对象
return 0;             // evec及eobj生命周期结束，自动调用析构函数撤销
                      // 撤销evec需调用析构函数3次（因为evec有3个元素）
}

```

运行该程序得到如下输出结果：

```

Exmpl()
Exmpl(const Exmpl&)
~Exmpl()
Exmpl()
Exmpl(const Exmpl&)
~Exmpl()
operator = (const Exmpl&)
~Exmpl()
Exmpl()
Exmpl()
Exmpl(const Exmpl&)
Exmpl(const Exmpl&)
Exmpl(const Exmpl&)
~Exmpl()
~Exmpl()
~Exmpl()
~Exmpl()
~Exmpl()
~Exmpl()
~Exmpl()

```

何时执行哪个构造函数和复制控制成员，详见程序代码中的注释。

### 习题13.15

下面的代码段中发生了多少次析构函数的调用？

```

void fcn(const Sales_item *trans, Sales_item accum)
{
    Sales_item item1(*trans), item2(accum);
    if (!item1.same_isbn(item2)) return;
    if (item1.avg_price() <= 99) return;
    else if (item2.avg_price() <= 99) return;
    // ...
}

```

### 【解答】

共发生了3次析构函数调用。分别用于在函数fcn返回时撤销形参对象accum、局部对象item1和item2。

**习题13.16**

编写本节中描述的Message类。

**【解答】**

习题13.19的解答中给出了本节中描述的Message类的定义，此处不再赘述。

**习题13.17**

为Message类增加与Folder的addMsg和remMsg操作类似的函数。这些函数可以命名为addFldr和remFldr，应接受一个指向Folder的指针并将该指针插入到folders。这些函数可为private的，因为它们将仅在Message类的实现中使用<sup>1</sup>。

**【解答】**

此题的要求有误。

正如Message类的实现需要使用Folder类的addMsg和remMsg操作一样，Message类的addFldr和remFldr操作也需要在Folder类的实现中使用，所以，这些函数都需要定义为public。

Message类的addFldr和remFldr函数可定义如下：

```
void Message::addFldr(Folder* fldr)
{
    folders.insert(fldr);
}

void Message::remFldr(Folder* fldr)
{
    folders.erase(fldr);
}
```

**习题13.18**

编写相应的Folder类。该类应保存一个set<Message\*>，包含指向Message的元素。

**【解答】**

Folder类可定义如下：

```
class Message;
class Folder {
public:
    Folder() { }
    // 复制控制成员
    Folder(const Folder&);
    Folder& operator=(const Folder&);
    ~Folder();

    // 在指定Message的目录集中增加/删除该目录
    void save (Message&);
    void remove(Message&);

    // 在该目录的消息集中增加/删除指定Message
    void addMsg(Message*);
    void remMsg(Message*);
```

1. 此处原文有误：正如Message类的实现需要使用Folder类的addMsg和remMsg操作一样，Message类的addFldr和remFldr操作也需要在Folder类的实现中使用，所以，这些函数都需要定义为public。

```

private:
    std::set<Message*> messages; // 该目录中的消息集
    // 复制控制成员所使用的实用函数:
    // 将目录加到形参所指的消息集中
    void put_Fldr_in_Messages(const std::set<Message*>&);

    // 从目录所指的所有消息中删除该目录
    void remove_Fldr_from_Messages();
};

Folder::Folder(const Folder &f):
messages(f.messages)
{
    // 将该目录加到f所指向的每个消息中
    put_Fldr_in_Messages(messages);
}

// 将该目录加到rhs所指的消息集中
void Folder::put_Fldr_in_Messages(const std::set<Message*> &rhs)
{
    for(std::set<Message*>::const_iterator beg = rhs.begin();
        beg != rhs.end(); ++beg)
        (*beg)->addFldr(this); // *beg指向一个消息
}

Folder& Folder::operator=(const Folder &rhs)
{
    if (&rhs != this) {
        remove_Fldr_from_Messages(); // 更新现有消息
        messages = rhs.messages; // 从rhs复制消息指针集
        // 将该目录加到rhs中的每个消息中
        put_Fldr_in_Messages(rhs.messages);
    }
    return *this;
}

// 从对应消息中删除该目录
void Folder::remove_Fldr_from_Messages()
{
    // 从对应消息中删除该目录
    for(std::set<Message*>::const_iterator beg =
        messages.begin(); beg != messages.end(); ++beg)
        (*beg)->remFldr(this); // *beg指向一个消息
}

Folder::~Folder()
{
    remove_Fldr_from_Messages();
}

void Folder::save(Message& msg)
{
    addMsg(&msg);
    msg.addFldr(this); // 更新相应的消息
}

void Folder::remove(Message& msg)
{
    remMsg(&msg);
    msg.remFldr(this); // 更新相应的消息
}

```

```

void Folder::addMsg(Message* msg)
{
    messages.insert(msg);
}

void Folder::remMsg(Message* msg)
{
    messages.erase(msg);
}

```

**习题13.19**

在Message类中增加save和remove操作。这些操作<sup>1</sup>应接受一个Folder，并将该Folder加入到指向这个Message的Folder集中（或从其中删除该Folder）。操作还必须更新Folder以反映它指向该Message，这可以通过调用addMsg或remMsg完成。

**【解答】**

Message类的save和remove操作可定义如下：

```

void Message::save(Folder& fldr)
{
    addFldr(&fldr);
    fldr.addMsg(this); // 更新相应的目录
}

void Message::remove(Folder& fldr)
{
    remFldr(&fldr);
    fldr.remMsg(this); // 更新相应的目录
}

```

下面给出满足习题13.16至习题13.19要求的Message类和Folder类的完整定义：

```

#include <set>
#include <string>
class Message;
class Folder {
public:
    Folder() { }
    // 复制控制成员
    Folder(const Folder&);
    Folder& operator=(const Folder&);
    ~Folder();

    // 在指定Message的目录集中增加/删除该目录
    void save (Message&);
    void remove(Message&);

    // 在该目录的消息集中增加/删除指定Message
    void addMsg(Message* );
    void remMsg(Message* );
private:

```

<sup>1</sup> 此处原文有误，应该是指Message类的save和remove操作。Folder类的save和remove操作应接受一个Message，并将该Message加入到这个Folder所指向的Message集中（或从其中删除该Message）。Folder类的save和remove操作还必须更新Message以反映它指向该Folder，这可以通过调用addFldr或remFldr完成。

```

std::set<Message*> messages; // 该目录中的消息集

// 复制控制成员所使用的实用函数:
// 将目录加到形参所指的消息集中
void put_Fldr_in_Messages(const std::set<Message*>&);

// 从目录所指的所有消息中删除该目录
void remove_Fldr_from_Messages();
};

class Message {
public:
    // folders自动初始化为空集
    Message(const std::string &str = "") :
        contents(str) {}

    // 复制控制成员
    Message(const Message&);
    Message& operator=(const Message&);
    ~Message();

    // 在指定Folder的消息集中增加/删除该消息
    void save(Folder&);
    void remove(Folder&);

    // 在包含该消息的目录集中增加/删除指定Folder
    void addFldr(Folder*);
    void remFldr(Folder*);
private:
    std::string contents;           // 实际消息文本
    std::set<Folder*> folders;     // 包含该消息的目录

    // 复制构造函数、赋值、析构函数所使用的实用函数:
    // 将消息加到形参所指的目录集中
    void put_Msg_in_Folders(const std::set<Folder*>&);

    // 从消息所在的所有目录中删除该消息
    void remove_Msg_from_Folders();
};

Folder::Folder(const Folder &f):
messages(f.messages)
{
    // 将该目录加到f所指向的每个消息中
    put_Fldr_in_Messages(messages);
}

// 将该目录加到rhs所指的消息集中
void Folder::put_Fldr_in_Messages(const std::set<Message*> &rhs)
{
    for(std::set<Message*>::const_iterator beg = rhs.begin();
        beg != rhs.end(); ++beg)
        (*beg)->addFldr(this); // *beg指向一个消息
}

Folder& Folder::operator=(const Folder &rhs)
{
    if (&rhs != this) {
        remove_Fldr_from_Messages(); // 更新现有消息
        messages = rhs.messages;    // 从rhs复制消息指针集
        // 将该目录加到rhs中的每个消息中
    }
}

```

```

        put_Fldr_in_Messages(rhs.messages);
    }
    return *this;
}

// 从对应消息中删除该目录
void Folder::remove_Fldr_from_Messages()
{
    // 从对应消息中删除该目录
    for(std::set<Message*>::const_iterator beg =
        messages.begin(); beg != messages.end(); ++beg)
        (*beg)->remFldr(this); // *beg指向一个消息
}

Folder::~Folder()
{
    remove_Fldr_from_Messages();
}

void Folder::save(Message& msg)
{
    addMsg(&msg);
    msg.addFldr(this); // 更新相应的消息
}

void Folder::remove(Message& msg)
{
    remMsg(&msg);
    msg.remFldr(this); // 更新相应的消息
}

void Folder::addMsg(Message* msg)
{
    messages.insert(msg);
}

void Folder::remMsg(Message* msg)
{
    messages.erase(msg);
}

Message::Message(const Message &m):
contents(m.contents), folders(m.folders)
{
    // 将该消息加到指向m的每个目录中
    put_Msg_in_Folders(folders);
}

// 将该消息加到rhs所指的目录集中
void Message::put_Msg_in_Folders(const std::set<Folder*> &rhs)
{
    for(std::set<Folder*>::const_iterator beg = rhs.begin();
        beg != rhs.end(); ++beg)
        (*beg)->addMsg(this); // *beg指向一个目录
}

Message& Message::operator=(const Message &rhs)
{
    if (&rhs != this) {
        remove_Msg_from_Folders(); // 更新现有目录
        contents = rhs.contents; // 从rhs复制消息内容
    }
}

```

```

        folders = rhs.folders;    // 从rhs复制目录指针集
        // 将该消息加到中的每个目录中
        put_Msg_in_Folders(rhs.folders);
    }
    return *this;
}

// 从对应目录中删除该消息
void Message::remove_Msg_from_Folders()
{
    // 从对应目录中删除该消息
    for(std::set<Folder*>::const_iterator beg =
        folders.begin(); beg != folders.end(); ++beg)
        (*beg)->remMsg(this); // *beg指向一个目录
}

Message::~Message()
{
    remove_Msg_from_Folders();
}

void Message::save(Folder& fldr)
{
    addFldr(&fldr);
    fldr.addMsg(this); // 更新相应的目录
}

void Message::remove(Folder& fldr)
{
    remFldr(&fldr);
    fldr.remMsg(this); // 更新相应的目录
}

void Message::addFldr(Folder* fldr)
{
    folders.insert(fldr);
}

void Message::remFldr(Folder* fldr)
{
    folders.erase(fldr);
}

```

### 习题13.20

对于HasPtr类的原始版本（依赖于复制控制的默认定义），描述下面代码中会发生什么：

```

int i = 42;
HasPtr p1(&i, 42);
HasPtr p2 = p1;
cout << p2.get_ptr_val() << endl;
p1.set_ptr_val(0);
cout << p2.get_ptr_val() << endl;

```

#### 【解答】

因为依赖于复制控制的默认定义，而默认复制控制实现逐个成员的复制，对于指针成员进行复制之后，会导致不同对象中的指针成员指向相同的对象，所以，上述代码中，p1和p2的指针成员ptr都指向对象i。执行语句cout << p2.get\_ptr\_val() << endl;会输出i的值42，而执行语句p1.set\_ptr\_val(0);会将i的值置0，然后执行语句cout << p2.get\_ptr\_val() << endl;会输出i的值0。

**习题13.21**

如果给HasPtr类添加一个析构函数，用来删除指针成员，会发生什么？

**【解答】**

如果给HasPtr类添加一个析构函数，用来删除指针成员，那么撤销一个HasPtr对象时会删除（撤销）其指针成员所指向的对象，从而使得当一个HasPtr对象被撤销之后，其他由该对象复制而创建的HasPtr对象中的指针成员也无法使用（因为该指针成员所指向的对象已被撤销，所以指针不再指向有效对象）。

**习题13.22**

什么是使用计数？

**【解答】**

使用计数（use count）是复制控制成员中使用的编程技术。将一个计数器与类指向的对象相关联，用于跟踪该类有多少个对象共享同一指针。创建一个单独类指向共享对象并管理使用计数。由构造函数设置共享对象的状态并将使用计数置为1。每当由复制构造函数或赋值操作符生成一个新副本时，使用计数加1。由析构函数撤销对象或作为赋值操作符的左操作数撤销对象时，使用计数减1。赋值操作符和析构函数检查使用计数是否已减至0，如果是，则撤销对象。

**习题13.23**

什么是智能指针？智能指针类如何与实现普通指针行为的类相区别？

**【解答】**

智能指针（smart pointer）是一个行为类似指针但也提供其他功能的类。

智能指针类与实现普通指针行为的类的区别在于：智能指针通常接受指向动态分配对象的指针并负责删除该对象。用户分配对象，但由智能指针类删除它，因此智能指针类需要实现复制控制成员来管理指向共享对象的指针。只有在撤销了指向共享对象的最后一个智能指针后，才能删除该共享对象。使用计数是实现智能指针类最常用的方式。

**习题13.24**

实现你自己的使用计数式HasPtr类的版本。

**【解答】**

该HasPtr类可实现如下（见13.5.1节）：

```
// 定义仅由HasPtr类使用的U_Ptr类，用于封装使用计数和相关指针
class U_Ptr {
    friend class HasPtr; // 将HasPtr类指定为U_Ptr类的友元
    int *ip;
    size_t use;
    U_Ptr(int *p): ip(p), use(1) { }
    ~U_Ptr() { delete ip; }
};

// 定义HasPtr类
class HasPtr {
public:
```



```

// 构造函数: p是指向已经动态创建的int对象的指针
HasPtr(int *p, int i): ptr(new U_Ptr(p)), val(i) { }

// 复制构造函数: 复制成员并将使用计数加1
HasPtr(const HasPtr &orig):
ptr(orig.ptr), val(orig.val) { ++ptr->use; }

// 赋值操作符
HasPtr& operator=(const HasPtr&);

// 析构函数: 如果使用计数为0, 则删除U_Ptr对象
~HasPtr() { if (--ptr->use == 0) delete ptr; }

// 获取数据成员
int *get_ptr() const { return ptr->ip; }
int get_int() const { return val; }

// 修改数据成员
void set_ptr(int *p) { ptr->ip = p; }
void set_int(int i) { val = i; }

// 返回或修改基础int对象
int get_ptr_val() const { return *ptr->ip; }
void set_ptr_val(int i) { *ptr->ip = i; }
private:
    U_Ptr *ptr; // points to use-counted U_Ptr class
    int val;
};

HasPtr& HasPtr::operator=(const HasPtr &rhs)
{
    // 增加右操作数中的使用计数
    ++rhs.ptr->use;
    // 将左操作数对象的使用计数减1,
    // 若该对象的使用计数减至0, 则删除该对象
    if (--ptr->use == 0)
        delete ptr;
    ptr = rhs.ptr; // 复制U_Ptr指针
    val = rhs.val; // 复制int成员
    return *this;
}

```

**习题13.25**

什么是值型类?

**【解答】**

所谓值型类, 是指具有值语义的类, 其特征为: 对该类对象进行复制时, 会得到一个不同的新副本, 对副本所做的改变不会影响原有对象。

**习题13.26**

实现你自己的值型HasPtr类版本。

**【解答】**

值型HasPtr类可定义如下(见13.5.2节):

```
// 定义值型HasPtr类
class HasPtr {
```

```

public:
    // 构造函数: 保存指向形参副本对象的指针
    HasPtr(const int &p, int i): ptr(new int(p)), val(i) {}

    // 复制构造函数: 创建新的基础int对象, 保存与被复制基础对象相同的值
    HasPtr(const HasPtr &orig):
        ptr(new int (*orig.ptr)), val(orig.val) {}

    // 赋值操作符
    HasPtr& operator=(const HasPtr&);

    // 析构函数: 无条件删除基础int对象
    ~HasPtr() { delete ptr; }

    // 获取数据成员
    int get_ptr_val() const { return *ptr; }
    int get_int() const { return val; }

    // 修改数据成员
    void set_ptr(int *p) { ptr = p; }
    void set_int(int i) { val = i; }

    // 返回指针成员
    int *get_ptr() const { return ptr; }

    // 设置指针成员所指向的基础对象
    void set_ptr_val(int p) const { *ptr = p; }
private:
    int *ptr; // ptr指向int对象
    int val;
};

// 赋值操作符不需要分配新对象, 只是给其指针所指向的对象赋新值
HasPtr& HasPtr::operator=(const HasPtr &rhs)
{
    *ptr = *rhs.ptr; // 复制ptr指向的基础对象
    val = rhs.val;   // 复制int成员
    return *this;
}

```

### 习题13.27

值型HasPtr类定义了所有复制控制成员。描述将会发生什么，如果该类：

- (a) 定义了复制构造函数和析构函数但没有定义赋值操作符。
- (b) 定义了复制构造函数和赋值操作符但没有定义析构函数。
- (c) 定义了析构函数但没有定义复制构造函数和赋值操作符。

#### 【解答】

(a) 如果值型HasPtr类定义了复制构造函数和析构函数但没有定义赋值操作符，则使用编译器合成的赋值操作符。因此，若将一个HasPtr对象赋值给另一HasPtr对象，则两个HasPtr对象的ptr成员值相同（即二者指向同一基础int对象）。当其中一个HasPtr对象被撤销之后，该基础int对象也被撤销，使得另一个HasPtr对象中的ptr成员指向一个不复存在的基础int对象，从而通过该HasPtr对象的ptr成员使用其指向的基础int对象会出现问题。而且当另一个HasPtr对象被撤销时，会因为析构函数对同一指针进行重复删除而出现内存访问非法的错误。此外，被赋值HasPtr对象的ptr成员原来所指

向的基础int对象不能再被访问，但也没有撤销。

(b) 如果值型HasPtr类定义了复制构造函数和赋值操作符但没有定义析构函数，就使用编译器合成的析构函数。因此，当一个HasPtr对象被撤销后，其成员所指向的基础int对象不会被撤销，会一直存在，从而导致内存泄漏。

(c) 如果值型HasPtr类定义了析构函数但没有定义复制构造函数和赋值操作符，就使用编译器合成的复制构造函数和赋值操作符。因此，若将一个HasPtr对象赋值给另一HasPtr对象（或用一个HasPtr对象复制构造另一个HasPtr对象），则两个HasPtr对象的ptr成员值相同（即二者指向同一基础int对象）。当其中一个HasPtr对象被撤销之后，该基础int对象也被撤销，使得另一个HasPtr对象中的ptr成员指向一个不复存在的基础int对象，从而通过该HasPtr对象的ptr成员使用其指向的基础int对象会出现问题。而且当另一个HasPtr对象被撤销时，会因为析构函数对同一指针进行重复删除而出现内存访问非法的错误。此外，被赋值的HasPtr对象的ptr成员原来所指向的基础int对象不能再被访问，但也没有撤销。

### 习题13.28

对于如下的类，实现默认构造函数和必要的复制控制成员。

<p>(a) class TreeNode {      public:      // ...      private:      std::string value;      int count;      TreeNode *left;      TreeNode *right;  };</p>	<p>(b) class BinStrTree {      public:      // ...      private:      TreeNode *root;  };</p>
---	---

### 【解答】

(a) 对于TreeNode类，默认构造函数和必要的复制控制成员如下：

```
TreeNode::TreeNode() :
    count(0), left(0), right(0)
{ }

TreeNode::TreeNode(const TreeNode &orig) :
    value(orig.value)
{
    count = orig.count;
    if (orig.left)
        left = new TreeNode(*orig.left);
    else
        left = 0;
    if (orig.right)
        right = new TreeNode(*orig.right);
    else
        right = 0;
}

TreeNode::~TreeNode()
{
    if (left)
        delete left;
    if (right)
```

```
    delete right;
}
```

(b) 对于 BinStrTree 类, 默认构造函数和必要的复制控制成员如下:

```
BinStrTree::BinStrTree() : root(0)
{
}

BinStrTree::BinStrTree(const BinStrTree &orig)
{
    if (orig.root)
        root = new TreeNode(*orig.root);
    else
        root = 0;
}

BinStrTree::~BinStrTree()
{
    if (root)
        delete root;
}
```

# 第 14 章

## 重载操作符与转换

### 习题14.1

在什么情况下重载操作符与内置操作符不同？在什么情况下重载操作符与内置操作符相同？

#### 【解答】

重载操作符与内置操作符的不同之处在于：重载操作符必须具有至少一个类类型或枚举类型的操作数；重载操作符不保证操作数的求值顺序，例如，`&&`和`||`的重载版本失去了“短路求值”特性，两个操作数都要进行求值，而且不规定操作数的求值顺序。

重载操作符与内置操作符的相同之处在于：操作符的优先级、结合性及操作数数目均相同。

### 习题14.2

为`Sales_item`编写输入、输出、加以及复合赋值操作符的重载声明。

#### 【解答】

```
class Sales_item {
    friend std::istream& operator>>
        (std::istream&, Sales_item&);
    friend std::ostream& operator<<
        (std::ostream&, const Sales_item&);

public:
    Sales_item& operator+=(const Sales_item&);
};

Sales_item operator+(const Sales_item&, const Sales_item&);
```

其中，复合赋值操作符定义为`public`成员，输入和输出操作符需要访问`Sales_item`类的成员，所以需定义为`Sales_item`类的友元。加操作符可以用`public`成员`+=`来实现，所以无需定义为`Sales_item`类的友元。

### 习题14.3

解释如下程序，假定接受一个`string`参数的`Sales_item`构造函数不为`explicit`。解释如果该构造函数为`explicit`会怎样。

```
string null_book = "9-999-99999-9";
Sales_item item(cin);
item += null_book;
```

#### 【解答】

如果接受一个`string`参数的`Sales_item`构造函数不为`explicit`，则该程序段的行为如下：

- 执行语句 `string null_book = "9-999-99999-9";`, 首先调用接受一个 C 风格字符串形参的 `string` 构造函数, 创建一个临时的 `string` 对象, 然后, 使用 `string` 复制构造函数将 `null_book` 初始化为该临时 `string` 对象的副本。
- 执行语句 `Sales_item item(cin);`, 从标准输入设备读入数据, 创建 `Sales_item` 对象 `item`。
- 执行语句 `item += null_book;`, 首先调用接受一个 `string` 参数的 `Sales_item` 构造函数, 从 `null_book` 创建一个临时的 `Sales_item` 对象 (即将一个 `string` 对象隐式转换为一个 `Sales_item` 对象), 然后将该临时对象加至 `item`。

如果接受一个 `string` 参数的 `Sales_item` 构造函数为 `explicit`, 则不能进行从 `string` 对象到 `Sales_item` 对象的隐式转换, 语句 `item += null_book;` 会导致编译错误。

#### 习题14.4

`string` 和 `vector` 类都定义了一个重载的 `==`, 可用于比较这些类的对象。指出下面的表达式中应用了哪个 `==` 版本:

```
string s; vector<string> svec1, svec2;
"cobble" == "stone"
svec1[0] == svec2[0];
svec1 == svec2
```

#### 【解答】

表达式 `"cobble" == "stone"` 中应用了 C++ 语言内置的 `==` 版本。

表达式 `svec1[0] == svec2[0]` 中应用了 `string` 类所定义的 `==` 版本。

表达式 `svec1 == svec2` 中应用了 `vector` 类所定义的 `==` 版本。

#### 习题14.5

列出必须定义为类成员的操作符。

#### 【解答】

赋值 (`=`)、下标 (`[]`)、调用 (`()`) 和成员访问箭头 (`->`) 等操作符必须定义为类成员。

#### 习题14.6

解释下面操作符是否应该为类成员, 为什么?

(a) `+`      (b) `+=`      (c) `++`      (d) `->`      (e) `<<`      (f) `&&`      (g) `==`      (h) `()`

#### 【解答】

`+`、`<<` 和 `==` 操作符通常应定义为非成员函数, 但 `<<` 操作符通常需要访问类的数据成员, 所以一般应指定为类的友元。

`+=` 和 `++` 会改变对象的状态, 通常应定义为类成员。

`->` 和 `()` 必须定义为类成员, 否则会出现编译错误。

`&&` 一般对类类型操作数没有意义, 通常不进行重载; 如果一定要重载, 可重载为非成员函数。

#### 习题14.7

为下面的 `CheckoutRecord` 类定义一个输出操作符:

```

class CheckoutRecord {
public:
// ...
private:
    double book_id;
    string title;
    Date date_borrowed;
    Date date_due;
    pair<string, string> borrower;
    vector< pair<string, string>*> wait_list;
};

```

**【解答】**

```

ostream& operator << (ostream& out, const CheckoutRecord& c)
{
    out << c.book_id << "\t" << c.title << endl
        << "date borrowed: " << c.date_borrowed << endl
        << "date due: " << c.date_due << endl
        << "borrower: " << c.borrower.first << ", "
            << c.borrower.second << endl;
    out << "wait list: " << endl;
    for (vector< pair<string, string>*>::const_iterator
        iter = c.wait_list.begin(); iter != c.wait_list.end(); ++iter)
        out << "\t" << (*iter)->first << ", "
            << (*iter)->second << endl;
    return out;
}

```

注意，Date为用户自定义类，该类一般可以提供重载的<<操作符，所以直接使用该操作符即可。pair和vector为标准库模板类，没有提供重载的<<操作符，所以在CheckoutRecord类的输出操作符中需要逐个输出相应元素。另外，注意输出操作符<<应指定为CheckoutRecord类的友元，为此，需要在CheckoutRecord类的定义体内增加如下声明：

```
friend ostream& operator<<(ostream&, CheckoutRecord&);
```

**习题14.8**

12.4节的习题中，你编写了下面某个类的框架：

- |             |            |              |
|-------------|------------|--------------|
| (a) Book    | (b) Date   | (c) Employee |
| (d) Vehicle | (e) Object | (f) Tree     |

为所选择的类编写输出操作符。

**【解答】**

为习题12.20中所定义的Date类编写输出操作符如下：

```

ostream& operator << (ostream& out, const Date& d)
{
    out << d.year << "/" << d.month << "/" << d.day;
    return out;
}

```

注意，应将<<操作符指定为 Date 类的友元。这只是一个简单的定义，按中国习惯输出“年/月/日”格式的 Date。也许有人认为输出时应该考虑 Date 的合法性，但事实上一个 Date 对象的合法性由 Date 类的其他成员函数（如构造函数、其他日期操作等）来保证。每个存在的 Date 对象所表示的日

期都应该是合法的（即不可能存在诸如表示2006年8月32日之类的Date对象），所以输出操作符无需进行相关检查，直接输出数据成员即可。

### 习题14.9

给定下述输入，描述Sales\_item输入操作符的行为。

- (a) 0-201-99999-9 10 24.95
- (b) 10 24.95 0-201-99999-9

### 【解答】

- (a) 将形参Sales\_item对象的isbn成员设置为0-201-99999-9，units\_sold成员设置为10，revenue成员设置为249.5。
- (b) 首先将形参Sales\_item对象的isbn成员设置为10，然后因输入数据的类型不符合要求导致输入失败，从而if语句将形参Sales\_item对象复位为空Sales\_item对象：isbn成员为空string，units\_sold和revenue成员为0。

### 习题14.10

下述Sales\_item输入操作符有什么错误？

```
istream& operator>>(istream& in, Sales_item& s)
{
    double price;
    in >> s.isbn >> s.units_sold >> price;
    s.revenue = s.units_sold * price;
    return in;
}
```

如果将习题14.9中的数据作为输入，将会发生什么？

### 【解答】

该Sales\_item输入操作符的错误在于：没有对输入失败的情况进行处理，从而导致对象s可能处于不一致的状态。例如，有可能在成功地读入了一个新的isbn之后遇到了流错误，因此对象s原有的units\_sold和revenue成员没变，结果会将另一个isbn与对象s原有的units\_sold和revenue数据相关联。

如果将上题中的数据作为输入，则：(a) 的结果与上题相同；(b) 的结果为——将形参Sales\_item对象的isbn成员设置为10，units\_sold成员和revenue成员保持原值不变。

### 习题14.11

为14.2.1节习题中定义的CheckoutRecord类定义一个输入操作符，确保该操作符处理输入错误。

### 【解答】

将如下定义的>>操作符指定为CheckoutRecord类的友元：

```
istream& operator >> (istream& in, CheckoutRecord& c)
{
    in >> c.book_id >> c.title
    >> c.date_borrowed >> c.date_due
    >> c.borrower.first >> c.borrower.second;
    if (!in) {
```

```

        c = CheckoutRecord();
        return in;
    }

    c.wait_list.clear(); // 删除元素
    while (in) {
        pair<string, string> *ppa = new pair<string, string>;
        in >> ppa -> first >> ppa -> second;
        if (!in) {
            return in;
        }
        c.wait_list.push_back(ppa);
    }
    return in;
}

```

注意，对于自定义的Date类的成员，直接使用Date类提供的重载`>>`操作符进行输入；在输入vector成员时需先将vector中原有的元素全部删除；vector成员的元素为指针，需动态创建相应的pair对象；CheckoutRecord类的析构函数中要删除相应的pair对象。

### 习题14.12

编写Sales\_item操作符，用`+`进行实际加法，而`+=`调用`+`。与本节中操作符的实现方法相比较，讨论这个方法的缺点。

#### 【解答】

将如下定义的`+`操作符定义为sales\_item类的成员：

```

Sales_item Sales_item::operator+(const Sales_item& rhs)
{
    units_sold += rhs.units_sold;
    revenue += rhs.revenue;
    return *this;
}

```

将如下定义的`+=`操作符指定为 Sales\_item 类的友元：

```

Sales_item& operator+=(Sales_item& lhs, const Sales_item& rhs)
{
    lhs = lhs + rhs; // 调用+操作符
    return lhs;
}

```

该方法的缺点是：在`+=`操作中需要创建和撤销一个临时Sales\_item对象来保存`+`操作的结果，不如本节中给出的操作符实现方法简单有效。

### 习题14.13

如果有，你认为Sales\_item还应该支持哪些其他算术操作符？定义你认为该类应包含的那些。

#### 【解答】

从Sales\_item类的应用需求来看，也许该类还可以支持算术操作符`-`，相应地可以定义一个`-=`操作符，以支持相应的复合赋值并支持`-`操作符的实现。

其中复合赋值操作符`-=`应定义为Sales\_item类的成员，而算术操作符`-`应定义为非成员操作符，它们均用于对ISBN相同的Sales\_item对象进行操作，定义如下：

```

Sales_item& Sales_item::operator-=(const Sales_item& rhs)
{
    units_sold -= rhs.units_sold;
    revenue -= rhs.revenue;
    return *this;
}

Sales_item operator-(const Sales_item& lhs, const Sales_item& rhs)
{
    Sales_item ret(lhs);
    ret -= rhs;
    return ret;
}

```

**习题14.14**

定义一个赋值操作符，将isbn赋值给Sales\_item对象。

**【解答】**

```

Sales_item& Sales_item::operator=(const string& str)
{
    isbn = str;
    return *this;
}

```

注意，赋值操作符必须定义为类的成员函数，且一般应返回左操作数的引用。

**习题14.15**

为14.2.1节习题中介绍的CheckoutRecord类定义赋值操作符。

**【解答】**

```

CheckoutRecord&
CheckoutRecord::operator=(const CheckoutRecord& rhs)
{
    book_id = rhs.book_id;
    title = rhs.title;
    date_borrowed = rhs.date_borrowed;
    date_due = rhs.date_due;
    borrower = rhs.borrower;

    // 对vector成员进行赋值
    wait_list.clear(); // 删除原有元素
    for (vector< pair<string, string>*>::const_iterator
        it = rhs.wait_list.begin();
        it != rhs.wait_list.end(); ++it) {
        pair<string, string> *ppa = new pair<string, string>;
        *ppa = **it; // 复制指针所指向的pair对象
        wait_list.push_back(ppa);
    }
    return *this;
}

```

注意，在对vector成员进行赋值时，要实现“深复制”，即不是简单地复制vector元素的值（为指针），而要对vector元素所指向的pair对象进行复制，以避免出现内存访问问题（见13.5节）。

**习题14.16**

CheckoutRecord类还应该定义其他赋值操作符吗？如果是，解释哪些类型应该用作操作数并解释为什么。为这些类型实现赋值操作符。

**【解答】**

从应用角度考虑，CheckoutRecord类还可以定义其他赋值操作符。例如，当读者办理续借手续的时候，可以通过赋值操作修改相应CheckoutRecord对象的date\_due成员，该赋值操作使用Date类型的操作数；当对于某本书有新的读者预约的时候，可以通过赋值操作在相应CheckoutRecord对象的wait\_list中增加该读者，该赋值操作使用pair<string, string>类型的操作数。

这两个赋值操作符可定义如下：

```
// 设置新的date_due
CheckoutRecord& CheckoutRecord::operator=(const Date& new_due)
{
    date_due = new_due;
    return *this;
}

// 增加一个等待者
CheckoutRecord& CheckoutRecord::operator=(const
                                         std::pair<string, string>& awaiter)
{
    pair<string, string> *ppa = new pair<string, string>;
    *ppa = awaiter; // 复制形参pair对象
    wait_list.push_back(ppa);
    return *this;
}
```

**习题14.17**

14.2.1节习题中定义了一个CheckoutRecord类，为该类定义一个下标操作符，从等待列表中返回一个名字。

```
pair<string, string>& CheckoutRecord::operator[]
(const vector< pair<string, string>*>::size_type index)
{
    return *wait_list.at(index); // 使用at可检查下标是否越界
}

const pair<string, string>& CheckoutRecord::operator[]
(const vector< pair<string, string>*>::size_type index) const
{
    return *wait_list.at(index); // 使用at可检查下标是否越界
}
```

注意，下标操作符必须定义为类成员函数，且返回引用以便可以用在赋值操作符的任意一边。类定义下标操作符时，一般需定义两个版本，即返回引用的非const成员及返回const引用的const成员，以便可以对const和非const对象使用下标；可以对下标是否越界进行检查（这与内置下标操作符的语言有所不同），以避免对内存的非法访问。

**习题14.18**

讨论用下标操作符实现这个操作的优缺点。

**【解答】**

优点：使用简单。

缺点：操作的语义不够清楚。下标操作符适合于表示“获取容器中的元素”这一语义，此处的CheckoutRecord对象并不是一个通常意义上的容器，而“等待者”更不是“CheckoutRecord容器中的一个元素”，因此使用下标操作符不能清楚地表示“获得一个等待者”这一应用语义。

**习题14.19**

提出另一种方法定义这个操作。

**【解答】**

可将这个操作定义为普通的成员函数，例如，`pair<string, string>& get_awaiter (const size_t index)` 及 `const pair<string, string>& get_awaiter (const size_t index) const`。

**习题14.20**

在ScreenPtr类的概略定义中，声明但没有定义赋值操作符。请实现ScreenPtr赋值操作符。

**【解答】**

```
ScreenPtr& ScreenPtr::operator=(const ScreenPtr &rhs)
{
    ++rhs.ptr->use; // 右操作数对象的使用计数加1
    if (--ptr->use == 0)
        delete ptr; // 如果左操作数对象的使用计数减至0，则删除该对象
    ptr = rhs.ptr;

    return *this;
}
```

**习题14.21**

定义一个类，该类保存一个指向ScreenPtr的指针。为该类定义一个重载的箭头操作符。

**【解答】**

可定义如下NoName类：

```
class NoName {
public:
    NoName(Screen *p): ptr(new ScreenPtr(p)) { }

    // 重载的箭头操作符
    ScreenPtr operator->()
    {
        return *ptr;
    }

    const ScreenPtr operator->() const
    {
        return *ptr;
    }
private:
    ScreenPtr *ptr;
};
```

注意，箭头操作符必须定义为类成员函数；重载的箭头操作符不接受显式形参，且必须返回指向

类类型的指针，或者返回定义了箭头操作符的类类型对象；需要定义箭头操作符的 const 和非 const 版本，以便可以对 const 和非 const 对象使用箭头操作符。

### 习题14.22

智能指针可能应该定义相等操作符和不等操作符，以便测试两个指针是否相等或不等。将这些操作加入到 ScreenPtr 类。

#### 【解答】

ScreenPtr 类的相等操作符和不等操作符可定义如下：

```
inline bool
operator==(const ScreenPtr &lhs, const ScreenPtr &rhs)
{
    return lhs.ptr == rhs.ptr;
}

inline bool
operator!=(const ScreenPtr &lhs, const ScreenPtr &rhs)
{
    return !(lhs == rhs); // 根据==操作符定义!=
}
```

注意，关系操作符一般定义为普通非成员函数；此处定义的==操作符应指定为 ScreenPtr 类的友元；通常==和!=操作符应相互联系起来定义（此处定义==操作符完成实际的对象比较工作，而!=操作符则调用==操作符，亦可定义!=操作符完成实际的对象比较工作，而==操作符则调用!=操作符）。

### 习题14.23

CheckedPtr 类表示指向数组的指针。为该类重载下标操作符和解引用操作符。使操作符确保 CheckedPtr 有效：它应该不可能对超出数组末端的元素进行解引用或索引。

#### 【解答】

```
int& CheckedPtr::operator[](const size_t index)
{
    if (beg + index >= end) // end 指向数组最后一个元素的下一个位置
        throw out_of_range
            ("invalid index");
    return *(beg + index);
}

const int& CheckedPtr::operator[](const size_t index) const
{
    if (beg + index >= end)
        throw out_of_range
            ("invalid index");
    return *(beg + index);
}

int& CheckedPtr::operator*()
{
    if (curr == end)
        throw out_of_range
            ("invalid current pointer");
    return *curr;
}
```

```

const int& CheckedPtr::operator*() const
{
    if (curr == end)
        throw out_of_range
            ("invalid current pointer");
    return *curr;
}

```

注意，下标操作符必须定义为类成员函数，解引用操作符一般也可以定义为类成员函数；提供操作符的const和非const版本，以便对const和非const对象均可使用。

### 习题14.24

习题14.23中定义的解引用操作符或下标操作符，是否也应该检查对数组起点之前的元素进行的解引用或索引？解释你的答案。

#### 【解答】

对于下标操作符，也应该检查对数组起点之前的元素进行的索引，因为用户有可能给出小于0的数作为索引。当给定的index为负数时，编译器在调用下标操作符时会将之隐式转换为无符号的size\_t类型，所以不会出现编译错误。如果不进行检查，会导致运行时错误。

可将下标操作符的定义修改如下：

```

int& CheckedPtr::operator[](const size_t index)
{
    // end指向数组最后一个元素的下一个位置，beg指向数组的第一个元素
    if (beg + index >= end || beg + index < beg)
        throw out_of_range
            ("invalid index");
    return *(beg + index);
}

const int& CheckedPtr::operator[](const size_t index) const
{
    if (beg + index >= end || beg + index < beg)
        throw out_of_range
            ("invalid index");
    return *(beg + index);
}

```

注意，黑体部分为所做的修改。

解引用操作符返回curr所指向的数组元素。在创建CheckedPtr对象时将curr初始化为指向数组的第一个元素，且只有--操作符会对curr进行减量操作，而--操作符已经可以保证curr不会小于beg，所以解引用操作符无需再检查对数组起点之前的元素进行的解引用。

### 习题14.25

为了表现得像数组指针，CheckedPtr类应实现相等和关系操作符，以便确定两个CheckedPtr对象是否相等，或者一个小于另一个，诸如此类。为CheckedPtr类增加这些操作。

#### 【解答】

可为CheckedPtr类定义==、!=、>、>=、<、<=等操作符如下：

```

// 相等操作符
bool operator==(const CheckedPtr& lhs, const CheckedPtr& rhs)

```

```

{
    return lhs.beg == rhs.beg && lhs.end == rhs.end
        && lhs.curr == rhs.curr;
}

bool operator!=(const CheckedPtr& lhs, const CheckedPtr& rhs)
{
    return !(lhs == rhs);
}

// 关系操作符
bool operator<(const CheckedPtr& lhs, const CheckedPtr& rhs)
{
    return lhs.beg == rhs.beg && lhs.end == rhs.end
        && lhs.curr < rhs.curr;
}

bool operator<=(const CheckedPtr& lhs, const CheckedPtr& rhs)
{
    return !(lhs > rhs);
}

bool operator>(const CheckedPtr& lhs, const CheckedPtr& rhs)
{
    return lhs.beg == rhs.beg && lhs.end == rhs.end
        && lhs.curr > rhs.curr;
}

bool operator>=(const CheckedPtr& lhs, const CheckedPtr& rhs)
{
    return !(lhs < rhs);
}

```

注意，这些操作符一般应定义为非成员函数，如果操作需要访问CheckedPtr类的数据成员，则应指定为CheckedPtr类的友元；对CheckedPtr对象的比较仅在两个对象指向同一数组时才有意义；这些操作符可以相互联系起来定义。

### 习题14.26

为CheckedPtr类定义加法或减法，以便这些操作符实现指针运算（4.2.4节）。

#### 【解答】

可定义如下加法及减法操作符：

```

CheckedPtr operator+(const CheckedPtr& lhs, const size_t n)
{
    CheckedPtr temp(lhs);
    temp.curr += n;
    if (temp.curr > temp.end)
        throw out_of_range
            ("too large n");
    return temp;
}

CheckedPtr operator-(const CheckedPtr& lhs, const size_t n)
{
    CheckedPtr temp(lhs);
    temp.curr -= n;
    if (temp.curr < temp.beg)

```

```

        throw out_of_range
            ("too large n");
    return temp;
}

ptrdiff_t operator-(CheckedPtr& lhs, CheckedPtr& rhs)
{
    // 检查左右操作数对象是否指向同一数组
    if (!(lhs.beg == rhs.beg && lhs.end == rhs.end))
        throw out_of_range
            ("can't subtract");
    return lhs.curr - rhs.curr;
}

```

注意，加法和减法操作符通常定义为非成员函数，应将这些操作符指定为类的友元；两个指针相减仅在两个指针指向同一数组时才有意义。

加法和减法操作符的实现中使用了复制构造函数，可定义如下：

```

// 复制构造函数
CheckedPtr(const CheckedPtr& c): beg(c.beg), end(c.end),
                                curr(c.curr) { }

```

### 习题14.27

讨论允许将空数组实参传给CheckedPtr构造函数的优缺点。

#### 【解答】

优点：构造函数的定义简单。

缺点：导致所构造的CheckedPtr对象有可能没有指向有效的数组，从而失去其使用价值。

比较完善的实现方法应该在构造函数中对参数进行检查（保证beg<end）。

### 习题14.28

没有定义自增和自减操作符的const版本，为什么？

#### 【解答】

因为对const对象不能使用自增和自减操作符。

### 习题14.29

我们也没有实现箭头操作符，为什么？

#### 【解答】

因为该CheckedPtr类所指向的是int型数组，int型为内置类型（非类类型），没有显式定义的成员函数，所以不需要通过CheckedPtr对象使用箭头操作符来访问int对象的成员，也无需为CheckedPtr类定义箭头操作符。

### 习题14.30

定义一个CheckedPtr版本，保存Screen数组。为该类实现重载的自增、自减、解引用和箭头等操作符。

#### 【解答】

```
// 智能指针：对元素访问进检查，如果试图访问不存在的元素，
```

```

// 则抛出out_of_range异常
// 用户负责分配和释放数组
class CheckedPtr {
public:
    // 没有默认构造函数, CheckedPtr对象必须绑定到一个对象(数组)
    CheckedPtr(Screen *b, Screen *e): beg(b), end(e), curr(b) { }

    // 解引用操作
    Screen& operator*();
    const Screen& operator*() const;

    // 自增和自减操作
    // 前缀式
    CheckedPtr& operator++();
    CheckedPtr& operator--();
    // 后缀式
    CheckedPtr operator++(int);
    CheckedPtr operator--(int);

    // 箭头操作
    Screen* operator->();
    const Screen* operator->() const;
private:
    Screen* beg;    // 指向数组起点的指针
    Screen* end;    // 数组结尾的下一个位置
    Screen* curr;   // 在数组中的当前位置
};

Screen& CheckedPtr::operator*()
{
    if (curr == end)
        throw out_of_range
            ("invalid current pointer");
    return *curr;
}

const Screen& CheckedPtr::operator*() const
{
    if (curr == end)
        throw out_of_range
            ("invalid current pointer");
    return *curr;
}

// 前缀式: 返回增量/减量之后的对象
CheckedPtr& CheckedPtr::operator++()
{
    if (curr == end)
        throw out_of_range
            ("increment past the end of CheckedPtr");
    ++curr;
    return *this;
}

CheckedPtr& CheckedPtr::operator--()
{
    if (curr == beg)
        throw out_of_range
            ("decrement past the beginning of CheckedPtr");
    --curr;
    return *this;
}

```



```

}

// 后缀式：对象进行增量/减量，返回未改变的值
CheckedPtr CheckedPtr::operator++(int)
{
    // 此处无需检查curr，由被调用的前自增操作进行检查
    CheckedPtr ret(*this);      // 保存当前值
    ++*this;                   // 调用前自增操作，进行检查
    return ret;                // 返回被保存状态
}

CheckedPtr CheckedPtr::operator--(int)
{
    // 此处无需检查curr，由被调用的前自减操作进行检查
    CheckedPtr ret(*this);      // 保存当前值
    --*this;                   // 调用前自减操作，进行检查
    return ret;                // 返回被保存状态
}

Screen* CheckedPtr::operator->()
{
    return curr;
}

const Screen* CheckedPtr::operator->() const
{
    return curr;
}

```

### 习题14.31

定义一个函数对象执行“如果……则……否则”操作：该函数对象应接受三个形参；它应该测试第一个形参；如果测试成功，则返回第二个形参；否则，就返回第三个形参。

#### 【解答】

可定义如下函数对象：

```

class ifThenElse {
public:
    int operator()(int val1, int val2, int val3)
    {
        return val1 ? val2 : val3;
    }
};

```

该函数对象接受三个int型形参，并测试第一个形参是否为true(非0)值。如果测试成功，就返回第二个形参；否则，就返回第三个形参。

如果希望该函数对象能接受任意类型的形参，可将其定义为类模板（见16.1.2节）：

```

template<typename T>
class ifThenElse {
public:
    T operator()(T val1, T val2, T val3)
    {
        return val1 ? val2 : val3;
    }
};

```



**习题14.32**

一个重载的函数调用操作符可以接受多少个操作数？

**【解答】**

与函数调用可以接受的操作数一样，为0个或多个（任意个）。

**习题14.33**

使用标准库算法和GT\_cls类，编写一个程序查找序列中第一个比指定值大的元素。

**【解答】**

如果使用vector存放int型序列，则可编写程序如下：

```
// 14-33.cpp
// 定义函数对象GT_cls,
// 使用标准库算法find_if和GT_cls类,
// 查找vector对象中第一个比指定值大的元素
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

class GT_cls {
public:
    GT_cls(int val = 0): bound(val) { }
    bool operator() (const int &ival)
    {
        return ival > bound;
    }
private:
    int bound;
};

int main()
{
    vector<int> ivec;

    int ival;
    cout << "Enter numbers(Ctrl+Z to end): " << endl;
    // 读入vector元素
    while (cin >> ival)
        ivec.push_back(ival);

    cin.clear(); // 使流重新有效

    int spval;
    cout << "Enter a specified value: " << endl;
    // 读入用于查找的指定值
    cin >> spval;

    // 查找第一个大于指定值的vector元素
    vector<int>::iterator iter;
    iter = find_if(ivec.begin(), ivec.end(), GT_cls(spval));

    // 输出结果
    if (iter != ivec.end())
        cout << "the first element that is larger than "
            << spval << " : "
            << *iter << endl;
}
```



```

    else
        cout << "no element that is larger than "
        << spval << endl;

    return 0;
}

```

注意，14.8.1节中定义的GT\_cls类不符合此处的需要，因此需另外定义GT\_cls类。为了提高GT\_cls类的通用性，可以将其定义为模板类（见习题14.31的解答）。

### 习题14.34

编写类似于GT\_cls的函数对象类，但测试两个值是否相等。使用该对象和标准库算法编写程序，替换序列中给定值的所有实例。

#### 【解答】

如果使用vector存放int型序列，则可编写程序如下：

```

// 14-34.cpp
// 定义函数对象EQ_cls，测试两个值是否相等：
// 使用标准库算法replace_if和EQ_cls类，
// 替换vector对象中给定值的所有实例
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

class EQ_cls {
public:
    EQ_cls(int val = 0): spValue(val) { }
    bool operator() (const int &ival)
    {
        return ival == spValue;
    }
private:
    int spValue;
};

int main()
{
    vector<int> ivec;
    int ival;
    cout << "Enter numbers(Ctrl+Z to end): " << endl;
    // 读入vector元素
    while (cin >> ival)
        ivec.push_back(ival);

    cin.clear(); // 使流重新有效

    int replacedVal, newVal;
    // 读入需替换的指定值
    cout << "Enter a value that will be replaced: " << endl;
    cin >> replacedVal;

    // 读入用于替换的新值
    cout << "Enter a new value: " << endl;
    cin >> newVal;

    // 替换等于指定值的vector元素
    replace_if(ivec.begin(), ivec.end(), EQ_cls(replacedVal),

```

```

        newVal);

// 输出替换后的vector对象以进行对比
cout << "new sequence: " << endl;
for (vector<int>::iterator iter = ivec.begin();
     iter != ivec.end(); ++iter)
    cout << *iter << " ";
}

return 0;
}

```

注意，为了提高 EQ\_cls 类的通用性，可以将其定义为模板类（见习题 14.31 的解答）。

### 习题14.35

编写类似于 GT\_cls 的类，但测试给定 string 对象的长度是否与其边界相匹配。使用该对象重写 11.2.3 节中的程序，以便报告输入中有多少个单词的长度在 1 到 10 之间（含 1 和 10）。

#### 【解答】

```

// 14-35.cpp
// 定义BET_cls类，用于测试给定string对象的长度是否与其边界相匹配；
// 读入文本文件，
// 使用BET_cls类，统计长度在1到10之间（含1和10）的单词的数目
#include <iostream>
#include <fstream>
#include <vector>
#include <algorithm>
#include <string>
using namespace std;

// 用于将单词按长度排序的比较函数
bool isShorter(const string &s1, const string &s2)
{
    return s1.size() < s2.size();
}

// 测试给定string对象的长度是否与其边界相匹配
class BET_cls {
public:
    BET_cls(string::size_type len1, string::size_type len2 )
    {
        if (len1 < len2) {
            minLength = len1;
            maxLength = len2;
        }
        else {
            minLength = len2;
            maxLength = len1;
        }
    }

    bool operator() (const string &s)
    {
        return s.size() >= minLength && s.size() <= maxLength;
    }
private:
    string::size_type minLength;
    string::size_type maxLength;
};

// 如果ctr不为1，返回word的复数版本

```

```

string make_plural(size_t ctr, const string &word,
                   const string &ending)
{
    return (ctr == 1) ? word : word + ending;
}

// main函数接受文件名为参数
int main(int argc, char **argv)
{
    // 检查命令行参数个数
    if (argc < 2) {
        cerr << "No input file!" << endl;
        return EXIT_FAILURE;
    }

    // 打开输入文件
    ifstream inFile;
    inFile.open(argv[1]);
    if (!inFile){
        cerr << "Can not open input file!" << endl;
        return EXIT_FAILURE;
    }

    vector<string> words;
    string word;
    // 读入要分析的输入序列，并存放在vector容器中
    while (inFile >> word)
        words.push_back(word);

    // 对输入排序以便去除重复的单词
    sort(words.begin(), words.end());

    // 使用算法unique对元素重新排序并返回一个迭代器,
    // 表示无重复的单词范围的结束
    // erase操作使用该迭代器删除输入序列中重复的单词
    words.erase(unique(words.begin(), words.end()),
                words.end());

    // 将单词按长度排序，等长的单词按字典顺序排列
    stable_sort(words.begin(), words.end(), isShorter);

    // 计算并输出长度在1到10之间（含1和10）的单词的数目
    vector<string>::size_type wc = count_if (words.begin(),
                                              words.end(), BET_cls(1,10));
    cout << wc << " " << make_plural(wc, "word", "s")
        << " are of sizes 1 through 10 inclusive" << endl;
}

return 0;
}

```

**习题14.36**

修改前一程序以报告长度在1~9之间以及10以上的单词的数目。

**【解答】**

```

// 14-36.cpp
// 定义BET_cls类，用于测试给定string对象的长度是否与其边界相匹配;
// 定义GT_cls类，用于测试给定string对象的长度是否在给定值以上;
// 读入文本文件，使用BET_cls类和GT_cls类,
// 统计长度在1到9之间（含1和9）及10以上的单词的数目

```

```

#include <iostream>
#include <fstream>
#include <vector>
#include <algorithm>
#include <string>
using namespace std;

// 用于将单词按长度排序的比较函数
bool isShorter(const string &s1, const string &s2)
{
    return s1.size() < s2.size();
}

// 测试给定string对象的长度是否与其边界相匹配
class BET_cls {
public:
    BET_cls(string::size_type len1, string::size_type len2 )
    {
        if (len1 < len2) {
            minLength = len1;
            maxLength = len2;
        }
        else {
            minLength = len2;
            maxLength = len1;
        }
    }

    bool operator() (const string &s)
    {
        return s.size() >= minLength && s.size() <= maxLength;
    }
private:
    string::size_type minLength;
    string::size_type maxLength;
};

// 测试给定string对象的长度是否在给定值以上
class GT_cls {
public:
    GT_cls(size_t val = 0): bound(val) { }
    bool operator()(const string &s)
    {
        return s.size() >= bound;
    }
private:
    std::string::size_type bound;
};

// 如果ctr不为1，返回word的复数版本
string make_plural(size_t ctr, const string &word,
                    const string &ending)
{
    return (ctr == 1) ? word : word + ending;
}

// main函数接受文件名为参数
int main(int argc, char **argv)
{
    // 检查命令行参数个数
    if (argc < 2) {
        cerr << "No input file!" << endl;
}

```

```

        return EXIT_FAILURE;
    }

    // 打开输入文件
    ifstream inFile;
    inFile.open(argv[1]);
    if (!inFile){
        cerr << "Can not open input file!" << endl;
        return EXIT_FAILURE;
    }

    vector<string> words;
    string word;
    // 读入要分析的输入序列，并存放在vector容器中
    while (inFile >> word)
        words.push_back(word);

    // 对输入排序以便去除重复的单词
    sort(words.begin(), words.end());

    // 使用算法unique对元素重新排序并返回一个迭代器,
    // 表示无重复的单词范围的结束
    // erase操作使用该迭代器删除输入序列中重复的单词
    words.erase(unique(words.begin(), words.end()),
                words.end());

    // 计算并输出长度在1到9之间(含1和9)的单词的数目
    vector<string>::size_type wc =
        count_if (words.begin(), words.end(), BET_cls(1,9));
    cout << wc << " " << make_plural(wc, "word", "s")
        << " are of sizes 1 through 9 inclusive" << endl;

    // 计算并输出长度在10以上(含10)的单词的数目
    wc = count_if (words.begin(), words.end(), GT_cls(10));
    cout << wc << " " << make_plural(wc, "word", "s")
        << " 10 characters or longer" << endl;

    return 0;
}

```

**习题14.37**

使用标准库函数对象和函数适配器，定义一个对象用于：

- (a) 查找大于1024的所有值。
- (b) 查找不等于pooh的所有字符串。
- (c) 将所有值乘以2。

**【解答】**

可分别定义如下函数对象：

- (a) bind2nd(greater<int>(), 1024)
- (b) bind2nd(not\_equal\_to<string>(), "pooh")
- (c) bind2nd(multiplies<int>(), 2)

可如下编写程序来使用这些函数对象以完成所需功能：

```

// 14-37.cpp
// 使用标准库函数对象和函数适配器，定义三个对象分别用于：
// (a) 查找大于1024的所有值。
// (b) 查找不等于"pooh"的所有字符串。
// (c) 将所有值乘以2。

```

```

// 使用vector容器存放序列
#include <iostream>
#include <functional>
#include <algorithm>
#include <string>
#include <vector>
using namespace std;

int main()
{
    const int ARR_SIZE = 7;
    int ia[ARR_SIZE] = {1, 1025, 2, 1026, 1030, 3, 1048};
    vector<int> ivec(ia, ia+ARR_SIZE);
    string sa[ARR_SIZE] = {"many", "mach", "that", "pooh", "this", "pooh", "happy"};
    vector<string> svec(sa, sa+ARR_SIZE);

    // (a) 查找大于1024的所有值
    cout << "all values that are greater than 1024: " << endl;
    vector<int>::iterator iter = ivec.begin();
    // 使用bind2nd函数适配器将greater对象的右操作数绑定为1024
    while ((iter = find_if(iter, ivec.end(),
                           bind2nd(greater<int>(), 1024)))
           != ivec.end()) { // 找到了下一个大于1024的元素
        // 输出元素
        cout << *iter << ' ';
        // iter加1以便在剩余元素中进行查找
        ++iter;
    }

    // (b) 查找不等于"pooh"的所有字符串
    cout << endl << "all strings that are not equal to pooh: "
        << endl;
    vector<string>::iterator it = svec.begin();
    // 使用bind2nd函数适配器将not_equal_to对象的右操作数绑定为"pooh"
    while ((it = find_if(it, svec.end(), bind2nd(not_equal_to<string>(), "pooh")))
           != svec.end()) { // 找到了下一个不等于"pooh"的元素
        // 输出元素
        cout << *it << ' ';
        // it加1以便在剩余元素中进行查找
        ++it;
    }

    // (c) 将所有值乘以2
    // 使用bind2nd函数适配器将multiplies对象的右操作数绑定为2
    transform(ivec.begin(), ivec.end(), ivec.begin(),
              bind2nd(multiplies<int>(), 2));
    // 输出元素
    cout << endl << "all values multiplied by 2: " << endl;
    for (vector<int>::iterator it2 = ivec.begin();
         it2 != ivec.end(); ++it2)
        cout << *it2 << ' ';

    return 0;
}

```

### 习题14.38

最后一个count\_if调用中，用not1将bind2nd(less\_equal<int>(), 10)的结果求反。为什么使用not1而不用not2？

#### 【解答】

not1用于将一元函数对象的真值求反，not2用于将二元函数对象的真值求反，函数适配器bind2nd

将less\_equal对象的右操作数绑定到10，从而将less\_equal对象由二元函数对象转换为一元函数对象，所以只能用not1对bind2nd(less\_equal<int>(), 10)的结果进行求反。

### 习题14.39

使用标准库函数对象代替GT\_cls来查找指定长度的单词。

#### 【解答】

可编写程序如下：

```
// 14-39.cpp
// 读入文本文件，使用标准库函数对象greater_equal,
// 统计长度在10以上（含10）的单词的数目
#include <iostream>
#include <fstream>
#include <vector>
#include <algorithm>
#include <functional>
#include <string>
using namespace std;

// 用于将单词按长度排序的比较函数
bool isShorter(const string &s1, const string &s2)
{
    return s1.size() < s2.size();
}

// 如果ctr不为1，返回word的复数版本
string make_plural(size_t ctr, const string &word,
                    const string &ending)
{
    return (ctr == 1) ? word : word + ending;
}

// main函数接受文件名为参数
int main(int argc, char **argv)
{
    // 检查命令行参数个数
    if (argc < 2) {
        cerr << "No input file!" << endl;
        return EXIT_FAILURE;
    }

    // 打开输入文件
    ifstream inFile;
    inFile.open(argv[1]);
    if (!inFile){
        cerr << "Can not open input file!" << endl;
        return EXIT_FAILURE;
    }

    vector<string> words;
    string word;
    // 读入要分析的输入序列，并存放在vector容器中
    while (inFile >> word)
        words.push_back(word);

    // 对输入排序以便去除重复的单词
    sort(words.begin(), words.end());
}
```

```

// 使用算法unique对元素重新排序并返回一个迭代器,
// 表示无重复的单词范围的结束
// erase操作使用该迭代器删除输入序列中重复的单词
words.erase(unique(words.begin(), words.end()),
             words.end());

// 计算并输出长度在10以上(含10)的单词的数目
// 因为没有可用于直接比较字符串长度的标准库函数对象,
// 所以不能直接使用count_if算法,
// 改用greater_equal对象循环实现单词计数
vector<string>::size_type wc = 0;
greater_equal<string::size_type> greq;
for (vector<string>::iterator iter = words.begin();
     iter != words.end(); ++iter) {
    if (greq(iter->size(), 10)) {
        ++wc;
    }
}
cout << wc << " " << make_plural(wc, "word", "s")
    << " 10 characters or longer" << endl;
return 0;
}

```

**习题14.40**

编写可将sales\_item对象转换为string类型和double类型的操作符。你认为这些操作符应返回什么值？你认为定义这些操作符是个好办法吗？解释你的结论。

**【解答】**

将Sales\_item对象转换为string类型和double类型的操作符的代码如下：

```

Sales_item::operator string() const
{
    return isbn;
}

Sales_item::operator double() const
{
    return revenue;
}

```

定义这些操作符并不是个好办法，因为一般不必在需要string类型和double类型对象的地方使用sales\_item对象，这种用法没有太大的实际意义。

**习题14.41**

解释这两个转换操作符之间的不同：

```

class Integral {
public:
    operator const int() const;
    operator int() const;
};1

```

这两个转换操作符是否太严格了？如果是，怎样使得转换更通用一些？

<sup>1</sup> 此处英文原书代码有误：转换操作符的定义以operator开头。

**【解答】**

这两个转换操作符之间的不同之处在于：前者将对象转换为const int值（int型const变量），而后者将对象转换为int值（int型变量）；前者太严格了，只能用于可以使用const int值的地方；将前者去掉只保留后者，即可使得转换更为通用。（事实上，如果这两个转换操作符同时存在，则在既可使用int型const变量又可使用int型变量的情况下，会因编译器无法作出抉择而产生错误。）

**习题14.42**

为14.2.1节习题中的CheckoutRecord类定义到bool的转换操作符。

**【解答】**

```
CheckoutRecord::operator bool() const
{
    return wait_list.empty();
}
```

**习题14.43**

解释bool转换操作符做了什么。这是这个CheckoutRecord类型转换唯一可能的含义吗？解释你是否认为这个转换是一种转换操作的良好使用。

**【解答】**

上述定义的转换操作符将等待列表为空的CheckoutRecord对象转换为bool真值（true），将等待列表不为空的CheckoutRecord对象转换为bool假值（false）。

这并不是这个CheckoutRecord类型转换唯一可能的含义，也可以将bool转换操作符定义为其他含义，如返回当前日期是否已超过date\_due的判断结果（可用于判断某本借出的书是否已过期）。

上述定义的转换操作符可用于判断是否有读者在等待借阅这本书。在应用中，当某本书被归还时，可以用相应的CheckoutRecord对象作为判断条件以确定是否通知该书的等待者。因此这个转换可算是一种转换操作的良好使用。

**习题14.44**

为下述每个初始化列出可能的类类型转换序列。每个初始化的结果是什么？

```
class LongDouble {
public:
    operator double();
    operator float();
};

LongDouble ldObj;
(a) int ex1 = ldObj;   (b) float ex2 = ldObj;
```

**【解答】**

(a) 有二义性：因为既可以先使用从LongDouble到double的转换操作，再使用从double到int的标准转换，也可以先使用从LongDouble到float的转换操作，再使用从float到int的标准转换，二者没有优劣之分。

1. 此代码英文原书有误：未将转换操作指定为public成员，将导致无法使用转换操作。

(b) 使用从 LongDouble 到 float 的转换操作，将 lobj 对象转换为 float 值用于初始化 ex2。

### 习题14.45

哪个 calc() 函数是如下函数调用的最佳可行函数？列出调用每个函数所需的转换序列，并解释为什么所选定的就是最佳可行函数。

```
class LongDouble {
public:
    LongDouble(double);
    // ...
};

void calc(int);
void calc(LongDouble);
double dval;
calc(dval); // which function?
```

### 【解答】

最佳可行函数为 void calc(int)。

调用 void calc(int) 所需的转换为：将实参 dval 由 double 类型转换为 int 类型（标准转换）。

调用 void calc(LongDouble) 所需的转换为：将实参 dval 由 double 类型转换为 LongDouble 类型（使用 LongDouble 类的构造函数，为类类型转换）。

因为标准转换优于类类型转换（见 7.8.4 节），所以 void calc(int) 为最佳可行函数。

### 习题14.46

对于 main 中的加操作，哪个 operator+ 是最佳可行函数？列出候选函数、可行函数以及对每个可行函数中实参的类型转换。

```
class Complex {
public:
    Complex(double);
    // ...
};

class LongDouble {
    friend LongDouble operator+(LongDouble&, int);
public:
    LongDouble(int);
    operator double();
    LongDouble operator+(const Complex &);3
    // ...
};
LongDouble operator+(const LongDouble &, double);

LongDouble ld(16.08);
double res = ld + 15.05; // which operator+ ?
```

### 【解答】

候选函数包括：

(1) 内置的 + 操作符。

1. 此处英文原书有误：缺少冒号。

2. 此处英文原书有误：未将构造函数指定为 public 成员，将导致无法使用构造函数。

3. 此处英文原书有误： complex 应为 Complex。

(2) LongDouble operator+(LongDouble&, int); (LongDouble类的友元)。

(3) LongDouble LongDouble::operator+(const Complex &);。

(4) LongDouble operator+(const LongDouble &, double); (全局函数)。

调用(1)所需的类型转换为：将ld从LongDouble类型转换为double类型（可使用LongDouble类中定义的转换操作，为类类型转换）。

调用(2)所需的类型转换为：将15.05从double类型转换为int类型（可使用标准转换）。

调用(3)所需的类型转换为：将15.05从double类型转换为Complex类型（可使用Complex类的构造函数，为类类型转换）。

调用(4)无需进行类型转换。

因此，4个候选函数均为可行函数。因为调用(4)无需进行类型转换，所以该函数为最佳可行函数。



# 第 15 章

## 面向对象编程

### 习题15.1

什么是虚成员？

#### 【解答】

所谓“虚成员”就是其声明中在返回类型前带有关键字“virtual”的类成员函数。C++中基类通过将成员函数指定为虚成员来指出希望派生类重定义的那些函数。除了构造函数外，任意非static成员函数都可以为虚成员。

### 习题15.2

给出protected访问标号的定义。它与private有何不同？

#### 【解答】

protected访问标号称为“受保护的访问标号”，定义在protected标号之后的成员称为protected成员，可以被类成员、友元和派生类成员（非友元）访问，而类的普通用户则不能访问protected成员。

protected访问标号与private的区别在于：由protected标号指定的成员可以被（非友元）派生类成员访问，而由private标号指定的成员不能被（非友元）派生类成员访问。

### 习题15.3

定义自己的Item\_base类版本。

#### 【解答】

参见15.2.1节，可定义如下Item\_base类：

```
class Item_base {
public:
    Item_base(const std::string &book = "", 
              double sales_price = 0.0):
        isbn(book), price(sales_price) { }

    std::string book() const
    {
        return isbn;
    }

    // 返回特定购书量的总价格
    // 派生类将重载该函数以应用不同的折扣策略
    virtual double net_price(size_t n) const
    {
        return n * price;
    }
}
```

```

    }

    virtual ~Item_base() { }

private:
    std::string isbn;
protected:
    double price;
};

```

**习题15.4**

图书馆可以借阅不同种类的资料——书、CD和DVD等。不同种类的借阅资料有不同的登记、检查和过期规则。下面的类定义了这个应用程序可以使用的基类。指出在所有借阅资料中，哪些函数可能定义为虚函数。如果有，哪些函数可能是公共的。（注：假定LibMember是表示图书馆读者的类，Date是表示特定年份的日历日期的类。）

```

class Library {
public:
    bool check_out(const LibMember&);
    bool check_in (const LibMember&);
    bool is_late(const Date& today);
    double apply_fine();
    ostream& print(ostream& = cout);
    Date due_date() const;
    Date date_borrowed() const;
    string title() const;
    const LibMember& member() const;
};

```

**【解答】**

因为不同种类的借阅资料有不同的登记、检查和过期规则，所以check\_out、check\_in、is\_late和apply\_fine等函数应定义为虚函数。print也应定义为虚函数，因为不同种类的借阅资料，需打印的项目可能不同。

due\_date、date\_borrowed、title和member等函数可能是公共的，因为应还日期、借出日期、标题和读者等应该是所有借阅资料的公共性质，相应的操作也应该是公共的。

**习题15.5**

如果有，下面的声明中哪些是错误的？

- ```

class Base { ... };

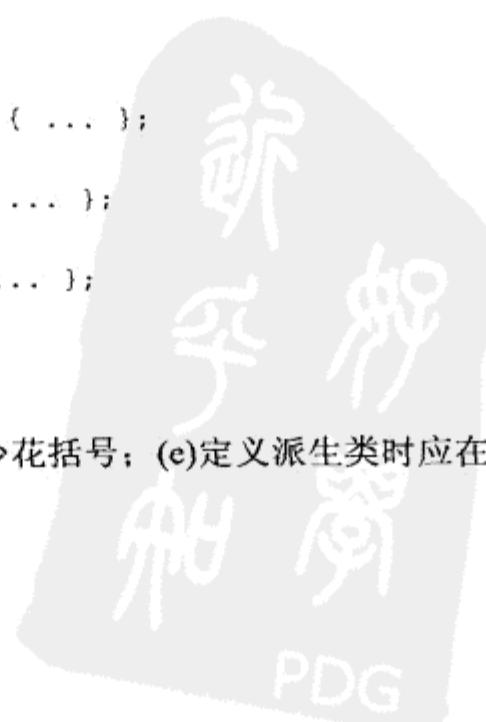
(a) class Derived : public Derived { ... };
(b) class Derived : Base { ... };
(c) class Derived : private Base { ... };
(d) class Derived : public Base;
(e) class Derived inherits Base { ... };

```

**【解答】**

错误的有(a)、(d)和(e)。

- (a)不能用类本身作为类的基类；(d)缺少花括号；(e)定义派生类时应在冒号后接类派生列表，而不是使用inherits。



**习题15.6**

编写自己的Bulk\_item类版本。

**【解答】**

见15.2.3节，可定义如下Bulk\_item类：

```
class Bulk_item : public Item_base {
public:
    // 重定义基类版本以实现批量购买折扣策略
    double net_price(std::size_t) const;
private:
    std::size_t min_qty;
    double discount;
};

// 若购书量高于下限，则使用折扣价格
double net_price(std::size_t cnt) const
{
    if (cnt > min_qty)
        return cnt * (1 - discount) * price;
    else
        return cnt * price;
}
```

**习题15.7**

可以定义一个类型实现有限折扣策略。这个类可以给低于某个上限的购书量一个折扣，如果购买的数量超过该上限，则超出部分的书应按正常价格购买。定义一个类实现这种策略。

**【解答】**

可定义如下Lds\_item类：

```
// 有限折扣类
class Lds_item : public Item_base {
public:
    // 构造函数
    Lds_item(const std::string& book = "",
              double sales_price = 0.0,
              std::size_t qty = 0, double disc_rate = 0.0):
        Item_base(book, sales_price),
        max_qty(qty), discount(disc_rate) {}

    // 重定义基类版本以实现有限折扣策略
    // 对低于上限的购书量使用折扣价格
    double Lds_item::net_price(std::size_t cnt) const
    {
        if (cnt <= max_qty)
            return cnt * (1 - discount) * price;
        else
            return cnt * price - max_qty * discount * price;
    }
private:
    std::size_t max_qty;      // 可打折的购书量上限
    double discount;         // 折扣率
};
```

注意，构造函数的定义见15.4节。

**习题15.8**

对于下面的类，解释每个函数：

```
struct base {
    string name() { return basename; }
    virtual void print(ostream &os) { os << basename; }
private:
    string basename;
};

struct derived {
    void print() { print(ostream &os); os << " " << mem; }
private:
    int mem;
};
```

如果该代码有问题，如何修正？

**【解答】**

`name`函数返回`base`类的`basename`成员；`base`类中的`print`函数输出`base`类的`basename`成员；`derived`类中的`print`函数试图调用`base`类的`print`函数输出`base`类的`basename`成员，然后输出空格及`derived`类的`mem`成员。

该代码的问题在于`derived`类的定义不正确：未指定基类且`print`函数的定义不正确。可更正为：

```
struct derived : public base {
    void print(ostream &os) { base::print(os); os << " " << mem; }
private:
    int mem;
};
```

注意，`base`类和`derived`类还应该定义适当的构造函数，如：

```
base::base(string name): basename(name) {}
derived::derived(string name, int val): base(name), mem(val) {}
```

**习题15.9**

给定上题中的类和如下对象，确定在运行时调用哪个函数：

|                   |                                  |                   |
|-------------------|----------------------------------|-------------------|
| base bobj;        | base *bp1 = &bobj <sup>1</sup> ; | base &br1 = bobj; |
| derived dobj;     | base *bp2 = &dobj <sup>2</sup> ; | base &br2 = dobj; |
| (a) bobj.print(); | (b) dobj.print();                | (c) bp1->name();  |
| (d) bp2->name();  | (e) br1.print();                 | (f) br2.print();  |

**【解答】**

(a) 调用`base`类中定义的`print`函数：因为`bobj`是`base`类对象。

(b) 调用`derived`类中定义的`print`函数：因为`dobj`是`derived`类对象。

(c) 调用`base`类中定义的`name`函数：因为`name`是非虚函数，且`bp1`的类型为`base*`。

(d) 调用`base`类中定义的`name`函数：因为`name`是非虚函数，且`bp2`的类型为`base*`。

(e) 调用`base`类中定义的`print`函数：因为通过引用调用虚函数实行动态绑定，且`br1`引用的实际对象为`base`类对象。

1. 此处英文原书有误。

2. 此处英文原书有误。

(f) 调用 derived类中定义的 print 函数：因为通过引用调用虚函数实行动态绑定，且 br2 引用的实际对象为 derived类对象。

### 习题15.10

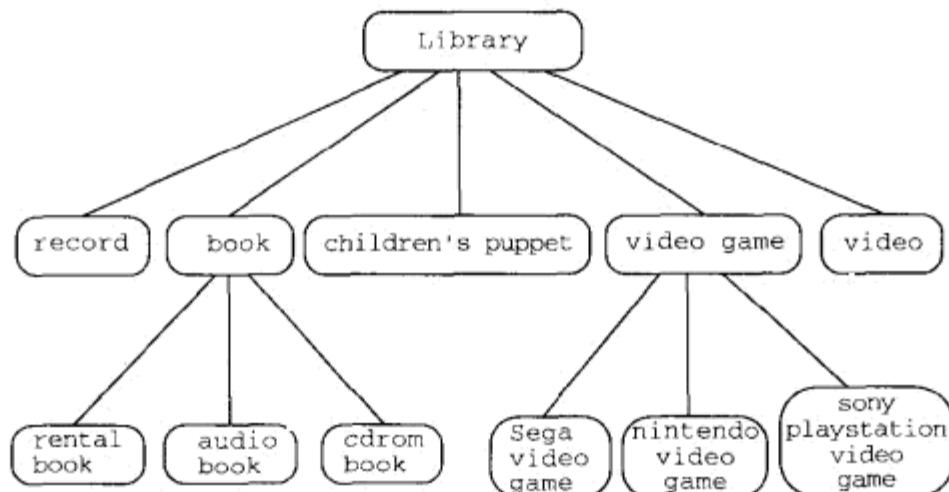
在15.2.1节的习题中编写了一个表示图书馆借阅政策的基类。假定图书馆提供下列种类的借阅资料，每一种有自己的检查和登记政策。将这些项目组织成一个继承层次：

|                        |                     |             |
|------------------------|---------------------|-------------|
| book                   | audio book          | record      |
| children's puppet      | sega video game     | video       |
| cdrom book             | nintendo video game | rental book |
| sony playstation video | game                |             |

### 【解答】

根据这些项目之间的“是一种 (Is A)”关系，可组织成如下继承层次：

- 类 book、record、video、children's puppet、video game 继承基类 Library。
  - 类 audio book、cdrom book、rental book 继承类 book。
  - 类 sega video game、nintendo video game、sony playstation video game 继承类 video game。
- 如下图所示。



注意，此处采用了与 *C++ Primer* (第4版) 一致的表示方式。如果采用UML类图表示，则有所不同。

### 习题15.11

在下列包含一族类型的一般抽象中选择一种（或者自己选择一个），将这些类型组织成一个继承层次。

- 图像文件格式（如 gif、tiff、jpeg 和 bmp）
- 几何图元（如矩形、圆、球形和锥形）
- C++语言的类型（如类、函数和成员函数）

### 【解答】

例如，对(b)中所给出的几何图元可组织成如下继承层次：

- 公共基类 Figure，表示几何图元。
- 类 Rectangle、Circle、Sphere 和 Cone 分别表示矩形、圆、球形和锥形等图元。这些类可定

义为 Figure 类的派生类。

### 习题15.12

对习题15.11中选择的类，标出可能的虚函数以及public和protected成员。

#### 【解答】

可给出如下类定义：

```
class Figure {
public:
    Figure(double, double);
protected:
    double xSize, ySize; // 图元的尺寸
}

class Figure_2D : public Figure {
public:
    Figure_2D(double, double);
    virtual double area() = 0;           // 求面积操作：纯虚函数
    virtual double perimeter() = 0;      // 求周长操作：纯虚函数
}

class Figure_3D : public Figure {
public:
    Figure_3D(double, double, double);
    virtual double cubage() = 0;         // 求体积操作：纯虚函数
protected:
    double zSize; // 立体图元的Z轴尺寸
}

class Rectangle : public Figure_2D {
public:
    Rectangle(double, double);
    virtual double area();             // 重定义求面积操作
    virtual double perimeter();        // 重定义求周长操作
}

class Circle : public Figure_2D {
public:
    Circle (double, double = 0);      // 圆形的尺寸只需给出半径
    virtual double area();             // 重定义求面积操作
    virtual double perimeter();        // 重定义求周长操作
}

class Sphere : public Figure_3D {
public:
    Sphere(double, double, double);
    virtual double cubage();          // 重定义求体积操作
}

class Cone : public Figure_3D {
public:
    Cone(double, double, double);
    virtual double cubage();          // 重定义求体积操作
}
```

注意，这里给出的是非常简单的类层次设计，可用于几何计算应用程序中。如果用于其他情况（如图形处理应用程序），则可能设计完全不同的类层次及类操作（如可设计像素、颜色、位置等数据成

员, 显示、隐藏、移动、放大、缩小等操作)。对于在基类中无法确定其行为的操作, 应定义为纯虚函数(见15.6节)。为了更清楚地表示几何图元的分类关系, 这里定义了二维图元(Figure\_2D)类及三维图元(Figure\_3D)类, 用于封装二维图元及三维图元的共性。

### 习题15.13

对于下面的类, 列出C1中的成员函数访问ConcreteBase的static成员的所有方式, 列出C2类型的对象访问这些成员的所有方式。

```
struct ConcreteBase {
    static std::size_t object_count();
protected:
    static std::size_t obj_count;
};

struct C1 : public ConcreteBase { /* ... */ };
struct C2 : public ConcreteBase { /* ... */ };
```

### 【解答】

C1中的成员函数访问ConcreteBase的static成员的方式如下:

- (1) 使用ConcreteBase::成员名(即基类名::成员名)。
- (2) 使用C1::成员名(即派生类名::成员名)。
- (3) 通过C1类对象或对象引用, 使用点(.)操作符。
- (4) 通过C1类对象的指针, 使用箭头(->)操作符。
- (5) 直接使用成员名。

C2类型的对象不能访问obj\_count成员, 因为该成员是受保护成员, 不能通过对象访问。

C2类型的对象访问object\_count的方式(假设obj为C2类型的对象)如下:

```
obj.object_count();
obj.ConcreteBase::object_count();
obj.C2::object_count();
```

### 习题15.14

重新定义Bulk\_item和Item\_base类, 使每个类只需定义一个构造函数。

### 【解答】

```
// 不使用折扣策略的基类
class Item_base {
public:
    Item_base(const std::string &book = "",
              double sales_price = 0.0):
        isbn(book), price(sales_price) {}

    std::string book() const
    {
        return isbn;
    }

    // 返回特定购书量的总价格
    // 派生类将重载该函数以应用不同的折扣策略
    virtual double net_price(size_t n) const
    {
        return n * price;
```

```

    }

    virtual ~Item_base() { }

private:
    std::string isbn;

protected:
    double price;
};

// 批量购买折扣类
class Bulk_item : public Item_base {
public:
    Bulk_item(const std::string& book = "",
              double sales_price = 0.0,
              size_t qty = 0, double disc_rate = 0.0):
        Item_base(book, sales_price),
        min_qty(qty), discount(disc_rate) {}

    // 重定义基类版本以实现批量购买折扣策略:
    // 若购书量高于下限，则使用折扣价格
    double net_price(size_t cnt) const
    {
        if (cnt >= min_qty)
            return cnt * (1 - discount) * price;
        else
            return cnt * price;
    }

private:
    size_t min_qty;           // 可打折的最小购买量
    double discount;          // 折扣率
};

```

**习题15.15**

对于15.2.5节第一个习题中描述的图书馆类层次，识别基类和派生类构造函数。

**【解答】**

为各个类设计数据成员，并根据数据成员确定相应的构造函数，派生类构造函数应在初始化列表中包含其直接基类以初始化继承成员。

**习题15.16**

对于下面的基类定义：

```

struct Base {
    Base(int val): id(val) {}
protected:
    int id;
};

```

解释为什么下述每个构造函数是非法的。

- (a) struct C1 : public Base {
 C1(int val): id(val) {}
 },
- (b) struct C2 : public C1 {
 C2(int val): Base(val), C1(val) {}
 },



```
(c) struct C3 : public C1 {
    C3(int val) : Base(val) { }
};

(d) struct C4 : public Base {
    C4(int val) : Base(id + val){ }
};

(e) struct C5 : public Base {
    C5() { }
};
```

**【解答】**

- (a) 没有在初始化列表中向基类构造函数传递实参。
- (b) 初始化列表中出现了非直接基类Base。
- (c) 初始化列表中出现了非直接基类Base而没有出现直接基类C1。
- (d) 初始化列表中使用了未定义的变量id。
- (e) 缺少初始化列表：Base类没有默认构造函数，其派生类必须用初始化列表的方式向Base类的构造函数传递实参。

**习题15.17**

说明在什么情况下类应该具有虚析构函数。

**【解答】**

作为基类使用的类应该具有虚析构函数，以保证在删除（指向动态分配对象的）基类指针时，根据指针实际指向的对象所属的类型运行适当的析构函数。

**习题15.18**

虚析构函数必须执行什么操作？

**【解答】**

这个题目有点令人费解。虚析构函数可以不执行任何操作，为空函数。

一般而言，析构函数的主要作用是清除本类中定义的数据成员。如果该类没有定义指针类成员，则使用合成版本即可；如果该类定义了指针成员，则一般需要自定义析构函数以对指针成员进行适当的清除。因此，如果有虚析构函数必须执行的操作，则就是清除本类中定义的数据成员的操作。

**习题15.19**

如果这个类定义有错，可能是什么错？

```
class AbstractObject {
public:
    virtual void doit();
    // other members not including any of the copy-control functions
};
```

**【解答】**

可能错在该类没有提供虚析构函数，因为该类有可能作为基类使用。

**习题15.20**

回忆在13.3节习题中编写的类，该类的复制控制成员打印一条消息，为Item\_base和Bulk\_item类

的构造函数增加打印语句。定义复制控制成员，使之完成与合成版本相同的工作外，还打印一条消息。应用使用了 Item\_base 类型的那些对象和函数编写一些程序，在每种情况下，预测将会创建和撤销什么对象，并将你的预测与程序所产生的结果进行比较。继续实验，直至你能够正确地预测对于给定的代码片段，会执行哪些复制控制成员。

### 【解答】

下面给出按题目修改后的 Item\_base 和 Bulk\_item 类，以及一个使用 Item\_base 和 Bulk\_item 对象的实例程序：

```
// 15-20.cpp
// 修改Item_base和Bulk_item类:
// 为构造函数增加打印语句;
// 定义复制控制成员，使之完成与合成版本相同的工作外，还打印有关本函数的信息。
// 用不同方式使用Item_base类型及Bulk_item类型的对象:
// 作为非引用形参和引用形参传递，动态分配,
// 作为函数返回值，进行赋值操作。
// 研究何时执行哪个构造函数和复制控制成员
#include <iostream>
#include <string>
using namespace std;

// 不使用折扣策略的基类
class Item_base {
public:
    Item_base(const std::string &book = "", double sales_price = 0.0):
        isbn(book), price(sales_price)
    {
        std::cout << "Item_base(const std::string&, double)" << std::endl;
    }

    std::string book() const
    {
        return isbn;
    }

    // 返回特定购书量的总价格
    // 派生类将重载该函数以应用不同的折扣策略
    virtual double net_price(size_t n) const
    {
        return n * price;
    }

    // 复制控制成员
    Item_base(const Item_base& ib) : isbn(ib.isbn), price(ib.price)
    {
        std::cout << "Item_base(const Item_base&)" << std::endl;
    }

    Item_base& operator= (const Item_base& rhs)
    {
        isbn = rhs.isbn;
        price = rhs.price;

        std::cout << "Item_base::operator=()" << std::endl;
        return *this;
    }
}
```

```

    }

    virtual ~Item_base()
    {
        std::cout << "~Item_base()" << std::endl;
    }

private:
    std::string isbn;
protected:
    double price;
};

// 批量购买折扣类
class Bulk_item : public Item_base {
public:
    Bulk_item(const std::string& book = "",
              double sales_price = 0.0,
              size_t qty = 0, double disc_rate = 0.0):
        Item_base(book, sales_price),
        min_qty(qty), discount(disc_rate)
    {
        std::cout << "Bulk_item(const std::string&, double, size_t, double)"
              << endl;
    }

    // 重定义基类版本以实现批量购买折扣策略:
    // 若购书量高于下限，则使用折扣价格
    double net_price(size_t cnt) const
    {
        if (cnt >= min_qty)
            return cnt * (1 - discount) * price;
        else
            return cnt * price;
    }

    // 复制控制成员
    Bulk_item(const Bulk_item& b) :
        Item_base(b), min_qty(b.min_qty),
        discount(b.discount)
    {
        std::cout << "Bulk_item(const Bulk_item&)" << std::endl;
    }

    Bulk_item& operator=(const Bulk_item& rhs)
    {
        if (this != &rhs)
            Item_base::operator=(rhs);
        min_qty = rhs.min_qty;
        discount = rhs.discount;

        std::cout << "Bulk_item::operator=(())" << std::endl;
        return *this;
    }

    virtual ~Bulk_item()
    {
        std::cout << "~Bulk_item()" << std::endl;
    }

private:
    size_t min_qty; // 可打折的最小购买量
    double discount; // 折扣率
}

```



```

    // 普通构造函数, 创建Bulk_item对象
    // 象中的Item_base子对象

    delete q;      // 调用Bulk_item类的析构函数撤销动态创建的Bulk_item对象,
    // 由此导致调用Item_base类的析构函数,
    // 撤销Bulk_item对象中的Item_base子对象

    return 0;      // bobj生命周期结束, 自动调用Bulk_item类的析构函数撤销
    // 由此导致调用Item_base类的析构函数,
    // 撤销bobj中的Item_base子对象
    // iobj生命周期结束, 自动调用Item_base类的析构函数撤销
}

```

运行上述程序，可得到如下输出结果：

```

Item_base(const std::string&, double)
Item_base(const Item_base&)
~Item_base()
Item_base(const std::string&, double)
Item_base(const Item_base&)
~Item_base()
Item_base::operator=()
~Item_base()
Item_base(const std::string&, double)
~Item_base()
Item_base(const std::string&, double)
Bulk_item(const std::string&, double, size_t, double)
Item_base(const Item_base&)
~Item_base()
Item_base(const std::string&, double)
Bulk_item(const std::string&, double, size_t, double)
~Bulk_item()
~Item_base()
~Bulk_item()
~Item_base()
~Item_base()

```

之所以产生上述输出的原因，详见程序代码中的注释。

### 习题15.21

重新定义Item\_base层次以包含Disc\_item类。

#### 【解答】

```

// 不使用折扣策略的基类
class Item_base {
public:
    Item_base(const std::string &book = "",           // 构造函数
              double sales_price = 0.0):             // 书名(book), 销售价格(sales_price) { }

    std::string book() const
    {
        return isbn;
    }

    // 返回特定购书量的总价格
    // 派生类将重载该函数以应用不同的折扣策略
    virtual double net_price(size_t n) const
    {
        return n * price;
    }
}

```

```

}

virtual ~Item_base() { }

private:
    std::string isbn;

protected:
    double price;
};

// 保存折扣率和可实行折扣策略的数量
// 派生类将使用这些数据实现定价策略
class Disc_item : public Item_base {
public:
    Disc_item(const std::string& book = "",
              double sales_price = 0.0,
              size_t qty = 0, double disc_rate = 0.0):
        Item_base(book, sales_price),
        quantity(qty), discount(disc_rate) {}

    double net_price(size_t) const = 0;

    std::pair<size_t, double> discount_policy() const
    {
        return std::make_pair(quantity, discount);
    }
protected:
    size_t quantity;      // 可实行折扣策略的购买量
    double discount;      // 折扣率
};

```

注意，Disc\_item中将函数net\_price重定义为纯虚函数以防止用户创建Disc\_item对象（见15.6节）。

### 习题15.22

重新定义Bulk\_item和你在15.2.3节习题中实现的那个表示有限折扣策略的类，以继承Disc\_item类。

#### 【解答】

```

// 批量购买折扣类
class Bulk_item : public Disc_item {
public:
    Bulk_item(const std::string& book = "",
              double sales_price = 0.0,
              size_t qty = 0, double disc_rate = 0.0):
        Disc_item(book, sales_price, qty, disc_rate) {}

    // 重定义基类版本以实现批量购买折扣策略：
    // 若购书量高于下限，则使用折扣价格
    double net_price(size_t cnt) const
    {
        if (cnt >= quantity)
            return cnt * (1 - discount) * price;
        else
            return cnt * price;
    }
};

// 有限折扣类
class Lds_item : public Disc_item {

```

```

public:
    // 构造函数
    Lds_item(const std::string& book = "", 
              double sales_price = 0.0,
              size_t qty = 0, double disc_rate = 0.0):
        Disc_item(book, sales_price, qty, disc_rate) ( )

    // 重定义基类版本以实现有限折扣策略:
    // 对低于上限的购书量使用折扣价格
    double net_price(size_t cnt) const
    {
        if (cnt <= quantity)
            return cnt * (1 - discount) * price;
        else
            return cnt * price - quantity * discount * price;
    }
};

```

**习题15.23**

对于下面的基类和派生类定义:

```

struct Base {
    foo(int);
protected:
    int bar;
    double foo_bar;
};

struct Derived : public Base {
    foo(string);
    bool bar(Base *pb);
    void foobar();
protected:
    string bar;
};

```

找出下述每个例子中的错误并说明怎样改正:

- (a) Derived d; d.foo(1024);
- (b) void Derived::foobar() { bar = 1024; }
- (c) bool Derived::bar(Base \*pb)
 { return foo\_bar == pb->foo\_bar; }

**【解答】**

(a) 调用foo函数所给定的实参类型错误。通过Derived类对象d调用foo函数，调用到的是Derived类中定义的foo函数，应使用string类型的实参。

(b) 用int型值1024对bar进行赋值错误。在Derived类中定义的数据成员bar屏蔽了基类Base中的同名成员，所以此处访问到的是Derived类中定义的bar，应赋以string类型的对象（或C风格字符串）。

(c) 通过指向Base类对象的指针访问其受保护成员foo\_bar错误。可改正为将pb定义为指向Derived类对象的指针。

**习题15.24**

对于如下代码:

```
Bulk_item bulk;
```

```
Item_base item(bulk);
Item_base *p = &bulk;
```

为什么表达式

```
p->net_price(10);
```

调用net\_price的Bulk\_item实例，而表达式

```
item.net_price(10);
```

调用Item\_base实例？

### 【解答】

net\_price是基类Item\_base中定义的虚函数。对虚函数而言，通过指向基类对象的指针调用进行动态绑定，即调用该指针实际指向的对象所属类型中定义的函数。p是指向Base\_item类对象的指针，实际指向Bulk\_item类对象bulk，所以，表达式p->net\_price(10)调用net\_price的Bulk\_item实例。

而通过对象调用虚函数，所调用到的总是该对象所属类型中定义的函数。item是一个Item\_base对象（虽然用Bulk\_item对象bulk对其进行初始化），所以，表达式item.net\_price(10)调用net\_price的Item\_base实例。

### 习题15.25

假定Derived继承Base，并且Base将下面的函数定义为虚函数；假定Derived打算定义自己的这个虚函数的版本，确定在Derived中哪个声明是错误的，并指出为什么错。

- (a) Base\* Base::copy(Base\*);  
Base\* Derived::copy(Derived\*);
- (b) Base\* Base::copy(Base\*);  
Derived\* Derived::copy(Base\*);
- (c) ostream& Base::print(int, ostream&=cout);  
ostream& Derived::print(int, ostream&);
- (d) void Base::eval() const;  
void Derived::eval();

### 【解答】

错误的有：

(a) 单纯从语法上看并没有错误，但Derived中声明的copy是一个非虚函数，而不是对Base中声明的虚函数copy的重定义，因为派生类中重定义的虚函数必须具有与基类中虚函数相同的原型（唯一的例外是返回类型可以稍有不同，见15.8.2节clone函数的定义）。而且Derived中定义的copy函数还屏蔽了基类Base的copy函数。

### 习题15.26

使你的Disc\_item类版本成为抽象类。

### 【解答】

只需在Disc\_item类中将函数net\_price重定义为纯虚函数（见习题15.21解答）：

```
double net_price(size_t) const = 0;
```

**习题15.27**

试着定义Disc\_item类型的一个对象，看看会从编译器得到什么错误。

**【解答】**

编译器给出错误提示，指出不能创建抽象类的对象。

**习题15.28**

定义一个vector来保存Item\_base类型的对象，并将一些Bulk\_item类型对象复制到vector中。遍历vector并计算容器中元素的net\_price。

**【解答】**

将习题15.21及习题15.22的解答中定义的Item\_base类层次放在头文件Item.hpp中。

可编写程序如下：

```
// 15-28.cpp
// 定义一个vector保存Item_base类型的对象,
// 将一些Bulk_item类型对象复制到vector中。
// 遍历vector并根据容器中元素计算net_price总和
#include "Item.hpp" // 引入Item_base类层次的定义
#include <iostream>
#include <string>
#include <utility>
#include <vector>
using namespace std;

int main()
{
    vector<Item_base> itemVec;
    string isbn;
    double price, qty, discount;

    // 读入Bulk_item对象并复制到vector中
    cout << "Enter some Bulk_item objects(Ctrl+Z to end): " << endl;
    while (cin >> isbn >> price >> qty >> discount) {
        itemVec.push_back(Bulk_item(isbn, price, qty, discount));
    }

    // 遍历vector并根据容器中元素计算购买100本书的net_price总和
    double sum = 0.0;
    for (vector<Item_base>::iterator iter = itemVec.begin();
                     iter != itemVec.end(); ++iter)
        sum += iter->net_price(100);

    // 输出结果
    cout << "summation of net price: " << sum << endl;
    return 0;
}
```

**习题15.29**

重复程序，但这次存储Item\_base类型对象的指针。比较结果总和。

**【解答】**

将习题15.21及习题15.22的解答中定义的Item\_base类层次放在头文件Item.hpp中。

可编写程序如下：

```
// 15-29.cpp
// 定义一个vector保存Item_base类型对象的指针,
// 将一些Bulk_item类型对象复制到vector中。
// 遍历vector并根据容器中元素计算net_price总和
#include "Item.hpp" // 引入Item_base类层次的定义
#include <iostream>
#include <string>
#include <utility>
#include <vector>
using namespace std;

int main()
{
    vector<Item_base*> itemVec;
    string isbn;
    double price, qty, discount;

    // 读入Bulk_item对象并复制到vector中
    cout << "Enter some Bulk_item objects(Ctrl+Z to end): " << endl;
    while (cin >> isbn >> price >> qty >> discount) {
        // 动态创建Bulk_item对象并将其指针复制到vector中
        Bulk_item *p = new Bulk_item(isbn, price, qty, discount);
        itemVec.push_back(p);
    }

    // 遍历vector并根据容器中元素计算购买100本书的net_price总和
    double sum = 0.0;
    for (vector<Item_base*>::iterator iter = itemVec.begin();
         iter != itemVec.end(); ++iter)
        sum += (*iter)->net_price(100);

    // 输出结果
    cout << "summation of net price: " << sum << endl;

    // 释放动态分配的Bulk_item对象
    for (vector<Item_base*>::iterator it = itemVec.begin();
         it != itemVec.end(); ++it)
        delete *it;
}

return 0;
}
```

此处的结果总和为实行批量购买折扣政策所得到的net price，而习题15.28所得到的结果总和是按原价计算得到的net price（未打折）。

注意，粗体部分表示的为与上题有区别的代码。

### 习题15.30

解释习题15.28和习题15.29中的程序所产生总和的差异。如果没有差异，解释为什么没有。

#### 【解答】

习题15.28和习题15.29中的程序所产生的总和有差异，原因在于：习题15.28通过Item\_base类型的对象（该对象虽然作为Bulk\_item对象的副本而建立，但仍是Item\_base对象）调用虚函数net\_price，不实行动态绑定，调用的是Item\_base类中定义的版本；习题15.29通过Item\_base类型的指针调用虚函数net\_price，实行动态绑定，而该指针实际指向Bulk\_item对象，所以调用的是Bulk\_item类中定义的版本。

**习题15.31**

为15.2.3节的习题中实现的有限折扣类定义和实现clone操作。

**【解答】**

```
// 有限折扣类
class Lds_item : public Item_base {
public:
    Lds_item* clone() const
    {
        return new Lds_item(*this);
    }

    // 其他成员同习题15.7中的定义，此处不再赘述
};
```

**习题15.32**

实际上，程序不太可能在第一次运行或第一次用真实数据运行时就能正确运行。在类设计中包括调试策略经常是有用的。为Item\_base类层次实现一个debug虚函数，显示各个类的数据成员。

**【解答】**

```
// 不使用折扣策略的基类
class Item_base {
    // ...其余成员（略）
public:
    virtual void debug(ostream& os = cout) const
    {
        os << isbn << "\t" << price;
    }
};

// 保存折扣率和可实行折扣策略的数量
// 派生类将使用这些数据实现定价策略
class Disc_item : public Item_base {
    // ...其余成员（略）
public:
    virtual void debug(ostream& os = cout) const
    {
        Item_base::debug(os);
        os << "\t" << quantity << "\t"
            << discount;
    }
};
```

注意，Bulk\_item类和Lds\_item类没有定义自己的数据成员，所以无需重定义debug函数，可直接使用Disc\_item类的debug函数。

为节省篇幅，此处省略了各个类中其余成员的定义（见习题15.21和习题15.22的解答）

**习题15.33**

对于Item\_base层次的包括Disc\_item抽象基类的版本，指出Disc\_item类是否应实现clone函数，为什么？

**【解答】**

Disc\_item类不应该实现clone函数，因为如果Disc\_item类是抽象类，则不能创建Disc\_item类对象。

**习题15.34**

修改调试函数以允许用户打开或关闭调试。用两种方式实现控制：

- 通过定义debug函数的形参。
- 通过定义类数据成员。该成员允许个体对象打开或关闭调试信息的显示。

**【解答】**

- 通过定义debug函数的形参。

在debug函数中增加一个形参ctrl，用于控制调试的打开或关闭。

将各类中的debug函数修改如下：

```
// 不使用折扣策略的基类
class Item_base {
    // ...其余成员（略）
public:
    // 参数ctrl置0将关闭调试
    virtual void debug(int ctrl = 1, ostream& os = cout) const
    {
        if (ctrl == 0)
            return;

        os << isbn << "\t" << price;
    }
};

// 保存折扣率和可实行折扣策略的数量
// 派生类将使用这些数据实现定价策略
class Disc_item : public Item_base {
    // ...其余成员（略）
public:
    // 参数ctrl置0将关闭调试
    virtual void debug(int ctrl = 1, ostream& os = cout) const
    {
        if (ctrl == 0)
            return;

        Item_base::debug(ctrl, os);
        os << "\t" << quantity << "\t"
            << discount;
    }
};
```

- 通过定义类数据成员。该成员允许个体对象打开或关闭调试信息的显示。

可在Item\_base类中定义一个受保护数据成员is\_debug，用于控制调试信息的显示与否，相应地需要修改类层次中各个类的构造函数以初始化该成员。Item\_base类和Disc\_item类中的debug函数根据is\_debug控制是否显示调试信息，而且可以在Item\_base类中提供一个公有的set\_debug函数，以允许个体对象打开或关闭调试信息的显示。

修改后的类定义代码如下：

```
// 不使用折扣策略的基类
class Item_base {
    // ...其余成员（略）
public:
    Item_base(const std::string &book = "",
              double sales_price = 0.0, bool dbg = false):
```

```

        isbn(book), price(sales_price), is_debug(dbg) { }

// 根据is_debug成员决定是否显示其他数据成员
virtual void debug(ostream& os = cout) const
{
    if (!is_debug)
        return;

    os << isbn << "\t" << price;
}

// 设置is_debug数据成员
void set_debug(bool dbg)
{
    is_debug = dbg;
}

protected:
    bool is_debug;
};

// 保存折扣率和可实行折扣策略的数量
// 派生类将使用这些数据实现定价策略
class Disc_item : public Item_base {
    // ...其余成员(略)
public:
    Disc_item(const std::string& book = "",
              double sales_price = 0.0, size_t qty = 0,
              double disc_rate = 0.0, bool dbg = false):
        Item_base(book, sales_price, dbg),
        quantity(qty), discount(disc_rate) {}

    // 根据debug成员决定是否显示其他数据成员
    virtual void debug(ostream& os = cout) const
    {
        if (!is_debug)
            return;

        Item_base::debug(os);
        os << "\t" << quantity << "\t"
            << discount;
    }
};

// 批量购买折扣类
class Bulk_item : public Disc_item {
    // ...其余成员(略)
public:
    Bulk_item(const std::string& book = "",
              double sales_price = 0.0, size_t qty = 0,
              double disc_rate = 0.0, bool dbg = false):
        Disc_item(book, sales_price, qty, disc_rate, dbg) {}

};

// 有限折扣类
class Lds_item : public Disc_item {
    // ...其余成员(略)
public:
    // 构造函数
    Lds_item(const std::string& book = "",
              double sales_price = 0.0, size_t qty = 0,
              double disc_rate = 0.0, bool dbg = false):
        Disc_item(book, sales_price, qty, disc_rate, dbg) {}

};

```

注意，各个类中省略的成员定义见习题15.21及习题15.22的解答。

### 习题15.35

编写自己的compare函数和Basket类的版本并使用它们管理销售。

#### 【解答】

定义compare函数和Basket类的头文件如下：

```
// Basket.hpp(for 15-35)
// 定义compare函数和Basket类的头文件
#ifndef BASKET_H
#define BASKET_H

#include "Sales_item.hpp"
#include <set>

// compare函数用于比较对象,
// 以确定Basket的multiset成员中各元素的排列次序
inline bool
compare(const Sales_item &lhs, const Sales_item &rhs)
{
    return lhs->book() < rhs->book();
}

class Basket {
    // 用于multiset元素排序的比较函数的类型
    typedef bool (*Comp)(const Sales_item&, const Sales_item&);
public:
    // multiset成员的类型
    typedef std::multiset<Sales_item, Comp> set_type;
    // 类型别名
    typedef set_type::size_type size_type;
    typedef set_type::const_iterator const_iter;

    Basket(): items(compare) {} // 初始化比较器

    void add_item(const Sales_item &item)
    {
        items.insert(item);
    }

    size_type size(const Sales_item &i) const
    {
        return items.count(i);
    }

    double total() const; // 返回购物篮中所有物品的总价格
private:
    std::multiset<Sales_item, Comp> items;
};

#endif
```

定义compare函数和Basket类的实现文件如下：

```
// Basket.cpp(for 15-35)
// 定义compare函数和Basket类的实现文件(源文件)
#include "Basket.hpp"

double Basket::total() const
```

```

{
    double sum = 0.0; // holds the running total

    // 找到具有相同ISBN的每一组物品，根据其总数计算价格
    // iter指向ISBN相同的每一组中的第一个元素
    // upper_bound指向具有不同ISBN的下一个元素
    for (const_iter iter = items.begin();
         iter != items.end();
         iter = items.upper_bound(*iter))
    {
        // 根据基础Item_base对象的类型调用相应的net_price函数
        sum += (*iter)->net_price(items.count(*iter));
    }
    return sum;
}

```

使用Basket类管理销售的主程序如下：

```

// 15-35.cpp
// 使用Basket类管理销售
#include "Basket.hpp"
#include "Sales_item.hpp"
#include <iostream>
using namespace std;

int main()
{
    Basket basket;
    Sales_item item1(Bulk_item("7-115-14554-7", 99, 20, 0.2));
    Sales_item item2(Item_base("7-115-14554-8", 39));
    Sales_item item3(Lds_item("7-115-14554-9", 50, 200, 0.2));
    Sales_item item4(Bulk_item("7-115-14554-7", 99, 20, 0.2));

    basket.add_item(item1);
    basket.add_item(item2);
    basket.add_item(item3);
    basket.add_item(item4);

    cout << basket.total() << endl;

    return 0;
}

```

Item\_base类层次的定义如下：

```

// Item.hpp(for 15-35)
// 定义Item_base类层次的头文件
#ifndef ITEM_H
#define ITEM_H
#include <string>

// 不使用折扣策略的基类
class Item_base {
public:
    Item_base(const std::string &book = "",
              double sales_price = 0.0):
        isbn(book), price(sales_price) {}

    std::string book() const
    {
        return isbn;
    }
private:
    std::string isbn;
    double price;
};

```

```

}

// 返回特定购书量的总价格
// 派生类将重载该函数以应用不同的折扣策略
virtual double net_price(size_t n) const
{
    return n * price;
}

virtual Item_base* clone() const
{
    return new Item_base(*this);
}

virtual ~Item_base() { }

private:
    std::string isbn;
protected:
    double price;
};

// 保存折扣率和可实行折扣策略的数量
// 派生类将使用这些数据实现定价策略
class Disc_item : public Item_base {
public:
    Disc_item(const std::string& book = "",
              double sales_price = 0.0,
              size_t qty = 0, double disc_rate = 0.0):
        Item_base(book, sales_price),
        quantity(qty), discount(disc_rate) {}

    double net_price(size_t) const = 0;

    std::pair<size_t, double> discount_policy() const
    {
        return std::make_pair(quantity, discount);
    }
protected:
    size_t quantity;      // 可实行折扣策略的购买量
    double discount;     // 折扣率
};

// 批量购买折扣类
class Bulk_item : public Disc_item {
public:
    Bulk_item(const std::string& book = "",
              double sales_price = 0.0,
              size_t qty = 0, double disc_rate = 0.0):
        Disc_item(book, sales_price, qty, disc_rate) {}

    // 重定义基类版本以实现批量购买折扣策略:
    // 若购书量高于下限，则使用折扣价格
    double net_price(size_t cnt) const
    {
        if (cnt >= quantity)
            return cnt * (1 - discount) * price;
        else
            return cnt * price;
    }

    virtual Bulk_item* clone() const

```

```

    {
        return new Bulk_item(*this);
    }
};

// 有限折扣类
class Lds_item : public Disc_item {
public:
    // 构造函数
    Lds_item(const std::string& book = "",
              double sales_price = 0.0,
              size_t qty = 0, double disc_rate = 0.0):
        Disc_item(book, sales_price, qty, disc_rate) { }

    // 重定义基类版本以实现有限折扣策略:
    // 对低于上限的购书量使用折扣价格
    double net_price(size_t cnt) const
    {
        if (cnt <= quantity)
            return cnt * (1 - discount) * price;
        else
            return cnt * price - quantity * discount * price;
    }

    virtual Lds_item* clone() const
    {
        return new Lds_item(*this);
    }
};
#endif

```

注意，本题中给出的 Item.hpp 头文件与习题 15.28 解答中使用的 Item.hpp 内容基本相同，只是在类层次中增加了一个虚函数 clone（见 15.8.2 节）。

Sales\_item 类的头文件如下：

```

// Sales_item.hpp(for 15-35)
// Sales_item类的头文件
#ifndef SALESITEM_H
#define SALESITEM_H
#include "Item.hpp"

// 用于Item_base层次的使用计数式句柄类
class Sales_item {
public:
    // 默认构造函数：创建未绑定的句柄
    Sales_item(): p(0), use(new std::size_t(1)) { }

    // 将创建绑定到Item_base对象副本的句柄
    Sales_item(const Item_base& item):
        p(item.clone()), use(new std::size_t(1)) { }

    // 复制控制成员管理使用计数和指针
    Sales_item(const Sales_item &i):
        p(i.p), use(i.use) { ++*use; }
    ~Sales_item() { decr_use(); }
    Sales_item& operator=(const Sales_item&);

    // 成员访问操作符
    const Item_base *operator->() const
    {

```

```

    if (p)
        return p;
    else
        throw std::logic_error("unbound Sales_item");
}

const Item_base &operator*() const
{
    if (p)
        return *p;
    else
        throw std::logic_error("unbound Sales_item");
}
private:
    Item_base *p;           // 指向共享Item_base对象的指针
    std::size_t *use;        // 指向共享的使用计数的指针

    // 为析构函数和赋值操作符所用的辅助函数
    void decr_use()
    {
        if (--*use == 0) {
            delete p;
            delete use;
        }
    }
};

#endif

```

Sales\_item类的实现文件如下：

```

// Sales_item.cpp(for 15-35)
// Sales_item类的实现文件(源文件)
#include "Sales_item.hpp"

Sales_item&
Sales_item::operator=(const Sales_item &rhs)
{
    ++*rhs.use;
    decr_use();
    p = rhs.p;
    use = rhs.use;
    return *this;
}

```

### 习题15.36

Basket::const\_iter的基础类型是什么？

**【解答】**

Basket::const\_iter的基础类型即multiset容器的元素类型Sales\_item。

### 习题15.37

为什么在Basket的private部分定义Comp类型别名？

**【解答】**

因为该类型别名仅在Basket类中使用。

### 习题15.38

为什么在Basket中定义两个private部分？

**【解答】**

也许是为了显式区分类型别名与数据成员，事实上，将数据成员也放在第一个private部分也是可以的。

**习题15.39**

给定s1、s2、s3和s4均为string对象，确定下述Query类的使用创建什么对象：

- (a) Query(s1) | Query(s2) & ~Query(s3);
- (b) Query(s1) | (Query(s2) & ~Query(s3));
- (c) (Query(s1) & (Query(s2) | (Query(s3) & Query(s4))));

**【解答】**

(a) 创建12个对象：6个Query\_base对象及其相关联的句柄（query对象）。6个Query\_base对象分别是3个WordQuery对象、1个NotQuery对象、1个AndQuery对象和1个OrQuery对象。

·(b) 同(a)。

(c) 创建14个对象：7个Query\_base对象及其相关联的句柄（query对象）。7个Query\_base对象分别是4个WordQuery对象、2个AndQuery对象和1个OrQuery对象。

**习题15.40**

对图15-4中建立的表达式：

- (a) 列出处理这个表达式所执行的构造函数。
- (b) 列出执行`cout << q`所调用的display函数和重载的`<<`操作符。
- (c) 列出计算`q.eval`时所调用的eval函数。

**【解答】**

(a) 处理表达式`Query("fiery") & Query("bird") | Query("wind")`所执行的构造函数如下：

- `Query(const std::string&)`（执行3次）。
- `WordQuery(std::string&)`（执行3次）。
- `AndQuery(Query, Query)`（执行1次）。
- `BinaryQuery(Query, Query, std::string)`（执行2次）。
- `Query_base()`（执行5次）。
- `OrQuery(Query, Query)`（执行1次）。
- `Query(Query_base*)`（执行2次）。

注意，如果要创建Query对象q，还需要执行`Query(const Query&)`1次。

(b) 执行`cout << q`所调用的display函数和重载的`<<`操作符如下：

- Query类的`operator<<`。
- Query类的`display`。
- BinaryQuery类的`display`。
- WordQuery类的`display`。

(c) 计算`q.eval`时所调用的eval函数如下：

- Query类的`eval`。

- OrQuery 类的 eval。
- AndQuery 类的 eval。
- WordQuery 类的 eval。

### 习题15.41

实现Query类和Query\_base等类，并为第10章的TextQuery类增加需要的size操作。通过计算和打印如图15-4所示的查询，测试你的应用程序。

#### 【解答】

定义Query类及Query\_base类层次的头文件如下：

```
// Query.hpp(for 15-41)
// Query类及Query_base类层次的头文件
#ifndef QUERY_H
#define QUERY_H
#include "TextQuery.hpp"
#include <string>
#include <set>
#include <iostream>
#include <fstream>
using namespace std;

// 用作具体查询类型的基类的抽象类
class Query_base {
    friend class Query;
protected:
    typedef TextQuery::line_no line_no;
    virtual ~Query_base() { }
private:
    // 返回与该查询匹配的行的行号集合
    virtual set<line_no>
        eval(const TextQuery&) const = 0;

    // 打印查询
    virtual ostream&
        display(ostream& = cout) const = 0;
};

// 管理Query_base继承层次的句柄类
class Query {
    // 这些操作符需要访问Query_base*构造函数
    friend Query operator~(const Query &);
    friend Query operator|(const Query&, const Query&);
    friend Query operator&(const Query&, const Query&);
public:
    Query(const string&); // 建立新的WordQuery对象

    // 管理指针及使用计数的复制控制成员
    Query(const Query &c): q(c.q), use(c.use) { ++*use; }
    ~Query() { decr_use(); }
    Query& operator=(const Query&);

    // 接口函数：将调用相应的Query_base操作
    set<TextQuery::line_no>
        eval(const TextQuery &t) const { return q->eval(t); }
    ostream &display(ostream &os) const
        { return q->display(os); }
}
```

```

private:
    Query(Query_base *query): q(query),
        use(new size_t(1)) { }
    Query_base *q;
    size_t *use;
    void decr_use()
    { if (--*use == 0) { delete q; delete use; } }
};

inline Query& Query::operator=(const Query &rhs)
{
    ++*rhs.use;
    decr_use();
    q = rhs.q;
    use = rhs.use;
    return *this;
}

inline ostream&
operator<<(ostream &os, const Query &q)
{
    return q.display(os);
}

class WordQuery: public Query_base {
    friend class Query; // Query使用WordQuery构造函数
    WordQuery(const string &s): query_word(s) {}

    // 具体类WordQuery定义所有继承而来的纯虚函数
    set<line_no> eval(const TextQuery &t) const
        { return t.run_query(query_word); }
    ostream& display (ostream &os) const
        { return os << query_word; }
    string query_word; // 要查找的单词
};

inline
Query::Query(const string &s): q(new WordQuery(s)),
    use(new size_t(1)) {}

class NotQuery: public Query_base {
    friend Query operator~(const Query &);

    NotQuery(Query q): query(q) {}

    // 具体类NotQuery定义所有继承而来的纯虚函数
    set<line_no> eval(const TextQuery&) const;
    ostream& display(ostream &os) const
        { return os << "~(" << query << ")"; }
    const Query query;
};

class BinaryQuery: public Query_base {
protected:
    BinaryQuery(Query left, Query right, string op):
        lhs(left), rhs(right), oper(op) {}

    // 抽象类BinaryQuery不定义eval
    ostream& display(ostream &os) const
    { return os << "(" << lhs << " " << oper << " "
        << rhs << ")"; }
}

```

```

    const Query lhs, rhs;           // 左右操作数
    const string oper;             // 操作符
};

class AndQuery: public BinaryQuery {
    friend Query operator&(const Query&, const Query&);
    AndQuery(Query left, Query right):
        BinaryQuery(left, right, "&") { }

    // 具体类AndQuery继承display并保持为虚函数
    set<line_no> eval(const TextQuery&) const;
};

class OrQuery: public BinaryQuery {
    friend Query operator|(const Query&, const Query&);
    OrQuery(Query left, Query right):
        BinaryQuery(left, right, "|") { }

    // 具体类OrQuery继承display并保持为虚函数
    set<line_no> eval(const TextQuery&) const;
};

inline Query operator&(const Query &lhs, const Query &rhs)
{
    return new AndQuery(lhs, rhs);
}

inline Query operator|(const Query &lhs, const Query &rhs)
{
    return new OrQuery(lhs, rhs);
}

inline Query operator~(const Query &oper)
{
    return new NotQuery(oper);
}
#endif

```

Query类及Query\_base类层次的实现文件（源文件）如下：

```

// Query.cpp(for 15-41)
// Query类及Query_base类层次的实现文件（源文件）
#include "Query.hpp"
#include "TextQuery.hpp"
#include <set>
#include <algorithm>
#include <iostream>
using namespace std;

// 返回不在其操作数结果集中的行号集合
set<TextQuery::line_no>
NotQuery::eval(const TextQuery& file) const
{
    // 计算其操作数的结果集
    set<TextQuery::line_no> has_val = query.eval(file);

    set<line_no> ret_lines;

    // 检查输入文件中的每一行，看该行的行号是否在has_val中
    // 如果不在，将该行的行号加入到ret_lines
    for (TextQuery::line_no n = 0; n != file.size(); ++n)

```

```

        if (has_val.find(n) == has_val.end())
            ret_lines.insert(n);

        return ret_lines;
    }

    // 返回其操作数结果集的交集
    set<TextQuery::line_no>
    AndQuery::eval(const TextQuery& file) const
    {
        // 计算其左右操作数的结果集
        set<line_no> left = lhs.eval(file),
                      right = rhs.eval(file);

        set<line_no> ret_lines; // 保存运算的结果

        // 将左右操作数结果集的交集写至ret_lines
        set_intersection(left.begin(), left.end(),
                         right.begin(), right.end(),
                         inserter(ret_lines, ret_lines.begin()));

        return ret_lines;
    }

    // 返回其操作数结果集的并集
    set<TextQuery::line_no>
    OrQuery::eval(const TextQuery& file) const
    {
        // 计算其左右操作数的结果集
        set<line_no> right = rhs.eval(file),
                      ret_lines = lhs.eval(file);

        // 将未在ret_lines中出现的right中的行号插入到ret_lines中
        // ret_lines.insert(right.begin(), right.end());
        // 所用的编译器不支持带两个迭代器参数的insert函数，改用如下代码：
        for (set<line_no>::const_iterator iter = right.begin();
   iter != right.end(); ++iter)
            ret_lines.insert(*iter);

        return ret_lines;
    }
}

```

定义TextQuery类的头文件如下：

```

// TextQuery.hpp(for 15-41)
// TextQuery类的头文件
// 使用set容器存储行号
#ifndef TEXTQUERY_H
#define TEXTQUERY_H
#include <string>
#include <vector>
#include <set>
#include <map>
#include <cctype>
#include <iostream>
#include <fstream>
#include <cstring>
using namespace std;

class TextQuery {
public:

```

```

// 类型别名
typedef string::size_type str_size;
typedef vector<string>::size_type line_no;
// 接口:
// read_file建立给定文件的内部数据结构
void read_file(ifstream &is)
    { store_file(is); build_map(); }
// run_query查询给定单词并返回该单词所在行的行号集合
set<line_no> run_query(const string&) const;
// text_line返回输入文件中指定行号对应的行
string text_line(line_no) const;
line_no size() const;

private:
    // read_file所用的辅助函数
    void store_file(ifstream&); // 存储输入文件
    void build_map(); // 将每个单词与一个行号集合相关联

    // 保存输入文件
    vector<string> lines_of_text;

    // 将单词与出现该单词的行的行号集合相关联
    map< string, set<line_no> > word_map;

    // 去掉标点并将字母变成小写
    static string cleanup_str(const string&);

};

#endif

```

TextQuery 类的实现文件（源文件）如下：

```

// TextQuery.cpp(for 15-41)
// TextQuery类的实现文件（源文件）
// 使用set容器存储行号
#include "TextQuery.hpp"
#include <iostream>

string TextQuery::text_line(line_no line) const
{
    if (line < lines_of_text.size())
        return lines_of_text[line];
    throw out_of_range("line number out of range");
}

// 读输入文件，将每行存储为lines_of_text的一个元素
void TextQuery::store_file(ifstream &is)
{
    string textline;
    while (getline(is, textline))
        lines_of_text.push_back(textline);
}

// 在输入vector中找以空白为间隔的单词
// 将单词以及出现该单词的行的行号一起放入word_map
void TextQuery::build_map()
{
    // 处理输入vector中的每一行
    for (line_no line_num = 0;
         line_num != lines_of_text.size();
         ++line_num)
    {

```

```

    // 一次读一个单词
    istringstream line(lines_of_text[line_num]);
    string word;
    while (line >> word)
        // 将行号加入到set容器中：
        // 若单词不在map容器中，下标操作将加入该单词
        word_map[cleanup_str(word)].insert(line_num);
    }
}

set<TextQuery::line_no>
TextQuery::run_query(const string &query_word) const
{
    // 注意，为了避免在word_map中加入单词，使用find函数而不用下标操作
    map<string, set<line_no>>::const_iterator
        loc = word_map.find(query_word);
    if (loc == word_map.end())
        return set<line_no>(); // 找不到，返回空的set对象
    else
        // 获取并返回与该单词关联的行号set对象
        return loc->second;
}

// 去掉标点并将字母变成小写
string TextQuery::cleanup_str(const string &word)
{
    string ret;
    for (string::const_iterator it = word.begin();
          it != word.end(); ++it) {
        if (!ispunct(*it))
            ret += tolower(*it);
    }
    return ret;
}

// 获取文本行数
vector<string>::size_type TextQuery::size() const
{
    return lines_of_text.size();
}

```

注意，此处定义的TextQuery类与习题10.32解答中定义的有如下区别：使用set容器而不是vector容器保存行号，增加了一个公有成员函数size以返回输入文本的行数。

主程序如下：

```

// 15-41.cpp
// 使用TextQuery类、Query类及Query_base类层次,
// 计算和打印如图15-4所示的查询:
// Query("fiery") & Query("bird") | Query("wind")
#include "Query.hpp"
#include "TextQuery.hpp"

string make_plural(size_t, const string&, const string&);

ifstream& open_file(ifstream&, const string&);

void print_results(const set<TextQuery::line_no>& locs,
                   const TextQuery &file)
{
    typedef set<TextQuery::line_no> line_nums;
    line_nums::size_type size = locs.size();

```

```

// 如果找到匹配的结果，则输出匹配的行数
cout << "match occurs "
    << size << " "
    << make_plural(size, "time", "s") << endl;

// 输出出现该单词的每一行
line_nums::const_iterator it = locs.begin();
for ( ; it != locs.end(); ++it) {
    cout << "\t(line "
        // 行号从1开始
        << (*it) + 1 << ") "
        << file.text_line(*it) << endl;
}
}

int main(int argc, char **argv)
{
    // 打开要在其中进行查询的文本文件
    ifstream infile;
    if (argc < 2 || !open_file(infile, argv[1])) {
        cerr << "No input file!" << endl;
        return EXIT_FAILURE;
    }

    TextQuery file;
    file.read_file(infile); // 读入文本，建立map容器

    typedef set<TextQuery::line_no> line_nums;
    // 构造查询
    Query q = Query("fiery") & Query("bird") | Query("wind");

    // 计算查询，获得匹配行的行号集合
    const line_nums &locs = q.eval(file);

    // 输出查询表达式
    cout << "\nExecuted Query for: " << q << endl;

    // 输出查询结果
    print_results(locs, file);

    return 0;
}

```

注意，主程序中所用到的make\_plural及open\_file函数与习题10.32的解答中定义的相同，此处不再赘述。

### 习题15.42

设计并实现下述增强中的一个：

- (a) 引入基于同一句子而不是同一行计算单词的支持。
- (b) 引入历史系统，用户可以用编号查阅前面的查询，并可以在其中增加内容或与其他查询组合。
- (c) 除了显示匹配数目和所有匹配行之外，允许用户对中间查询计算和最终查询指出要显示的行的范围。

#### 【解答】

选做(a)：引入基于同一句子而不是同一行计算单词的支持。

要支持基于同一句子而不是同一行计算单词，只需将文本按句子而不是按文本行存储到vector

容器。

将 TextQuery 类的成员函数 store\_file 修改如下：

```
// 读输入文件，将每个句子存储为 lines_of_text 的一个元素
void TextQuery::store_file(ifstream &is)
{
    char ws[] = { '\t', '\r', '\v', '\f', '\n' };
    char eos[] = { '?', '.', '!' };
    set<char> whiteSpace(ws, ws+5); // 空白符
    set<char> endOfSentence(eos, eos+3); // 句子结束符
    string sentence;
    char ch;

    while (is.get(ch)) { // 未遇到文件结束符
        if (!whiteSpace.count(ch)) // 非空白符
            sentence += ch;

        if (endOfSentence.count(ch)) { // 读完了句子
            lines_of_text.push_back(sentence);
            sentence.assign(""); // 将 sentence 清空，准备读下一个句子
        }
    }
}
```

此外，将 print\_results 函数中输出提示的“line”改为“sentence”。

注意，如果要同时支持基于同一句子及同一行计算单词，则可以在 TextQuery::store\_file 函数中增加形参进行控制，或者在 TextQuery 类中增加数据成员进行控制（见习题 15.34 的解答）；此处判断一个句子的结束只采用了句号、问号、感叹号三个符号作为判断条件，对英文句子的断句是否还有其他情况，有待思考。

## 模板与泛型编程

### 习题16.1

编写一个模板返回形参的绝对值。至少用三种不同类型的值调用模板。注意，在16.3节讨论编译器怎样处理模板实例化之前，你应该将每个模板定义和该模板的所有使用放在同一文件中。

#### 【解答】

```
// 16-1.cpp
// 编写一个模板函数返回形参的绝对值。
// 用三种不同类型的值调用该模板
#include <iostream>
using namespace std;

template <typename T>
T absVal(T val)
{
    return val > 0 ? val : -val;
}

int main()
{
    double dval = 0.88;
    float fval = -12.3;

    cout << absVal(-3) << endl;
    cout << absVal(dval) << endl;
    cout << absVal(fval) << endl;

    return 0;
}
```

### 习题16.2

编写一个函数模板，接受一个ostream引用和一个值，将该值写入流。用至少4种不同类型调用函数。通过写至cout、文件和stringstream来测试你的程序。

#### 【解答】

```
// 16-2.cpp
// 编写一个函数模板，接受一个ostream引用和一个值，将该值写入流。
// 用4种不同类型调用函数，
// 分别写至cout、文件和stringstream
#include <iostream>
#include <string>
#include <fstream>
#include <sstream>
using namespace std;
```

```

template <typename T1, typename T2>
T1& print(T1& s, T2 val)
{
    s << val;
    return s;
}

int main()
{
    double dval = 0.88;
    float fval = -12.3;
    string oristr = "this is a test", desstr;
    ostringstream oss(desstr);
    ofstream outFile("result.dat");

    // 写至cout
    print(cout, -3) << endl;;
    print(cout, dval) << endl;
    print(cout, fval) << endl;
    print(cout, oristr) << endl;

    // 写至文件
    print(outFile, -3) << endl;;
    print(outFile, dval) << endl;
    print(outFile, fval) << endl;
    print(outFile, oristr) << endl;
    outFile.close();

    // 写至stringstream
    print(oss, -3) << endl;;
    print(oss, dval) << endl;
    print(oss, fval) << endl;
    print(oss, oristr) << endl;

    // 将stringstream中的字符串输出到cout以进行验证
    cout << oss.str() << endl;

    return 0;
}

```

**习题16.3**

当调用两个string对象的compare时，传递用字符串字面值初始化的两个string对象。如果编写以下代码会发生什么？

```
compare ("hi", "world");
```

**【解答】**

该代码会出现编译错误。因为根据第一个实参"hi"可将模板形参T推断为char [3]，而根据第二个实参"world"可将模板形参T推断为char [6]，T被推断为两个不同的类型，所以编译器无法使用函数模板compare进行适当的实例化以满足需要。

**习题16.4**

什么是函数模板？什么是类模板？

**【解答】**

函数模板(function template)是可用于不同类型的函数的定义。函数模板用template关键字后接

用尖括号(<>)括住、以逗号分隔的一个或多个模板形参的列表来定义。

类模板(class template)是一个类定义，可以用来定义一组特定类型的类。类模板用template关键字后接用尖括号(<>)括住、以逗号分隔的一个或多个模板形参的列表来定义。

### 习题16.5

定义一个函数模板，返回两个值中较大的一个。

#### 【解答】

```
template <typename T>
T& bigger(const T& v1, const T& v2)
{
    return v1 > v2 ? v1 : v2;
}
```

### 习题16.6

类似于我们的queue简化版本，编写一个名为List的类模板，作为标准list类的简化版本。

#### 【解答】

```
template <class Type>
class List {
public:
    // 默认构造函数
    List();

    // 复制控制成员
    List(const List&);
    List& operator=(const List&);
    ~List();

    // 其他操作
    void insert(Type *ptr, const Type& value); // 在指定位置插入元素
    void del(Type *ptr); // 删除指定位置的元素
    Type *find(const Type& value); // 查找特定元素
    Type& getElem(Type *ptr); // 返回指定位置的元素
    bool empty() const; // 判断List是否为空
private:
    // ...
};
```

### 习题16.7

解释下面每个函数模板的定义并指出是否有非法的。改正所发现的错误。

- (a) template <class T, U, typename V> void f1(T, U, V) ;
- (b) template <class T> T f2(int &T) ;
- (c) inline template <class T> T foo(T, unsigned int\*) ;
- (d) template <class T> f4 (T, T) ;
- (e) typedef char Ctype ;
 template <typename Ctype> Ctype f5(Ctype a) ;

#### 【解答】

(a) 非法。模板类型形参前必须带有关键字typename(或class)，模板非类型形参前必须带有类型名，而这里的U作为f1的形参类型使用，应该是一个类型形参，所以应在模板形参表中U的前面加上class或typename。

(b) 非法。如果单纯从模板函数的定义语法来看，该定义是合法的。但是，模板形参T没有作为类

型在模板函数的形参表中出现，因此将无法对其进行模板实参推断，所以，该模板函数的定义是错误的。

- (c) 非法。`inline`不能放在关键字`template`之前，应放在模板形参表之后、函数返回类型之前。
- (d) 在标准C++中非法：没有指定函数`f4`的返回类型。（早期的C++版本可以接受：将返回类型隐式定义为`int`。）
- (e) 合法。定义了一个模板函数`f5`，该函数的返回类型与形参类型相同，均可绑定到任意类型（而不一定是`char`类型）。

### 习题16.8

如果有，解释下面哪些声明是错误的并说明为什么。

- (a) `template <class Type> Type bar(Type, Type) ;`  
`template <class Type> Type bar(Type, Type) ;`
- (b) `template <class T1, class T2> void bar(T1, T2) ;`  
`template <class C1, typename C2> void bar(C1, C2) ;`

### 【解答】

均正确。

### 习题16.9

编写行为类似于标准库中`find`算法的模板。你的模板应接受一个类型形参，该形参指定函数形参（一对迭代器）的类型。使用你的函数在`vector<int>`和`vector<string>`中查找给定值。

### 【解答】

```
// 16-9.cpp
// 编写行为类似于标准库中find算法的模板函数,
// 使用该函数在vector<int>和vector<string>中查找给定值
#include <iostream>
#include <string>
#include <vector>
using namespace std;

template <typename InIt, typename T>
InIt findElem(InIt first, InIt last, const T& val)
{
    while (first != last) {
        if (*first == val)
            return first;
        ++first;
    }
    return last;
}

int main()
{
    int ia[] = {1, 2, 3, 4, 5, 6, 7};
    string sa[] = {"this", "is", "Mary", "test", "example"};
    vector<int> ivec(ia, ia+7);
    vector<string> svec(sa, sa+5);
```

1. 两个声明相同，估计是原书的失误。

```

!= ivec.end())
cout << "found this element: " << *iit << endl;
else
    cout << "no such element" << endl;

vector<string>::iterator sit;
if ((sit = findElem(svec.begin(), svec.end(), "Mary"))
!= svec.end())
    cout << "found this element: " << *sit << endl;
else
    cout << "no such element" << endl;

return 0;
}

```

**习题16.10**

声明为typename的类型形参与声明为class的类型形参有区别吗？区别在哪里？

**【解答】**

在标准C++中，声明为typename的类型形参与声明为class的类型形参没有区别。但是，标准C++之前的系统有可能只支持使用关键字class来声明模板类型形参。

**习题16.11**

何时必须使用typename？

**【解答】**

如果要在函数模板内部使用在类中定义的类型成员，必须在该成员名前加上关键字typename，以告知编译器将该成员当作类型。

**习题16.12**

编写一个函数模板，接受表示未知类型迭代器的一对值，找出在序列中出现得最频繁的值。

**【解答】**

```

// mostFre接受一对迭代器first和last,
// 返回由first和last所指定的序列中出现最频繁的值
// 如果有多个值出现的次数相同，则返回其中最小的值
template <typename T>
typename T::value_type mostFre(T first, T last)
{
    allocator<typename T::value_type> alloc; // 用于分配内存的对象

    // 分配内存，用于保存输入序列的副本
    T newFirst = alloc.allocate(last - first);
    T newLast = newFirst + (last - first);

    // 将输入序列复制到新分配的内存空间
    std::uninitialized_copy(first, last, newFirst);

    std::sort (newFirst, newLast); // 对副本序列进行排序，使得相同的值出现在相邻位置

    std::size_t maxOccu = 0, occu = 0; // 最频繁的出现次数，当前值的出现次数
    T preIter = newFirst, maxOccuElemIt = newFirst; // 指向当前值的前一个值、
  // 当前出现最频繁的值

    while (newFirst != newLast) {

```

```

        if (*newFirst != *preIter) {           // 当前值与前一值不同
            if (occu > maxOccu) {             // 当前值的出现次数为目前的最大次数
                maxOccu = occu;                 // 修改最大次数
                maxOccuElemIt = preIter;       // 修改指向当前出现最频繁的值的迭代器
            }
            occu = 0;
        }
        ++occu;
        preIter = newFirst;
        ++newFirst;
    }

    // 最后一个值的出现次数与目前的最大次数进行比较
    if (occu > maxOccu) {                 // 最后一个值的出现次数为目前的最大次数
        maxOccu = occu;                   // 修改最大次数
        maxOccuElemIt = preIter;         // 修改指向当前出现最频繁的值的迭代器
    }

    .
}

return *maxOccuElemIt;
}

```

注意，上述解答中采用的问题解决方案为：对输入序列进行排序，然后在有序序列中寻找目标值。为了不改变输入序列本身，首先对输入序列进行了复制，然后针对副本序列进行排序及后续操作；另外，因为其中使用了标准库算法sort以及内存管理类allocator（见18.1.2节），使得该解决方案有一定的局限性，只适用于vector容器的迭代器。为了提高通用性，下面给出另一个解答：不用allocator类进行内存分配，而是将输入序列复制到一个vector容器中。该方案适用于vector、list、deque等容器的迭代器。

```
// 解答二:  
// mostFre接受一对迭代器first和last,  
// 返回由first和last所指定的序列中出现最频繁的值  
// 如果有多个值出现的次数相同, 则返回其中最小的值  
template <typename T>  
typename T::value_type mostFre(T first, T last)  
{  
    // 计算需分配内存的大小  
    std::size_t amount = 0;  
    T start = first;  
    while (start != last) {  
        amount++;  
        start++;  
    }  
  
    // 定义类型别名  
    typedef std::vector<typename T::value_type> VecType;  
  
    // 创建vector对象, 用于保存输入序列的副本  
    VecType vec(amount);  
  
    VecType::iterator newFirst = vec.begin();  
    VecType::iterator newLast = vec.end();  
  
    // 将输入序列复制到vector对象  
    std::uninitialized_copy(first, last, newFirst);  
  
    std::sort (newFirst, newLast); // 对副本序列进行排序,  
                                // 使得相同的值出现在相邻位置
```

```

std::size_t maxOccu = 0, occu = 0;           // 最频繁的出现次数,
  // 当前值的出现次数
VecType::iterator preIter = newFirst;        // 指向当前值的前一个值
VecType::iterator maxOccuElemIt = newFirst;   // 指向当前出现最频繁的值

while (newFirst != newLast) {
    if (*newFirst != *preIter) {               // 当前值与前一值不同
        if (occu > maxOccu) {                 // 当前值的出现次数为目前的最大次数
            maxOccu = occu;                   // 修改最大次数
            maxOccuElemIt = preIter;         // 修改指向当前出现最频繁的
  // 值的迭代器
        }
        occu = 0;
    }
    ++occu;
    preIter = newFirst;
    ++newFirst;
}

// 最后一个值的出现次数与目前的最大次数进行比较
if (occu > maxOccu) {                         // 最后一个值的出现次数为目前的最大次数
    maxOccu = occu;                           // 修改最大次数
    maxOccuElemIt = preIter;                  // 修改指向当前出现最频繁的值的迭代器
}

return *maxOccuElemIt;
}

```

### 习题16.13

编写一个函数，接受一个容器的引用并打印该容器的元素。使用容器的size\_type 和size成员控制打印元素的循环。

#### 【解答】

```

// 16-13.cpp
// 编写一个函数，接受一个容器的引用并打印该容器的元素。
// 使用容器的size_type 和size成员控制打印元素的循环
#include <iostream>
#include <string>
#include <vector>
using namespace std;

template <typename Parm>
void print(const Parm& c)
{
    typename Parm::size_type index = 0;
    while (index != c.size()) {
        cout << c[index] << ' ';
        ++index;
    }
}

int main()
{
    int ia[] = {1, 2, 1, 4, 1, 6, 1};
    string sa[] = {"this", "is", "Mary", "test", "example"};
    vector<int> ivec(ia, ia+7);
    vector<string> svec(sa, sa+5);

```

```

    print(ivec);
    cout << endl;
    print(svec);

    return 0;
}

```

注意，对不支持下标操作的容器，不能使用上述print函数。

### 习题16.14

重新编写习题16.13的函数，使用从begin和end返回的迭代器来控制循环。

#### 【解答】

```

template <typename Parm>
void print(const Parm& c)
{
    typename Parm::const_iterator iter;
    for (iter = c.begin(); iter != c.end(); ++iter)
        cout << *iter << ' ';
}

```

### 习题16.15

编写可以确定数组长度的函数模板。

#### 【解答】

可以使用非类型模板形参编写如下函数模板：

```

template <typename T, std::size_t N>
std::size_t size(T (&arr)[N])
{
    return N;
}

```

### 习题16.16

将7.2.4节的printValues函数重新编写为可用于打印不同长度数组内容的函数模板。

#### 【解答】

```

template <typename T, std::size_t N>
void printValues(T (&arr)[N])
{
    for (std::size_t i = 0; i != N; ++i)
        std::cout << arr[i] << std::endl;
}

```

### 习题16.17

在3.3.2节的“关键概念”中，我们注意到，C++程序员习惯于使用!=而不用<，解释这一习惯的基本原理。

#### 【解答】

C++程序员经常会用到标准库的内容。标准库中的类及泛型算法大多定义为模板类及模板函数，它们的实例化版本是否合法取决于用作模板实参的类型是否支持模板所要求的操作。而对于用作模板实参的类型而言，支持相等操作（==和!=）的可能性比支持关系操作（如<）的可能性更大，因此，

使用!=而不用<这一习惯，更有利于保证所编写程序的正确性。

### 习题16.18

本节中我们提到应该慎重地编写compare中的比较，以避免要求类型同时具有<和>操作符；另一方面，往往假定类型既有==又有!=。解释为什么这一看似不一致的处理实际上反映了良好的编程风格。

#### 【解答】

从类设计的角度而言，如果一个类定义了operator==，则它也应该定义operator!=，因为用户一般会认为如果这两个操作符中的一个存在，则另一个也应该存在，所以一个设计良好的类应该注意到并满足用户的这一使用习惯，从而，往往可以假定类型既有==又有!=。

本节中所提到的应该慎重地编写compare中的比较，以避免要求类型同时具有<和>操作符，是从模板设计的角度，减少对可用于模板函数的用户类型的要求，也是为了方便用户的使用。

因此，这些处理实际上都是为了让用户更方便地使用我们的设计，反映了良好的编程风格。

### 习题16.19

什么是实例化？

#### 【解答】

所谓“实例化”，指的是产生模板的特定类型实例的过程。模板不能直接使用，必须在使用时进行实例化。类模板的实例化在引用实际模板类类型时进行，函数模板的实例化在调用它或用它对函数指针进行初始化或赋值时进行。

### 习题16.20

在模板实参推断期间发生什么？

#### 【解答】

根据函数调用中给出的实参确定模板实参的类型和值。

### 习题16.21

指出对模板实参推断中涉及的函数实参允许的类型转换。

#### 【解答】

const转换：接受const引用或const指针的函数可以分别用非const对象的引用或指针来调用，无需产生新的实例化。如果函数接受非引用类型，形参类型和实参都忽略const，即无论传递const或非const对象给接受非引用类型的函数，都使用相同的实例化。

数组或函数到指针的转换：如果模板形参不是引用类型，则对数组或函数类型的实参应用常规指针转换。数组实参将当作指向其第一个元素的指针，函数实参当作指向函数类型的指针。

### 习题16.22

对于下面的模板：

```
template <class Type>
Type calc (const Type* array, int size);
```

```
template <class Type>
Type fcn(Type p1, Type p2);
```

下面这些调用有错吗？如果有，哪些是错误的？为什么？

```
double dobj; float fobj; char cobj;
int ai[5] = { 511, 16, 8, 63, 34 };
```

- (a) calc(cobj, 'c');
- (b) calc(dobj, fobj);
- (c) fcn(ai, cobj)<sup>1</sup>;

### 【解答】

错误的有：

- (a) 实参cobj的类型为char，但是，不能使用函数模板calc产生第一个形参为非指针类型的函数实例。
- (b) 实参dobj的类型为double，但是，不能使用函数模板calc产生第一个形参为非指针类型的函数实例。
- (c) 函数模板fcn中两个形参的类型必须是相同的，而函数调用fcn(ai, cobj)中给出的两个实参数类型不同，不能进行实例化。

### 习题16.23

标准库函数max接受单个类型形参，可以传递int和double对象调用max吗？如果可以，怎样做？如果不能，为什么？

### 【解答】

可以传递int和double对象调用max：只需使用强制类型转换将其中一个对象转换为int或double类型，使其与另一对象类型相同即可。

### 习题16.24

在16.2.1节我们看到，对于具有单个模板类型形参的compare版本，传给它的实参必须完全匹配，如果想要用兼容类型如int和short调用该函数，可以使用显式模板实参指定int或short作为形参类型。编写程序使用具有一个模板形参的compare版本，使用允许你传递int和short类型实参的显式模板实参调用compare。

### 【解答】

```
// 16-24.cpp
// 使用具有一个模板形参的compare版本,
// 使用允许传递int和short类型实参的显式模板实参调用compare
#include <iostream>
using namespace std;

template <typename T>
int compare(const T& v1, const T& v2)
{
    if (v1 < v2) return -1;
    if (v2 < v1) return 1;
    return 0;
}
```

1. 此处英文原文中缺少一个右括号。

```

int main()
{
    short sval = 123;
    int ival = 1024;
    cout << compare(static_cast<int>(sval), ival) << endl;
    cout << compare(sval, static_cast<short>(ival)) << endl;
    cout << compare<short>(sval, ival) << endl;
    cout << compare<int>(sval, ival) << endl;

    return 0;
}

```

**习题16.25**

使用显式模板实参，使得可以传递两个字符串字面值调用compare。

**【解答】**

只需按如下方式使用显式模板实参：

```
compare<std::string>("mary", "mac")
```

亦可采用如下强制类型转换方式：

```
compare(static_cast<std::string>("mary"),
        static_cast<std::string>("mac"))
```

或

```
compare(std::string("mary"), std::string("mac"))
```

**习题16.26**

对于下面的sum模板定义：

```
template <class T1, class T2, class T3> T1 sum(T2, T3);
```

解释下面的每个调用。如果有，指出哪些是错误的；对每个错误，解释错在哪里。

```

double dobj1, dobj2; float fobj1, fobj2; char cobj1, cobj2;

(a) sum(dobj1, dobj2);
(b) sum<double, double, double>(fobj1, fobj2);
(c) sum<int>(cobj1, cobj2);
(d) sum<double, ,double>(fobj2, dobj2);

```

**【解答】**

(a) 错误。没有为模板类型形参T1指定相应的类型实参。

(b) 正确。编译器将根据显式模板实参为该调用产生函数实例double sum(double, double)，并将两个函数实参由float类型转换为double类型来调用该实例。

(c) 正确。编译器将根据显式模板实参及函数实参为该调用产生函数实例int sum(char, char)。

(d) 错误。只有最右边形参的显式模板实参可以省略，不能用“,”代替被省略的显式模板实参。

**习题16.27**

确定你的编译器使用的是哪种编译模型。编写并调用函数模板，在保存未知类型对象的vector中查找中间值。（注：中间值是这样一个值，一半元素比它大，一半元素比它小。）用常规方式构造

你的程序：函数定义应放在一个文件中，它的声明放在一个头文件中，定义和使用函数模板的代码应包含该头文件。

### 【解答】

定义函数模板的头文件如下：

```
// median.hpp
// 定义median函数模板的头文件
// median函数模板在保存未知类型对象的vector中查找中间值
#ifndef MEDIAN_H
#define MEDIAN_H
#include <vector>
#include <algorithm>
using std::vector;

// 如果vector中有中间值，则返回true，并将中间值记录在第二个参数中；
// 否则，返回false
template <typename T>
bool median(const vector<T>&, T&);

#include "median.cpp" // 引入函数模板的实现文件
#endif
```

函数模板的实现文件如下：

```
// median.cpp
// 定义median函数模板的实现文件（源文件）
template <typename T>
bool median(const vector<T>& c, T& m)
{
    // 构造temp为c的副本
    vector<T> temp(c);

    // 如果容器中包含偶数个元素，则没有中间值，返回false
    if (temp.size() % 2 == 0)
        return false;

    // 将元素排序
    sort(temp.begin(), temp.end());

    // 判断中间点元素是否为中间值，是则返回true，并用m记录中间值；
    // 否则返回false
    vector<T>::size_type index = temp.size() / 2;
    if (temp[index] > temp[index - 1]
        && temp[index] < temp[index + 1]) {
        m = temp[index];
        return true;
    }
    else
        return false;
}
```

使用函数模板的主程序如下：

```
// 16-27.cpp
// 调用函数模板median，在保存int类型对象的vector中查找中间值

#include "median.hpp"
#include <iostream>
using namespace std;
```

```

int main()
{
    int ia1[] = {1, 2, 3, 4, 5, 6, 7};
    int ia2[] = {1, 2, 3, 4};
    int ia3[] = {1, 2, 2, 3, 4, 5, 6};
    vector<int> ivec1(ia1, ia1+7);
    vector<int> ivec2(ia2, ia2+4);
    vector<int> ivec3(ia3, ia3+7);
    int m;

    if (median(ivec1, m))
        cout << "median: " << m << endl;
    else
        cout << "no median" << endl;

    if (median(ivec2, m))
        cout << "median: " << m << endl;
    else
        cout << "no median" << endl;

    if (median(ivec3, m))
        cout << "median: " << m << endl;
    else
        cout << "no median" << endl;

    return 0;
}

```

注意，Microsoft Visual C++ .NET 2003支持模板的包含编译模型，但要注意不要将模板的实现文件显式地加入到project中，否则会引起编译错误。

### 习题16.28

如果所用的编译器支持分别编译模型，将类模板的成员函数和static数据成员的定义放在哪里？为什么？

#### 【解答】

如果所用的编译器支持分别编译模型，则类模板的内联成员函数应该与类模板的定义一起放在头文件中，而非内联成员函数和static数据成员的定义应该放在实现文件中。因为内联函数的定义必须在函数被扩展时能为编译器所见，而非内联成员函数一般而言并不希望让用户看见，所以将其放在实现文件中（同时，在类的实现文件中应该导出（export）类模板，以便让编译器了解要记住该模板定义，并自动跟踪相关的模板定义）。

另外，如果不想导出整个类模板，而只是导出个别成员，则非导出成员的定义应放在头文件中，并且实现文件中不在类模板本身指定export，而是在被导出的特定成员定义上指定export。

### 习题16.29

如果你的编译器使用包含模型，将那些模板成员定义放在哪里？为什么？

#### 【解答】

如果编译器使用包含模型，则那些模板成员定义均可放在类的实现文件中。因为这种情况下，需在类模板的头文件中使用#include指示引入相关类模板的实现文件，从而编译器在使用该类模板的地方能够看到类模板成员的定义。也可以直接将整个模板的定义放在头文件中，效果相同，但可能会导

致头文件过长，不便于阅读。

### 习题16.30

如果有，指出下面类模板声明（或声明对）中哪些是非法的。

- (a) template <class Type> class C1;
- template <class Type, int size> class C1;
- (b) template <class T, U, class V> class C2;
- (c) template <class C1, typename C2> class C3 { };
- (d) template <typename myT, class myT> class C4 { };
- (e) template <class Type, int \*ptr> class C5;
- template <class T, int \*pi> class C5;

### 【解答】

非法的有(b)和(d)。(b)模板形参U之前缺少关键字class（或typename）或类型名，(d)两个模板形参myT同名。

### 习题16.31

下面List的定义不正确，怎样改正？

```
template <class elemType> class ListItem;
template <class elemType> class List {
public:
    List<elemType>();
    List<elemType>(const List<elemType> &);
    List<elemType>& operator=(const List<elemType> &);
    ~List();
    void insert(ListItem *ptr, elemType value);
    ListItem *find(elemType value);
private:
    ListItem *front;
    ListItem *end;
};
```

### 【解答】

错误在于：在类模板List的定义体中使用类模板ListItem时未指定模板实参。将ListItem改为ListItem<elemType>即可。

### 习题16.32

为Queue类实现赋值操作符。

### 【解答】

```
template <class Type>
Queue<Type>& Queue<Type>::operator = (const Queue &orig)
{
    destroy();
    copy_elems(orig);

    return *this;
}
```

注意，赋值操作实现对Queue对象的复制。复制Queue对象时应该对队列中的元素进行复制，而不仅是复制队头和队尾指针。为了加深读者的印象，下面给出一个不使用Queue类中其他成员函数的独

立赋值操作符的实现:

```
template <class Type>
Queue<Type>& Queue<Type>::operator = (const Queue &orig)
{
    QueueItem<Type> *p = orig.head, *q;
    while (p != 0) {
        q = new QueueItem<Type>(p->item);
        if (p == orig.head)
            head = tail = q;
        tail->next = q;
        tail = q;
        p = p->next;
    }
    return *this;
}
```

### 习题16.33

解释在copy\_elems函数中新创建的Queue对象中的next指针怎样设置。

**【解答】**

copy\_elems函数从形参Queue对象中复制元素到调用copy\_elems函数的Queue对象中。新元素(QueueItem对象)的创建由push操作完成, push操作调用QueueItem类的构造函数创建新元素。该构造函数将新创建的元素(QueueItem对象)的next指针设置为0, push操作将该元素追加到Queue对象的末尾。

### 习题16.34

编写16.1.2节习题中定义的List类的成员函数定义。

**【解答】**

参照本节给出的Queue类定义, 对习题16.6的解答中给出的List类进行补充, 可得到如下List类定义:

```
template <class Type> class List;
template <class Type> class ListItem {
    friend class List<Type>;
    // 私有类: 没有public部分
    ListItem(const Type &t): item(t), next(0) { }
    Type item;           // 元素中存储的数据
    ListItem *next;      // 指向下一元素的指针
};

template <class Type> class List {
public:
    // 默认构造函数
    List() : front(0), end(0) { }

    // 复制控制成员
    List(const List& l) : front(0), end(0)
    {
        copy_elems(l)
    }

    List& operator=(const List&);
```

```

~List()
{
    destroy();
}

// 其他操作
void insert(ListItem<Type> *ptr, const Type& value);
void del(ListItem<Type> *ptr);
ListItem<Type> *find(const Type& value);
ListItem<Type> *first()
{
    return front;
}

ListItem<Type> *last()
{
    return end;
}

bool empty() const // 判断List是否为空
{
    return front == 0;
}

Type& getElem(ListItem<Type> *ptr)
{
    // 不检查ptr
    return ptr -> item;
}

private:
    ListItem<Type> *front, *end; // 指向List中头尾元素的指针
    void destroy();
    void copy_elems(const List&);

};

// 删除List中所有元素
template <class Type>
void List<Type>::destroy()
{
    while (!empty())
        del(front);
}

// 删除ptr所指向的元素
template <class Type>
void List<Type>::del(ListItem<Type> *ptr)
{
    // 不检查ptr
    ListItem<Type>* p = front;

    // 获得ptr所指元素的前一元素的指针p
    while (p != ptr && p != 0 && p -> next != ptr)
        p = p -> next;

    if (p != 0) // 找到这样的指针p: 说明ptr指向List中的元素
        if (p == ptr) { // 要删除的是第一个元素
            front = ptr -> next;
        }
        else {
            p -> next = ptr -> next;
        }
}

```

```

        if (ptr == end)
            end = p -> next;
        delete ptr;
    }
    else
        throw out_of_range("no such element");
}

// 在ptr所指元素的后面插入元素
template <class Type>
void List<Type>::insert(ListItem<Type> *ptr, const Type& val)
{
    // 不检查ptr
    // 创建ListItem对象
    ListItem<Type> *pt = new ListItem<Type>(val);

    // 将ListItem插入List
    if (empty()) // 原List为空
        front = pt; // List现在只有一个元素
    else { // 将新元素插入到ptr所指元素的后面
        pt -> next = ptr -> next;
        ptr -> next = pt;
    }

    if (ptr == end)
        end = pt; // 修改List的end指针
}

// 将orig中的元素复制到这个List中
template <class Type>
void List<Type>::copy_elems(const List &orig)
{
    // 当pt == 0 (即到达orig.end时) 循环结束
    for (ListItem<Type> *pt = orig.front; pt; pt = pt->next) {
        insert(end, pt -> item);
    }
}

// 赋值操作符: 复制所有元素
template <class Type>
List<Type>& List<Type>::operator = (const List &orig)
{
    front = end = 0;
    copy_elems(orig);
    return *this;
}

// 查找值为value的元素, 返回其指针
template <class Type>
ListItem<Type>* List<Type>::find(const Type& value)
{
    ListItem<Type>* pt = front;
    while (pt && pt -> item != value)
        pt = pt -> next;
    return pt;
}

```

注意, 这里给出的是一个简单的例子List类, 其中的操作为简单起见未对形参指针进行检查, 更为完善的List类可参照标准库中给出的list容器类进行定义。

**习题16.35**

编写14.7节中描述的CheckedPtr类的泛型版本。

**【解答】**

将CheckedPtr类定义为如下类模板：

```
template <typename T>
class CheckedPtr {
public:
    // 没有默认构造函数：CheckedPtr对象必须绑定到一个对象
    CheckedPtr(T *b, T *e): beg(b), end(e), curr(b) {}

    // 自增及自减操作
    // 前缀形式
    CheckedPtr& operator++();
    CheckedPtr& operator--();
    // 后缀形式
    CheckedPtr operator++(int);
    CheckedPtr operator--(int);

    // 下标及解引用操作
    T& operator[](const size_t index);
    const T& operator[](const size_t index) const;
    T& operator*();
    const T& operator*() const;
private:
    T* beg; // 指向数组的起始位置（第一个元素）
    T* end; // 指向数组中最后一个元素的下一个位置
    T* curr; // 数组中的当前位置
};

// 前缀式：返回被自增/自减对象的引用
template <typename T>
CheckedPtr<T>& CheckedPtr<T>::operator++()
{
    if (curr == end)
        throw out_of_range
            ("increment past the end of CheckedPtr");
    ++curr;
    return *this;
}

template <typename T>
CheckedPtr<T>& CheckedPtr<T>::operator--()
{
    if (curr == beg)
        throw out_of_range
            ("decrement past the beginning of CheckedPtr");
    --curr;
    return *this;
}

// 后缀式：对象自增/自减，但返回未改变的值
template <typename T>
CheckedPtr<T> CheckedPtr<T>::operator++(int)
{
    // 此处无需检查，对前自增操作的调用将进行检查
    CheckedPtr ret(*this); // 保存当前值
    ++*this; // 向前移动一个元素，对增量进行检查
}
```

```

        return ret;           // 返回被保存状态
    }

template <typename T>
CheckedPtr<T> CheckedPtr<T>::operator--(int)
{
    // 此处无需检查，对前自减操作的调用将进行检查
    CheckedPtr ret(*this); // 保存当前值
    --*this;               // 向前移动一个元素，对减量进行检查
    return ret;             // 返回被保存状态
}

template <typename T>
T& CheckedPtr<T>::operator[](const size_t index)
{
    if (beg + index >= end)
        throw out_of_range
            ("invalid index");
    return *(beg + index);
}

template <typename T>
const T& CheckedPtr<T>::operator[](const size_t index) const
{
    if (beg + index >= end)
        throw out_of_range
            ("invalid index");
    return *(beg + index);
}

template <typename T>
T& CheckedPtr<T>::operator*()
{
    if (curr == end)
        throw out_of_range
            ("invalid current pointer");
    return *curr;
}

template <typename T>
const T& CheckedPtr<T>::operator*() const
{
    if (curr == end)
        throw out_of_range
            ("invalid current pointer");
    return *curr;
}

```

注意，相对于14.7节中定义的CheckedPtr类，主要的修改在于：增加模板形参表，在使用具体类型int的地方改用模板形参T，使用CheckedPtr类名的地方改用CheckedPtr<T>。

可参照习题14.25、习题14.26和习题14.30的解答进一步完善上述定义的CheckedPtr类模板。

### 习题16.36

每个带标号的语句，会导致实例化吗？如果会，解释导致什么样的实例化。

```

template <class T> class Stack { };
void f1(Stack<char>);           // (a)

```

```

class Exercise {
    Stack<double> &rsd;                                // (b)
    Stack<int> si;                                    // (c)
};
int main() {
    Stack<char> *sc;                                // (d)
    f1(*sc);   // (e)
    int iObj = sizeof(Stack< string >);           // (f)
}

```

**【解答】**

(a) 不会导致实例化：函数声明不会导致实例化。

(b) 不会导致实例化：定义引用不会导致实例化。

(c) 会导致实例化：在创建Exercise对象时会实例化Stack<int>类及其默认构造函数（严格来说，这样的实例化不是由语句(c)直接导致的，而是由定义Exercise对象的语句导致的）。

(d) 不会导致实例化：定义指针不会导致实例化。

(e) 会导致实例化：调用函数f1时需创建形参对象，此时导致实例化Stack<char>类及其默认构造函数。注意，题目此处有误：指针sc尚未赋值就使用了。

(f) 会导致实例化：sizeof操作需要使用具体类，此时导致实例化Stack<string>。

**习题16.37**

下面哪些模板实例化是有效的？解释为什么实例化无效。

```

template <class T, int size> class Array { /* . . . */ };
template <int hi, int wid> class Screen { /* . . . */ };

```

(a) const int hi = 40, wi = 80; Screen<hi, wi+32> sObj;

(b) const int arr\_size = 1024; Array<string, arr\_size> a1;

(c) unsigned int asize = 255; Array<int, asize> a2;

(d) const double db = 3.1415; Array<double, db> a3;

**【解答】**

有效的模板实例化包括(a)和(b)。

(c)之所以无效，是因为非类型模板实参必须是编译时常量表达式，不能用变量asize作模板实参。

(d)之所以无效，是因为db是double型常量，而该模板实例化所需要的非类型模板实参为int型常量。

**习题16.38**

编写Screen类模板，使用非类型形参定义Screen的高度和宽度。

**【解答】**

参照习题12.13的解答，可编写Screen类模板如下：

```

template<int hi, int wid>
class Screen {
public:
    typedef std::string::size_type index;

```

1. 此处原题有误：指针 sc 尚未赋值就使用了。

```

Screen() : contents(hi * wid, '#'),
           cursor(0), height(hi), width(wid) {}

Screen(const std::string& cont);

char get() const
{
    return contents[cursor];
}

char get(index ht, index wd) const;

index get_cursor() const
{
    return cursor;
}

Screen& move(index r, index c);
Screen& set(char);
Screen& display(ostream &os);
const Screen& display(ostream &os) const;
private:
    std::string contents;
    index cursor;
    index height, width;
};

template<int hi, int wid>
Screen<hi, wid>::Screen<hi, wid>(const std::string& cont) :
    cursor(0), height(hi), width(wid)
{
    // 将整个屏幕内容置为空格
    contents.assign(hi * wid, ' ');
    // 用形参string对象的内容设置屏幕的相应字符
    if (cont.size() != 0)
        contents.replace(0, cont.size(), cont);
}

template<int hi, int wid>
char Screen<hi, wid>::get(index r, index c) const
{
    index row = r * width;
    return contents[row + c];
}

template<int hi, int wid>
Screen<hi, wid>& Screen<hi, wid>::set(char c)
{
    contents[cursor] = c;
    return *this;
}

template<int hi, int wid>
Screen<hi, wid>& Screen<hi, wid>::move(index r, index c)
{
    // 行、列号均从0开始
    if (r >= height || c >= width) {
        cerr << "invalid row or column" << endl;
        throw EXIT_FAILURE;
    }

    index row = r * width;
}

```

```

        cursor = row + c;
        return *this;
    }

template<int hi, int wid>
Screen<hi, wid>& Screen<hi, wid>::display(ostream &os)
{
    string::size_type index = 0;
    while (index != contents.size()) {
        os << contents[index];
        if ((index+1) % width == 0) {
            os << '\n';
        }
        ++index;
    }
    return *this;
}

template<int hi, int wid>
const Screen<hi, wid>& Screen<hi, wid>::display(ostream &os) const
{
    string::size_type index = 0;
    while (index != contents.size()) {
        os << contents[index];
        if ((index+1) % width == 0) {
            os << '\n';
        }
        ++index;
    }
    return *this;
}

```

**习题16.39**

为Screen模板类实现输入和输出操作符。

**【解答】**

为Screen模板类实现的输入和输出操作符如下：

```

template<int hi, int wid>
std::ostream& operator <<(std::ostream &os,
                           const Screen<hi, wid> &s)
{
    os << "height: " << s.height
    << "width: " << s.width
    << "contents: " << s.contents;
    return os;
}

template<int hi, int wid>
std::istream& operator >>(std::istream &is, Screen<hi, wid> &s)
{
    std::string cont;
    is >> s.height >> s.width >> cont;

    // 将整个屏幕内容置为空格
    s.contents.assign(s.height * s.width, ' ');

    // 用读入的字符串设置屏幕的相应字符
    if (cont.size() != 0)
        s.contents.replace(0, cont.size(), cont);
}

```

```

    return is;
}

```

**习题16.40**

要使输入和输出操作符能够工作，Screen类需要友元吗？如果需要，要哪些友元？解释为什么需要每个友元声明。

**【解答】**

因为输入和输出操作符需要访问Screen类的私有数据成员，所以要使输入和输出操作符能够工作，Screen类需要将这两个操作符设为友元，可在screen类的定义体内增加如下友元声明：

```

friend std::ostream&
operator << <hi, wid> (std::ostream&, const Screen<hi, wid>&);

friend std::istream&
operator >> <hi, wid> (std::istream&, Screen<hi, wid>&);

```

**习题16.41**

Queue类中的operator<<的友元声明是：

```

friend std::ostream&
operator<< <Type> (std::ostream&, const Queue<Type>&);

```

将Queue形参写为const Queue&而不是const Queue<Type>&，会有什么结果？

**【解答】**

如果将Queue形参写为const Queue&而不是const Queue<Type>&，会出现编译错误。因为Queue类是一个模板类，在使用时必须指定模板实参。

**习题16.42**

编写一个输入操作符，读一个istream对象并将读取的值放入一个Queue对象中。

**【解答】**

可编写输入操作符如下：

```

template<class Type>
istream& operator>> (istream &is, Queue<Type> &q)
{
    Type val;
    while (is >> val)
        q.push(val);
    return is;
}

```

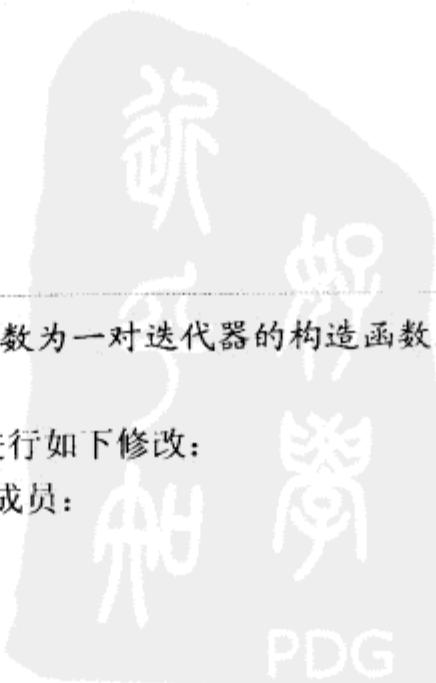
**习题16.43**

为你的List类增加assign成员和一个参数为一对迭代器的构造函数。

**【解答】**

可对习题16.34解答中给出的List类进行如下修改：

(1) 在类定义体中public部分增加如下成员：



```

template <class Iter>
List(Iter first, Iter last) : front(first), end(last)
{
    copy_elems(beg, last)
}

template <class Iter>
void assign(Iter, Iter);

```

(2) 在List类定义体中private部分增加如下成员:

```
template <class Iter> void copy_elems(Iter, Iter);
```

assign 和 copy\_elems 函数可在类定义体外实现如下:

```

template <class T> template <class Iter>
void List<T>::assign(Iter first, Iter last)
{
    destroy();
    copy_elems(first, last);
}

template <class T> template <class Iter>
void List<T>::copy_elems(Iter first, Iter last)
{
    while(first != last) {
        insert(end, first -> item);
        first = first -> next;
    }
    // 插入最后一个元素
    insert(end, first -> item);
}

```

### 习题16.44

为了举例说明怎样实现类模板，我们实现了自己的Queue类。可以简化实现的一种方式可能是将Queue定义在一个现存的标准库容器类型之上，用这种方法，可以避免必须管理Queue元素的分配和回收。用std::list<sup>1</sup>保存实际的Queue元素，重新实现Queue类。

#### 【解答】

用std::list保存实际的Queue元素，Queue类可实现如下:

```

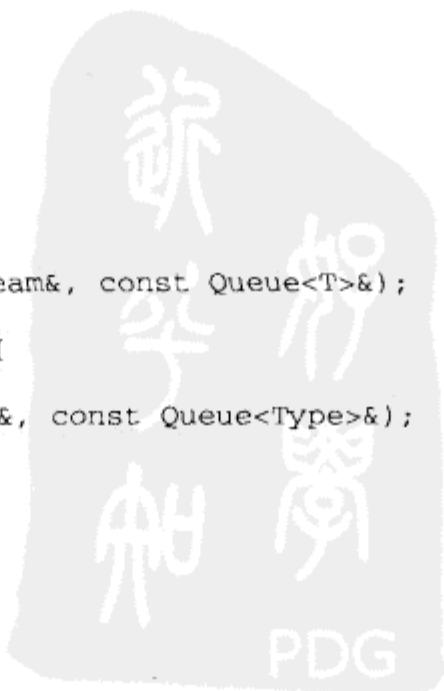
// Queue.hpp(for16-44)
// 定义Queue类
#ifndef QUEUE_H
#define QUEUE_H
#include <iostream>
#include <list>

template <class Type> class Queue;
template <class T>
std::ostream& operator<<(std::ostream&, const Queue<T>&);

template <class Type> class Queue {
    friend std::ostream&
    operator<< <Type> (std::ostream&, const Queue<Type>&);
public:

```

1. 此处英文原书误为List, L应小写。



```

Queue() { }
public:
    // 接受一对迭代器形参的构造函数
    template <class It> Queue(It beg, It end): items(beg, end) { }

    // 用输入范围的元素给当前Queue赋值
    template <class Iter> void assign(Iter beg, Iter end)
    {
        items.assign(beg, end);
    }

    // 返回队头元素
    // 不进行检查：空Queue上的front操作是未定义的
    Type& front()
    {
        return items.front();
    }

    const Type &front() const
    {
        return items.front();
    }

    // 在队尾增加元素
    void push(const Type &t)
    {
        items.push_back(t);
    }

    // 删除队头元素
    void pop()
    {
        items.erase(items.begin());
    }

    // 如果Queue中没有元素，则返回true
    bool empty() const
    {
        return items.empty();
    }
private:
    std::list<Type> items;
};

template <class Type>
std::ostream& operator<<(std::ostream &os, const Queue<Type> &q)
{
    os << "< ";
    typename std::list<Type>::const_iterator beg = q.items.begin();
    while (beg != q.items.end()) {
        os << *beg << " ";
        ++beg;
    }
    os << ">";
    return os;
}

#endif

```

注意, 因为笔者手头的C++编译器只支持模板的包含编译模型, 所以为了节省篇幅, 本习题解答中给出的模板大多定义在同一文件中。因为如果编译器不支持模板的分别编译模型, 则区分定义模板的头文件和实现文件并没有太大的实际意义。

### 习题16.45

实现一个Handle类的自己的版本。

#### 【解答】

```
// Handle.hpp(for 16-45)
// 定义泛型句柄类Handle
#ifndef HANDLE_H
#define HANDLE_H
#include <cstddef>

// 泛型句柄类
template <class T>
class Handle {
public:
    // 若p为0, 则为未绑定的句柄
    Handle(T *p = 0): ptr(p), use(new std::size_t(1)) { }

    // 支持指针行为的重载操作符
    T& operator*();
    T* operator->();
    const T& operator*() const;
    const T* operator->() const;

    // 复制控制: 提供常规指针行为, 但最后一个句柄删除基础对象
    Handle(const Handle& h): ptr(h.ptr), use(h.use)
    {
        ++*use;
    }

    Handle& operator=(const Handle&);

    ~Handle()
    {
        rem_ref();
    }
private:
    T* ptr;           // 共享的对象
    std::size_t *use; // 使用计数: 多少个句柄指向*ptr
    void rem_ref()
    {
        if (--*use == 0) {
            delete ptr;
            delete use;
        }
    };
};

template <class T>
inline Handle<T>& Handle<T>::operator=(const Handle &rhs)
{
    ++*rhs.use;      // 防止自身赋值
    rem_ref();        // 使用计数减1并在必要时删除指针
    ptr = rhs.ptr;
    use = rhs.use;
    return *this;
}
```

```

template <class T>
inline T& Handle<T>::operator*()
{
    if (ptr) return *ptr;
    throw std::runtime_error
        ("dereference of unbound Handle");
}

template <class T>
inline T* Handle<T>::operator->()
{
    if (ptr) return ptr;
    throw std::runtime_error
        ("access through unbound Handle");
}

template <class T>
inline const T& Handle<T>::operator*() const
{
    if (ptr) return *ptr;
    throw std::runtime_error
        ("dereference of unbound Handle");
}

template <class T>
inline const T* Handle<T>::operator->() const
{
    if (ptr) return ptr;
    throw std::runtime_error
        ("access through unbound Handle");
}

#endif

```

该Handle类提供类似于指针的行为：复制Handle对象将不会复制基础对象；复制之后，两个Handle对象将引用同一基础对象。

要创建Handle对象，用户需要动态分配由Handle管理的类型（或从该类型派生的类型）的对象，并将该对象的地址传递给Handle类的构造函数。从此刻起，Handle将“拥有”这个对象，而且，一旦不再有任何Handle对象与该对象关联，Handle类将负责删除该对象。

### 习题16.46

解释复制Handle类型的对象时会发生什么。

#### 【解答】

如果是使用复制构造函数进行复制（包括：根据已存在的Handle对象创建新的Handle对象，用Handle对象作函数参数，函数返回Handle对象等），则复制Handle对象时将复制两个指针：指向基础对象的指针和指向使用计数的指针，并将使用计数加1；如果是使用赋值操作符进行复制，则首先将右操作数的使用计数加1，然后将左操作数的使用计数减1（如果使用计数减至0，则删除相应基础对象），再将右操作数的两个指针复制给左操作数。

复制之后，两个Handle对象将引用同一基础对象。

**习题16.47**

Handle类对用来实例化实际Handle类的类型有限制吗？如果有，有哪些限制？

**【解答】**

Handle类对用来实例化实际Handle类的类型有限制：用来实例化实际Handle类的类型不能为没有后代具体类的抽象类（即要能够创建该类（或其后代类）的对象），否则，因为不能创建基础对象，将无法创建有实用意义的Handle对象，也就是说，无法创建绑定到具体基础对象的Handle对象，只能创建未绑定的Handle对象。

**习题16.48**

解释如果用户将Handle对象与局部对象关联会发生什么。解释如果用户删除Handle对象所关联的对象会发生什么。

**【解答】**

如果用户将Handle对象与局部对象关联，那么，当该局部对象因生命期结束被系统自动撤销后，绑定到该局部对象的Handle对象中的指针ptr将不再指向有效基础对象。当绑定到该局部对象的最后一个Handle对象撤销时将执行`delete ptr`语句，从而因`delete`一个无效指针而出现运行时错误。

如果用户删除Handle对象所关联的对象（即基础对象），则该Handle对象中的ptr指针也将成为无效指针。当绑定到该基础对象的最后一个Handle对象撤销时，也同样会因`delete`一个无效指针而出现运行时错误。

**习题16.49**

实现本节提出的Sales\_item句柄的版本，该版本使用泛型Handle类管理Item\_base指针。

**【解答】**

```
// Sales_item.hpp(for 16-49)
// 定义Sales_item句柄类
#ifndef SALESITEM_H
#define SALESITEM_H

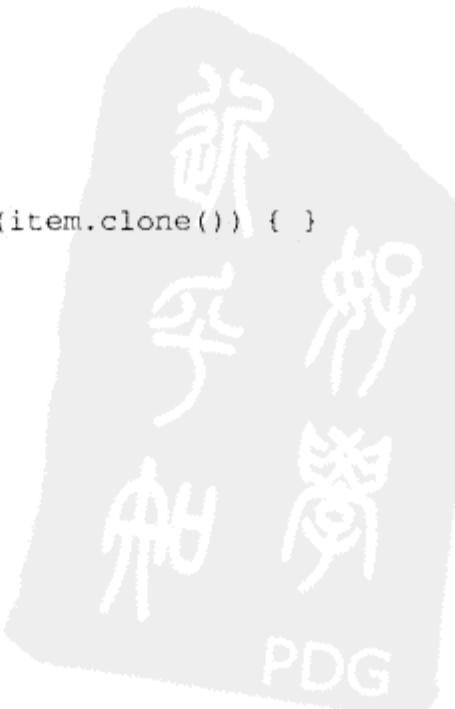
#include "Handle.hpp" // Handle.hpp定义泛型句柄类，见习题16.45解答
#include "Item.hpp"   // Item.hpp定义Item_base类层次，见习题15.35解答

// 用于Item_base层次的使用计数式句柄类
class Sales_item {
public:
    // 默认构造函数：创建未绑定的句柄
    Sales_item(): h() { }

    // 将创建绑定到Item_base对象副本的句柄
    Sales_item(const Item_base &item) : h(item.clone()) { }

    // 成员访问操作符
    const Item_base* operator->() const
    {
        return h.operator ->();
    }

    const Item_base& operator*() const
    {
        return *h;
    }
};
```



```

    }

private:
    Handle<Item_base> h; // 使用计数式句柄
};

#endif

```

**习题16.50**

重新运行函数计算销售总额。列出让你的代码工作必须进行的所有修改。

**【解答】**

与习题 15.35 给出的程序比较，Item\_base 类层次、Basket 类及主程序的代码无需改变，Sales\_item 类定义改变为习题 16.49 解答中所给出的。具体而言，对 Sales\_item.hpp 文件进行如下修改：

- 加入 #include "Handle.hpp" 指示（以引入习题 16.45 中定义的泛型句柄类 Handle）；
- 去掉 Sales\_item 类中的指针成员 p 和 use，代之以一个 Handle<Query\_base> 对象 h 作为数据成员，因为 Handle 类可以自行处理指针操作及相关的使用计数，所以去掉 Sales\_item 类中的复制控制成员以及成员函数 decr\_use（因此，不再需要对应的 Sales\_item 类的实现文件 Sales\_item.cpp），同时需修改 Sales\_item 类中其余成员函数的定义，以便适应数据成员的改变。

**习题16.51**

重新编写 15.9.4 节的 Query 类以使用泛型 Handle 类。注意你需要将 Handle 类设为 Query\_base 类的友元，以使它能够访问 Query\_base 构造函数。列出并解释让程序工作要做的其他所有修改。

**【解答】**

```

// 使用泛型Handle类实现的Query类
class Query {
    // 这些操作符需要访问Query_base*构造函数
    friend Query operator~(const Query &);
    friend Query operator|(const Query&, const Query&);
    friend Query operator&(const Query&, const Query&);

public:
    Query(const string&); // 建立新的WordQuery对象

    // 接口函数：将调用相应的Query_base操作
    set<TextQuery::line_no> eval(const TextQuery &t) const
    {
        return h -> eval(t);
    }

    ostream &display(ostream &os) const
    {
        return h -> display(os);
    }

private:
    Query(Query_base *query): h(query) { }
    Handle<Query_base> h;
};

```



```

inline
Query::Query(const string &s): h(new WordQuery(s)) {}

inline Query operator&(const Query &lhs, const Query &rhs)
{
    return new AndQuery(lhs, rhs);
}

inline Query operator|(const Query &lhs, const Query &rhs)
{
    return new OrQuery(lhs, rhs);
}

inline Query operator~(const Query &oper)
{
    return new NotQuery(oper);
}

```

除了将类Handle<Query\_base>设为Query\_base类的友元之外，还需进行以下修改：

- 在Query.hpp中加入#include "Handle.hpp";
- 去掉Query类中的指针成员p和use，代之以一个Handle<Query\_base>对象h作为数据成员。因为Handle类可以自行处理指针操作及相关的使用计数，所以去掉Query类中的复制控制成员以及成员函数decr\_use，同时需修改Query类的构造函数，以便适应数据成员的修改；
- eval函数和display函数中改用Handle对象h调用eval和display函数，以便适应数据成员的修改（此修改不是必须的：如果将Handle<Query\_base>对象命名为q，则无需进行此修改）。

### 习题16.52

定义函数模板count计算一个vector中某些值的出现次数。

#### 【解答】

下面定义的函数模板count计算sought中的元素在vec中的出现次数：

```

// count.hpp(for 16-52)
// 定义函数模板count计算一个vector中某些值的出现次数
#ifndef COUNT_H
#define COUNT_H

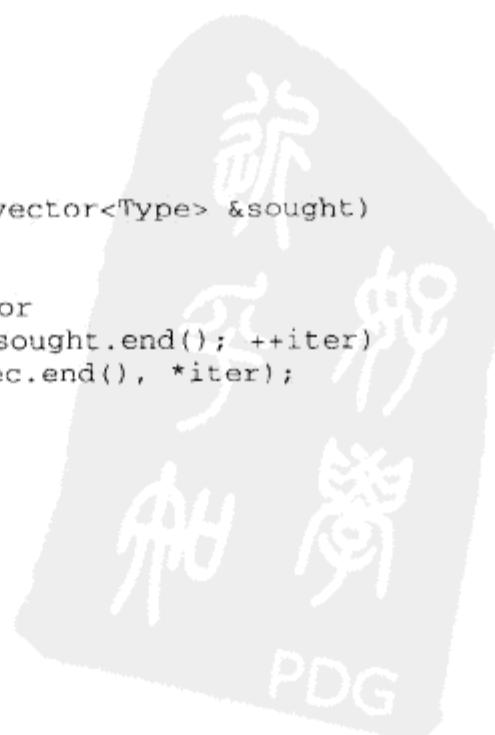
#include <vector>
#include <algorithm>
using std::vector;
using std::size_t;

template <typename Type>
size_t count(const vector<Type> &vec, const vector<Type> &sought)
{
    size_t result = 0;
    for (typename vector<Type>::const_iterator
        iter = sought.begin(); iter != sought.end(); ++iter)
        result += std::count(vec.begin(), vec.end(), *iter);

    return result;
}

#endif

```



注意，利用标准库中提供的泛型算法count计算指定值在vector中的出现次数。sought也是一个vector，用于存放要查找的那些值。

### 习题16.53

编写一个程序调用上题中定义的count函数，首先传给该函数一个double型vector，然后传递一个int型vector，最后传递一个char型vector。

#### 【解答】

```
// 16-53.cpp
// 分别用double型、int型和char型的vector
// 调用习题16.52中定义的count函数
#include "count.hpp" // 引入上题中定义的count函数
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    double da[] = {2.1, 2.2, 2.3, 2.4, 2.3, 2.8};
    double dsa[] = {2.3, 2.4};
    int ia[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 2};
    int isa[] = {2, 4, 10, 9};
    char ca[] = {'a', 'b', 'o', 'a', 'r', 'd'};
    char csa[] = {'a', 'b'};
    vector<double> dvec(da, da+6);
    vector<double> dsought(dsa, dsa+2);
    vector<int> ivec(ia, ia+10);
    vector<int> isought(isa, isa+4);
    vector<char> cvec(ca, ca+6);
    vector<char> csought(csa, csa+2);

    // 传递double型vector调用count函数
    cout << count(dvec, dsought) << endl;

    // 传递int型vector调用count函数
    cout << count(ivec, isought) << endl;

    // 传递char型vector调用count函数
    cout << count(cvec, csought) << endl;

    return 0;
}
```

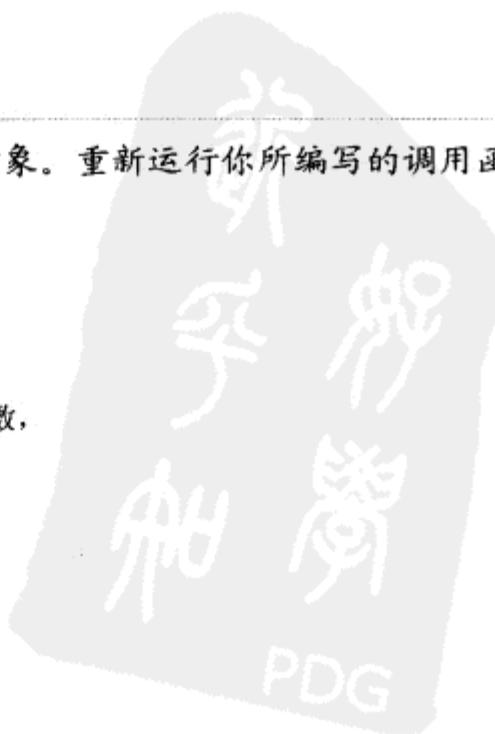
### 习题16.54

引入count函数的一个特化模板实例以处理string对象。重新运行你所编写的调用函数模板实例化的程序。

#### 【解答】

将习题 16.52 中给出的函数的 count 定义修改为：

```
// count.hpp(for 16-54)
// 定义函数模板count计算一个vector中某些值的出现次数,
// 引入count函数的一个特化模板实例以处理string对象
#ifndef COUNT_H
#define COUNT_H
#include <vector>
```



```

#include <string>
#include <cstddef>
#include <algorithm>
using std::vector;
using std::string;
using std::size_t;

template <typename Type>
size_t count(const vector<Type> &vec, const vector<Type> &sought)
{
    size_t result = 0;
    for (typename vector<Type>::const_iterator iter = sought.begin();
         iter != sought.end(); ++iter)
        result += std::count(vec.begin(), vec.end(), *iter);

    return result;
}

// 模板特化
template <>
size_t count<string>(const vector<string> &vec, const vector<string> &sought)
{
    size_t result = 0;
    for (vector<string>::const_iterator iter = sought.begin();
         iter != sought.end(); ++iter)
        result += std::count(vec.begin(), vec.end(), *iter);

    return result;
}

#endif

```

则可以修改习题 16.53 的解答中给出的主程序，以使用 string 型的 vector 调用 count 函数：

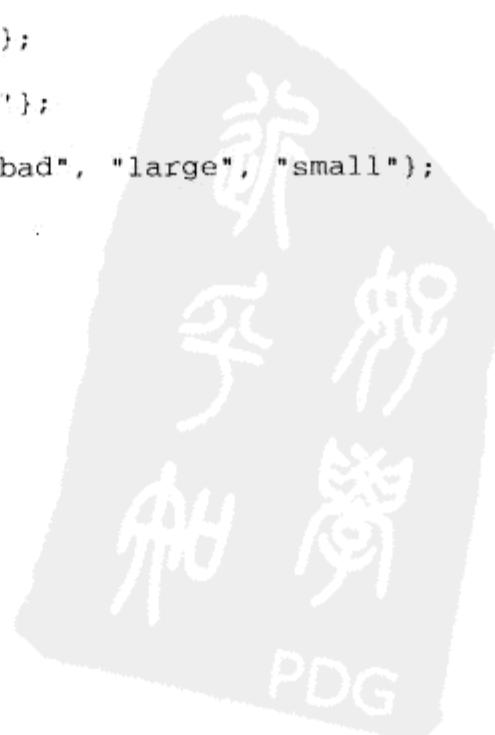
```

// 16-54.cpp
// 分别用double型、int型、char型和string型的vector调用count函数
#include "count.hpp"
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
    double da[] = {2.1, 2.2, 2.3, 2.4, 2.3, 2.8};
    double dsa[] = {2.3, 2.4};
    int ia[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 2};
    int isa[] = {2, 4, 10, 9};
    char ca[] = {'a', 'b', 'o', 'a', 'r', 'd'};
    char csa[] = {'a', 'b'};
    string sa[] = {"many", "much", "good", "bad", "large", "small"};
    string ssa[] = {"many", "large", "what"};
    vector<double> dvec(da, da+6);
    vector<double> dsought(dsa, dsa+2);
    vector<int> ivec(ia, ia+10);
    vector<int> isought(isa, isa+4);
    vector<char> cvec(ca, ca+6);
    vector<char> csought(csa, csa+2);
    vector<string> svec(sa, sa+6);
    vector<string> ssought(ssa, ssa+3);

    // 传递double型vector调用count函数
    cout << count(dvec, dsought) << endl;
}

```



```

    // 传递int型vector调用count函数
    cout << count(ivec, isought) << endl;

    // 传递char型vector调用count函数
    cout << count(cvec, csought) << endl;

    // 传递string型vector调用count函数
    cout << count(svec, ssought) << endl;

    return 0;
}

```

注意，对于这个程序，如果没有提供count函数的特化版本，则传递string型vector调用count函数时，所调用的是将模板形参推断为string类型的实例化版本；如果提供了count函数的处理string对象的特化版本，则传递string型vector调用count函数时，所调用的是该特化版本，这一事实对模板用户而言是透明的。如果在特化版本中提供一个显式的打印语句，如：std::cout << "template spacialization" << std::endl;，则运行上述程序可以清楚地看到这一事实。

### 习题16.55

Queue针对const char\*的特化版本中的注释指出，不必定义默认构造函数或复制控制成员，解释为什么对于Queue的这个版本合成成员就足够了。

#### 【解答】

因为Queue针对const char\*的特化版本中只有一个数据成员，该数据成员的类型为类类型。这个类类型在被复制、被赋值以及被撤销时可以完成正确的工作。

### 习题16.56

我们已经解释过未针对const char\*特化的Queue的泛型行为，使用泛型Queue模板解释下面代码中会发生什么：

```

Queue<const char*> q1;
q1.push("hi"); q1.push("bye"); q1.push("world");
Queue<const char*> q2(q1); // q2是q1的副本
Queue<const char*> q3; // 空的Queue
q1 = q3;

```

具体而言，就是说明在q2的初始化和q1的赋值之后，q1和q2是什么值。

#### 【解答】

使用q1进行初始化之后，q2成为q1的副本，即q2的链表中具有三个QueueItem节点。各个节点的item部分分别保存C风格字符串字面值"hi"、"bye"和"world"的地址；使用q3进行赋值之后，q1成为q3的副本，即成为一个空Queue对象（其链表中没有QueueItem节点）。

### 习题16.57

我们的Queue特化版本从front函数返回string对象而不是const char\*，你认为为什么这样做？你能够怎样实现Queue以返回const char\*？讨论每种方法的优缺点。

#### 【解答】

Queue特化版本从front函数返回string对象而不是const char\*，目的是为了避免管理字符数组。

将函数的定义修改为：

```
const char* front()
{
    return real_queue.front().c_str();
}
```

即可返回const char\*。

front函数返回string对象的优点是：无需管理字符数组，不会出现内存问题。缺点是：返回的Queue元素与特化版本所指定类型const char\*不一致，可能会让用户有点困惑。

front函数返回const char\*的优点是：返回的Queue元素与特化版本所指定的类型const char\*一致。缺点是：当Queue中对应节点被删除（pop）掉之后，用户仍可能使用该指针，从而导致内存问题，例如，将front函数的返回值赋值给const char\*型指针变量p，然后执行Queue对象的pop操作，此后指针p就指向一块不再有效的内存了，对p的使用会导致错误；而且，用户代码对指针p的使用有可能破坏Queue元素，例如，将front函数的返回值经过强制类型转换后赋值给char \*类型的指针q，则可以通过q改变对应内存的内容，从而破坏Queue元素。

注意，也可以通过只特化Queue类中的push和pop成员而实现front函数返回const char\*。

### 习题16.58

前一小节中给出的Queue类的特化，以及本小节中push和pop函数的特化，只适用于const char\*类型的Queue。为普通char\*实现特化Queue的这两种不同方式。

#### 【解答】

针对char\*的Queue类特化与针对const char\*的Queue类特化类似，只需将所有const char\*改为char\*即可，代码如下：

```
// 针对char*的特化
template<> class Queue<char*> {
public:
    void push(char*);
    void pop()
    {
        real_queue.pop();
    }

    bool empty() const
    {
        return real_queue.empty();
    }

    // 注意，返回类型与模板形参类型不匹配
    std::string front()
    {
        return real_queue.front();
    }

    const std::string &front() const
    {
        return real_queue.front();
    }

private:
```



```

    Queue<std::string> real_queue;
};

void Queue<char*>::push(char* val)
{
    return real_queue.push(val);
}

```

针对char\*的push和pop函数的特化与针对const char\*的push和pop函数的特化类似，只需将所有const char\*改为char\*即可，代码如下：

```

template <>
void Queue<char*>::push(char *const &val)
{
    // 分配新的字符数组并从val复制字符
    char* new_item = new char[strlen(val) + 1];
    strncpy(new_item, val, strlen(val) + 1);

    // 新分配一个元素并进行初始化
    QueueItem<char*> *pt =
        new QueueItem<char*>(new_item);

    // 在队列中增加元素
    if (empty())
        head = tail = pt; // 队列中只有一个元素
    else {
        tail->next = pt; // 将新元素加到队尾
        tail = pt;
    }
}

template <>
void Queue<char*>::pop()
{
    // 保存头指针以进行元素删除
    QueueItem<char*> *p = head;
    delete head->item; // 删除push操作中分配的字符数组
    head = head->next; // 修改头指针
    delete p; // 删除原来的队头元素
}

```

### 习题16.59

如果走只特化push函数的路线，对于C风格字符串的Queue，front返回什么值？

#### 【解答】

front返回类型为const char\*的指针。

### 习题16.60

讨论这两个设计的优缺点：为const char\*定义该类的特化版本和只特化push和pop函数。具体而言，比较front的行为的异同以及用户代码中的错误破坏Queue元素的可能性。

#### 【解答】

在为const char\*定义该类的特化版本的设计中：front函数返回string对象，该string对象是Queue中QueueItem节点中item部分所包含的string对象的副本（复制品）。用户代码对该string对象的任意使用都不可能破坏Queue元素，这是其优点。缺点是定义一个特化类所涉及的代码修改量较大。

只特化push和pop函数的设计中：front函数返回类型为const char\*的指针，该指针是queue中QueueItem节点中item部分所包含的指针的副本（复制品）。两个指针指向同一块内存，用户代码对该指针的使用有可能破坏Queue元素，例如，将front函数的返回值经过强制类型转换后赋值给char\*类型的指针q，则可以通过q改变对应内存的内容，从而破坏Queue元素，这是其缺点。优点是只需定义push和pop函数的特化版本，所涉及的代码修改量较小。

### 习题16.61

实现compare函数的三个版本。在每个函数中包含一个输出语句，指出正在调用哪个函数。使用这些函数检查对其余问题的回答。

#### 【解答】

```
// 比较两个对象
template <typename T> int compare(const T& v1, const T& v2)
{
    cout << "compares two objects" << endl;
    if (v1 < v2)
        return -1;
    if (v2 < v1)
        return 1;
    return 0;
}

// 比较两个序列中的元素
template <class U, class V> int compare(U v1, U v2, V beg)
{
    cout << "compares elements in two sequences" << endl;
    return 0;
}

// 处理C风格字符串的普通函数
int compare(const char* p1, const char* p2)
{
    cout << "plain function to handle C-style character strings"
        << endl;
    return strcmp(p1, p2);
}
```

### 习题16.62

对于本节定义的compare函数和变量，解释下面每个函数调用中，哪个函数被调用以及为什么。

```
compare(ch_arr1, const_arr1);
compare(ch_arr2, const_arr2);
compare(0, 0);
```

#### 【解答】

compare(ch\_arr1, const\_arr1); 和 compare(ch\_arr2, const\_arr2); 调用比较C风格字符串的普通函数。对于这两个函数调用，候选函数只有比较C风格字符串的普通函数，因为比较简单对象的模板函数只能实例化为两个形参类型相同的函数，而比较两个序列元素的模板函数带有三个形参。

compare(0, 0); 调用比较简单对象的模板函数的实例（将T绑定到int）。对于这个函数调用，将T绑定到int的比较简单对象的模板函数实例化以及比较C风格字符串的普通函数均可作为候选函数，但前者是完全匹配，而后者需要从int到const char\*的转换，所以优先选用前者。

**习题16.63**

对于下面的每个调用，列出候选函数和可行函数，指出调用是否有效，以及如果有效，调用哪个函数。

```
template <class T> T calc(T, T);
double calc(double, double);
template <> char calc<char>(char, char);
int ival; double dval; float fd;
calc(0, ival); calc(0.25, dval);
calc(0, fd); calc(0, 'J');
```

**【解答】**

对于`calc(0, ival);`，候选函数为：将T绑定到int的函数模板实例和`double calc(double, double);`。二者均为可行函数，但前者是完全匹配，后者需进行从int到double的转换，所以，优先选择前者。

对于`calc(0.25, dval);`，候选函数为：将T绑定到double的函数模板实例和`double calc(double, double);`。二者均为可行函数，且均为完全匹配，从可行函数中去掉模板实例，只剩下后者，所以该调用有效，调用`double calc(double, double);`。

对于`calc(0, fd);`，候选函数为：`double calc(double, double);`。因为两个实参一个为int，一个为float，均可转换为double，所以该函数也是唯一的可行函数，调用有效。

对于`calc(0, 'J');`，候选函数为：`double calc(double, double);`。因为两个实参一个为int，一个为char，均可转换为double，所以该函数也是唯一的可行函数，调用有效。



# 第 17 章

## 用于大型程序的工具

### 习题17.1

下面的throw语句中，异常对象的类型是什么？

- (a) range\_error r("error");    (b) exception \*p = &r;  
    throw r;                            throw \*p;

#### 【解答】

(a) 异常对象 r 的类型是 range\_error。

(b) 被抛出的异常对象是对指针 p 解引用的结果，其类型与 p 的静态类型相匹配，为 exception。

### 习题17.2

如果第二个throw语句写成throw p，会发生什么情况？

#### 【解答】

如果r是一个局部对象，则throw p抛出的p是指向局部对象的指针，那么，在执行对应异常处理代码时，有可能对象r已不再存在，从而导致程序不能正确运行。所以，通常throw语句不应该抛出指针，尤其是不应该抛出指向局部对象的指针。

### 习题17.3

解释下面这个try块为什么不正确，并改正它。

```
try {  
    // use of the C++ standard library  
} catch(exception) {  
    // ...  
} catch(const runtime_error &re) {  
    // ...  
} catch(overflow_error eobj) { /* ... */ }
```

#### 【解答】

该try块中使用的exception、runtime\_error及overflow\_error是标准库中定义的异常类。它们是因继承而相关的：runtime\_error类继承exception类，overflow\_error类继承runtime\_error类。在使用来自继承层次的异常时，catch子句应该从最低派生类型到最高派生类型排序，以便派生类型的处理代码出现在其基类类型的catch之前，所以，上述块中catch子句的顺序错误。

可更正为：

```
try {
```

```

    // use of the C++ standard library
} catch(overflow_error eobj) {
    /* ... */
} catch(const runtime_error &re) {
    // ...
} catch(exception) {
    // ...
}

```

**习题17.4**

对于如下的基本C++程序

```

int main() {
    // use of the C++ standard library
}

```

修改main函数以捕获由C++标准库中函数抛出的任何异常。处理代码应该在调用abort函数（在头文件cstdlib中定义）终止main函数之前显示与异常相关的错误信息。

**【解答】**

```

int main()
{
    try {
        // use of the C++ standard library
    }
    catch(const exception &e) {
        cerr << e.what() << endl;
        abort();
    }
    return 0;
}

```

**习题17.5**

对于下面的异常类型以及catch子句，编写一个throw表达式，该表达式创建一个可被每个catch子句捕获的异常对象。

- (a) class exceptionType { };  
catch(exceptionType \*pet) { }
- (b) catch(...) { }
- (c) enum mathErr { overflow, underflow, zeroDivide };  
catch(mathErr &ref) { }
- (d) typedef int EXCPTYPE;  
catch(EXCPTYPE) { }

**【解答】**

- (a) throw new exceptionType(); // 动态创建一个异常对象并抛出其指针
- (b) throw 8; // 被抛出的表达式可为任意类型
- (c) throw overflow; // 被抛出的表达式为mathErr类型即可
- (d) throw 10; // 被抛出的表达式为int类型即可

**习题17.6**

给定下面的函数，解释当发生异常时会发生什么。

```
void exercise(int *b, int *e)
```

```

{
    vector<int> v(b, e);
    int *p = new int[v.size()];
    ifstream in("ints");
    // exception occurs here
    // ...
}

```

**【解答】**

在new操作之后发生的异常使得动态分配的数组没有被撤销。

**习题17.7**

有两种方法可以使上面的代码是异常安全的，描述并实现它们。

**【解答】**

一种方法是将有可能发生异常的代码放在try块中，以便在异常发生时捕获异常：

```

void exercise(int *b, int *e)
{
    vector<int> v(b, e);
    int *p = new int[v.size()];
    try {
        ifstream in("ints");
        // 此处发生异常
    }
    catch {
        delete p; // 释放数组
        //...进行其他处理
    }
    // ...
}

```

另一种方法是定义一个类来封装数组的分配和释放，以保证正确释放资源：

```

class Resource {
public:
    Resource(size_t sz) : r(new int[sz]) { }
    ~Resource() { if (r) delete r; }
    //...其他操作
private:
    int *r;
};

```

函数exercise相应修改为：

```

void exercise(int *b, int *e)
{
    vector<int> v(b, e);
    Resource res(v.size()); // 使用Resource对象
    ifstream in("ints");
    // exception occurs here
    // ...
}

```

注意，此处给出的Resource类非常简略，要达到实用，还需定义其他操作，包括复制构造函数、赋值操作、解引用操作、箭头操作、下标操作等，以支持内置指针及数组的使用方式并保证自动删除Resource对象所引用的数组。另外，可将该Resource类定义为类模板，以支持多种数组元素类型。

**习题17.8**

下面的auto\_ptr声明中，哪些是不合法的或者可能导致随后的程序错误？解释每个声明的问题。

```
int ix = 1024, *pi = &ix, *pi2 = new int(2048);
typedef auto_ptr<int> IntP;
```

- (a) IntP p0(ix);
- (b) IntP p1(pi);
- (c) IntP p2(pi2);
- (d) IntP p3(&ix);
- (e) IntP p4(new int(2048));
- (f) IntP p5(p2.get());

**【解答】**

- (a) 不合法。必须向auto\_ptr的构造函数传递一个由new操作返回的指针，ix不是指针。
- (b) 可能导致随后的程序错误：auto\_ptr只能用于管理从new操作返回的一个对象，pi是指向ix的指针，而ix是静态分配的对象。
- (c) 正确。
- (d) 可能导致随后的程序错误：auto\_ptr只能用于管理从new操作返回的一个对象，此处传给auto\_ptr构造函数的是静态分配的对象ix的地址。
- (e) 正确。
- (f) 可能导致随后的程序错误：因为两个auto\_ptr对象p2和p5拥有同一基础对象（保存相同的指针），会导致同一指针被delete两次。

**习题17.9**

假定ps是一个指向string的auto\_ptr对象<sup>1</sup>，如果有的话，下面两个assign（9.6.2节）调用有什么不同？你认为哪个更好，为什么？

- (a) ps.get()->assign("Danny");
- (b) ps->assign("Danny");

**【解答】**

两个assign调用的效果没什么不同。因为auto\_ptr对象的get操作和->操作都返回该auto\_ptr对象保存的指针，因此二者都是通过指向string对象的指针调用assign函数。

但(b)的形式更简单直接，与内置指针的使用形式一样，因此更好。

**习题17.10**

如果函数有形如throw()的异常说明，它能抛出什么异常？如果没有异常说明呢？

**【解答】**

如果函数有形如throw()的异常说明，则该函数不抛出任何异常。

如果函数没有异常说明，则该函数可以抛出任意类型的异常。

**习题17.11**

如果有，下面哪个初始化是错误的？为什么？

<sup>1</sup> 此处英文原文有误。

```
void example() throw(string);
(a) void (*pf1)() = example;
(b) void (*pf2)() throw() = example;
```

**【解答】**

(b)是错误的。

因为用另一指针初始化带异常说明的函数指针时，源指针的异常说明必须至少与目标指针一样严格。函数指针pf2的声明指出，pf2指向不抛出任何异常的函数，而example函数的声明指出它能抛出string类型的异常，example函数抛出的异常类型超过了pf2所指定的，所以，对pf2而言，example函数不是有效的初始化式，会引发编译时错误。

**习题17.12**

下面的函数可以抛出哪些异常？

- (a) void operate() throw(logic\_error);
- (b) int op(int) throw(underflow\_error, overflow\_error);
- (c) char manip(string) throw();
- (d) void process();

**【解答】**

异常说明指定，如果函数抛出异常，被抛出的异常将是包含在该说明中的一种，或者是从列出的异常类型中派生的类型。因此：

- (a) operate函数可以抛出logic\_error、domain\_error、invalid\_argument、out\_of\_range和length\_error类型的异常。
- (b) op函数可以抛出underflow\_error和overflow\_error类型的异常。
- (c) manip函数不抛出任何异常。
- (d) process函数可以抛出任意类型的异常。

**习题17.13**

将17.1.7节描述的书店异常类定义为名为Bookstore的命名空间的成员。

**【解答】**

```
namespace Bookstore {
    class out_of_stock: public std::runtime_error {
        public:
            explicit out_of_stock(const std::string &s):
                std::runtime_error(s) { }
    };
    class isbn_mismatch: public std::logic_error {
        public:
            explicit isbn_mismatch(const std::string &s):
                std::logic_error(s) { }
            isbn_mismatch(const std::string &s,
                const std::string &lhs, const std::string &rhs):
                std::logic_error(s), left(lhs), right(rhs) { }
            const std::string left, right;
            virtual ~isbn_mismatch() throw() { }
    };
}
```

**习题17.14**

在命名空间Bookstore内部定义Sales\_item及其操作符。定义加操作符抛出一个异常。

**【解答】**

```

namespace Bookstore {
    class Sales_item {
    public:
        Sales_item(const std::string &book = "") :
            isbn(book), units_sold(0), revenue(0.0) { }
        Sales_item(std::istream &is) { is >> *this; }

        bool same_isbn(const Sales_item &rhs) const
        { return isbn == rhs.isbn; }

        std::string book() const
        { return isbn; }

        double avg_price() const;

        friend std::istream& operator>>(std::istream& in, Sales_item& s);

        friend std::ostream& operator<<(std::ostream& out, const Sales_item& s);

        Sales_item& operator=(const std::string& str)
        {
            isbn = str;
            return *this;
        }

        Sales_item& operator += (const Sales_item& rhs)
        {
            units_sold += rhs.units_sold;
            revenue += rhs.revenue;
            return *this;
        }

    private:
        std::string isbn;
        unsigned units_sold;
        double revenue;
    };
}

double Sales_item::avg_price() const
{
    if (units_sold)
        return revenue / units_sold;
    else
        return 0;
}

std::istream& operator>>(std::istream& in, Sales_item& s)
{
    double price;
    in >> s.isbn >> s.units_sold >> price;

    // 检查输入是否有效
    if (in)
        s.revenue = s.units_sold * price;
    else
        s = Sales_item(); // 输入失败：将对象置为默认状态
}

```

```

        return in;
    }

std::ostream& operator<<(std::ostream& out, const Sales_item& s)
{
    out << s.isbn << "\t" << s.units_sold << "\t"
        << s.revenue ;
    return out;
}

Sales_item operator + (const Sales_item& lhs, const Sales_item& rhs)
{
    if (!lhs.same_isbn(rhs))
        throw isbn_mismatch("isbn mismatch", lhs.book(),
                             rhs.book());
    Sales_item ret(lhs);
    ret += rhs;
    return ret;
}
}

```

注意，此处为了节省篇幅，将命名空间成员的声明和定义放在一起，更规范的做法应该是将接口和实现分离，分别放在相关的头文件和实现文件中（见17.2.1节）。

### 习题17.15

编写一个程序，使用Sales\_item加操作符并处理任何异常。使这个程序成为名为MyApp的另一命名空间的成员。这个程序应使用上题中在命名空间Bookstore中定义的异常类。

#### 【解答】

可编写一个函数processTrans，使用Sales\_item加操作符并处理任何异常，将该函数定义为名为MyApp的另一命名空间的成员。

程序如下：

```

namespace MyApp {
    void processTrans()
    {
        Bookstore::Sales_item item1, item2, sum;
        while (std::cin >> item1 >> item2) { // 读入两个交易
            try {
                sum = item1 + item2; // 计算它们的和sum
                // ...使用sum
            }
            catch (const Bookstore::isbn_mismatch &e) {
                std::cerr << e.what() << ": left isbn(" << e.left
                    << ") right isbn(" << e.right << ")"
                    << std::endl;
            }
        }
    }
}

```

### 习题17.16

将为回答每章中的问题而编写的程序组织到每一章自己的命名空间中，也就是说，命名空间

`chapterrefinheritance`将包含Query程序的代码，而`chapterrefalgs`将包含TextQuery代码。使用这个结构，编译Query代码示例。

### 【解答】

将Query类以及Query\_base类层次定义为命名空间`chapterrefinheritance`的成员，将TextQuery类定义为命名空间`chapterrefalgs`的成员，并相应修改主函数中的代码（使用限定名引用这些类，或者使用相关的using声明）。

代码略。

### 习题17.17

在本书中，我们定义了两个名为Sales\_item的不同类：在第一部分定义和使用的初始简单类，以及在15.8.1节定义的与Item\_base继承层次接口的句柄类。在命名空间`cplusplus_primer`内部定义两个嵌套命名空间，用于区别这两个类定义。

### 【解答】

可如下定义嵌套命名空间：

```
namespace cplusplus_primer {
    // 包含简单Sales_item类的嵌套命名空间
    namespace simpleSI {
        // 将本书第一部分中所定义的Sales_item类的定义放在此处
    }

    // 包含Item_base继承层次及句柄Sales_item类的嵌套命名空间
    namespace handleSI {
        // 将15.8.1节定义的Item_base继承层次
        // 及句柄Sales_item类的定义放在此处
    }
}
```

### 习题17.18

为什么在程序中定义自己的命名空间？何时可以使用未命名的命名空间？

### 【解答】

在一个给定作用域中定义的每个名字在该作用域中必须是唯一的。当开发大型复杂程序时，往往需要在全局作用域中定义许多名字，而且往往需要使用他人所提供的代码（如标准库或第三方供应商提供的库）来编写程序，这时很容易因库用户所使用的名字与库中定义的全局名字相同而发生名字冲突。在程序中定义自己的命名空间，正是为了避免这种名字冲突。

通常，当需要声明局部于文件的实体时，可以使用未命名的命名空间（在文件的最外层作用域中定义未命名的命名空间）。

### 习题17.19

假定有下面的operator\*的声明，operator\*是嵌套命名空间`cplusplus_primer:: MatrixLib`的成员：

```
namespace cplusplus_primer {
    namespace MatrixLib {
        class matrix { /* ... */ };
    }
}
```

```

        matrix operator*
        (const matrix &, const matrix &);

    // ...
}

}

```

怎样在全局作用域中定义这个操作符？只需给出操作符定义的原型。

### 【解答】

将函数返回类型及函数名加上命名空间名字限定即可：

```
cplusplus_primer::MatrixLib::matrix cplusplus_primer::MatrixLib::operator*
(const matrix &, const matrix &)
{ /*...*/ }
```

### 习题17.20

解释using声明和using指示之间的区别。

### 【解答】

区别在于：一个using声明只能引入特定命名空间中的一个成员，一个using指示使得特定命名空间中的所有名字都成为可见的。

### 习题17.21

考虑下面的代码样本：

```

namespace Exercise {
    int ivar = 0;
    double dvar = 0;
    const int limit = 1000;
}
int ivar = 0;
// position 1
void manip() {
    // position 2
    double dvar = 3.1416;
    int iobj = limit + 1;
    ++ivar;
    +++:ivar;
}

```

如果命名空间Exercise的所有成员的using声明放在标为position 1的地方，这个代码样本中的声明和表达式的效果是什么？如果放在position 2位置呢？用命名空间Exercise的using指示代替using声明，回答同一问题。

### 【解答】

如果命名空间Exercise的所有成员的using声明放在标为position 1的地方，则Exercise中的成员在全局作用域中可见。using Exercise::ivar;会导致ivar重复定义的编译错误，因为在全局作用域中也定义了一个同名变量（注意，由using声明引起的二义性错误在声明点检测）；而manip中的double dvar = 3.1416;声明了一个局部的变量dvar，在函数体作用域中它将屏蔽Exercise::dvar; int iobj = limit + 1;声明了一个局部变量iobj，并用Exercise::limit加1的结果对其进行初始化。

如果命名空间Exercise的所有成员的using声明放在标为position 2的地方，则manip中的double

`dvar = 3.1416;` 属于对变量`dvar`的重复定义，会出现编译错误；`int iobj = limit + 1;` 声明了一个局部变量`iobj`，并用`Exercise::limit`加1的结果对其进行初始化；`++ivar;` 访问到的是`Exercise::ivar`，而`++::ivar;` 访问的是全局变量`ivar`。

如果命名空间`Exercise`的`using`指示放在`position 1`处，则`manip`中的`double dvar = 3.1416;` 声明了一个局部的变量`dvar`，在函数体作用域中它将屏蔽`Exercise::dvar`；`int iobj = limit + 1;` 声明了一个局部变量`iobj`，并用`Exercise::limit`加1的结果对其进行初始化；`++ivar;` 访问到的是`Exercise::ivar`，而`++::ivar;` 访问的是全局变量`ivar`。

如果命名空间`Exercise`的`using`指示放在`position 2`处，则`Exercise`的成员看来好像是声明在全局作用域中的一样，`manip`中的`double dvar = 3.1416;` 声明了一个局部的变量`dvar`，在函数体作用域中它将屏蔽`Exercise::dvar`；`int iobj = limit + 1;` 声明了一个局部变量`iobj`，并用`Exercise::limit`加1的结果对其进行初始化；`++ivar;` 出现二义性错误，因为编译器无法分辨是访问`Exercise::ivar`，还是访问全局变量`ivar`；而`++::ivar;` 访问的是全局变量`ivar`。

### 习题17.22

给定下面的代码，如果有，确定哪个函数与`compute`函数的调用匹配，列出候选函数与可行函数。如果有，对实参应用什么类型转换序列，以匹配每个可行函数的形参？

```
namespace primerLib {
    void compute();
    void compute(const void *);
}
using primerLib::compute;
void compute(int);
void compute(double, double = 3.4);
void compute(char*, char* = 0);
int main()
{
    compute(0);
    return 0;
}
```

如果将`using`声明放在`main`中的`compute`调用之前，会发生什么情况？回答与前面相同的问题。

#### 【解答】

全局作用域中声明的函数`void compute(int)`与`compute`函数的调用匹配。

候选函数：命名空间`primerLib`中声明的两个`compute`函数（因`using`声明使得它们在全局作用域中可见），以及全局作用域中声明的三个`compute`函数。

可行函数：因函数调用中给出的实参`0`为`int`类型，所以可行函数为以下4个函数。

- `void compute(int)。`
- `void compute(double, double = 3.4)。`
- `void compute(char*, char* = 0)。`
- `primerLib` 中声明的 `void compute(const void*)`。

其中，第一个为完全匹配，第二个需要将实参隐式转换为`double`类型，第三个需要将实参隐式转换为`char*`类型，第四个需要将实参隐式转换为`void*`类型方可匹配，所以第一个为最佳匹配。

如果将`using`声明放在`main`中的`compute`调用之前，则`primerLib`中声明的`void compute(const`

`void*`)与`compute`函数的调用匹配。

候选函数：命名空间`primerLib`中声明的两个`compute`函数（因`using`声明使得它们在`main`函数的函数体作用域中可见）。

可行函数：因函数调用中给出的实参`0`为`int`类型，所以可行函数为`primerLib`中声明的`void compute(const void*)`。需要将实参隐式转换为`void*`类型方可匹配。

### 习题17.23

如果有，下面哪些声明是错误的。解释为什么。

- (a) `class CADVehicle : public CAD, Vehicle { ... };`
- (b) `class DoublyLinkedList:`  
    `public List, public List { ... };`
- (c) `class iostream: public istream, public ostream { ... };`

### 【解答】

(b) 错误。在一个派生列表中，同一基类只能出现一次，这里`List`出现了两次。

### 习题17.24

给定下面的类层次，其中，每个类定义了一个默认构造函数：

```
class A { ... };
class B : public A { ... };
class C : public B { ... };
class X { ... };
class Y { ... };
class Z : public X, public Y { ... };
class MI : public C, public Z { ... };
```

对于下面的定义，构造函数的执行次序是什么？

```
MI mi;
```

### 【解答】

构造函数的执行次序：

- (1) A的构造函数；
- (2) B的构造函数；
- (3) C的构造函数；
- (4) X的构造函数；
- (5) Y的构造函数；
- (6) Z的构造函数；
- (7) MI的构造函数。

### 习题17.25

给定下面的类层次，其中每个类定义了一个默认构造函数：

```
class X { ... };
class A { ... };
class B : public A { ... };
```

```
class C : private B { ... };
class D : public X, public C { ... };
```

如果有，下面转换中哪些是不允许的？

- ```
D *pd = new D;
```
- (a) X \*px = pd;    (b) A \*pa = pd;  
(c) B \*pb = pd;    (d) C \*pc = pd;

### 【解答】

(c)和(b)是不允许的。

因为C对B的继承是私有继承，使得在D中B的默认构造函数成为不可访问的（见15.2.5节），所以尽管存在从“D\*”到“B\*”以及从“D\*”到“A\*”的转换，但这些转换不可访问。

### 习题17.26

本节给出了一系列通过指向Panda对象的Bear指针所进行的调用。我们指出，如果指针是ZooAnimal指针，则将以同样方式确定函数调用。解释为什么。

### 【解答】

如果使用ZooAnimal指针，则只能使用ZooAnimal类中定义的操作。

pb->print(cout); 通过基类指针调用虚函数，使用动态绑定，pb目前指向Panda对象，所以调用Panda::print(ostream&).

pb->cuddle(); 因为ZooAnimal类中没有定义cuddle操作，所以该调用出错。

pb->highlight(); 因为ZooAnimal类中没有定义highlight操作，所以该调用出错。

delete pb; 因为ZooAnimal类中定义了虚析构函数，所以Panda类中的（合成的）析构函数也是虚函数（见15.4.4节），因此delete pb;通过虚机制调用Panda析构函数。随着Panda析构函数的执行，依次调用Endangered、Bear和ZooAnimal的析构函数。

所以说，通过指向Panda对象的Bear指针或ZooAnimal指针进行上述调用，将以同样方式确定函数调用。

### 习题17.27

假定有两个基类Base1和Base2，其中每一个定义了一个名为print的虚成员和一个虚析构函数。从这些基类派生下面的类，其中每一个类都重定义了print函数：

```
class D1 : public Base1 { /* ... */ };
class D2 : public Base2 { /* ... */ };
class MI : public D1, public D2 { /* ... */ };
```

使用下面的指针确定在每个调用中使用哪个函数：

```
Base1 *pb1 = new MI; Base2 *pb2 = new MI;
D1 *pd1 = new MI; D2 *pd2 = new MI;
```

- (a) pb1->print(); (b) pd1->print(); (c) pd2->print();
(d) delete pb2; (e) delete pd1; (f) delete pd2;

### 【解答】

(a)、(b)和(c)均通过基类指针调用虚函数print，这些基类指针当前都指向MI类对象，所以均调用

MI::print()。

(d)、(e)和(f)均通过基类指针删除对象，这些基类指针当前都指向MI类对象，所以均通过虚机制调用MI析构函数。随着MI析构函数的执行，依次调用D2、Base2、D1和Base1的析构函数。

### 习题17.28

编写与图17-2对应的类定义。

#### 【解答】

该类层次的概略定义如下：

```
class ZooAnimal {
    //...成员略
};

class Bear : public ZooAnimal {
    //...成员略
};

class Endangered {
    //...成员略
};

class Panda : public Bear, public Endangered {
    //...成员略
};
```

### 习题17.29

给定前面给出的类层次以及下面的MI::foo成员函数框架：

```
int ival;
double dval;
void MI::foo(double dval) { int id; /* ... */ }
```

- (a) 识别从MI中可见的成员名字。有从多个基类中都可见的名字吗？
- (b) 识别从MI::foo中可见的成员的集合。

#### 【解答】

从MI中可见的成员名字包括MI中定义的成员名，以及在其直接和间接基类中public和protected部分定义的成员名：print、ival、dvec、sval、dval、fval和cval。其中，从多个基类中都可见的名字有：print（从Driven、Base2和Base1均可见）和dval（从Base1和Driven均可见）。

从MI::foo中可见的成员的集合如下：MI::print、MI::ival、MI::dvec、Driven::print、Driven::sval、Driven::dval、Base1::print、Base1::ival、Base1::dval、Base1::cval、Base2::print和Base2::fval。

### 习题17.30

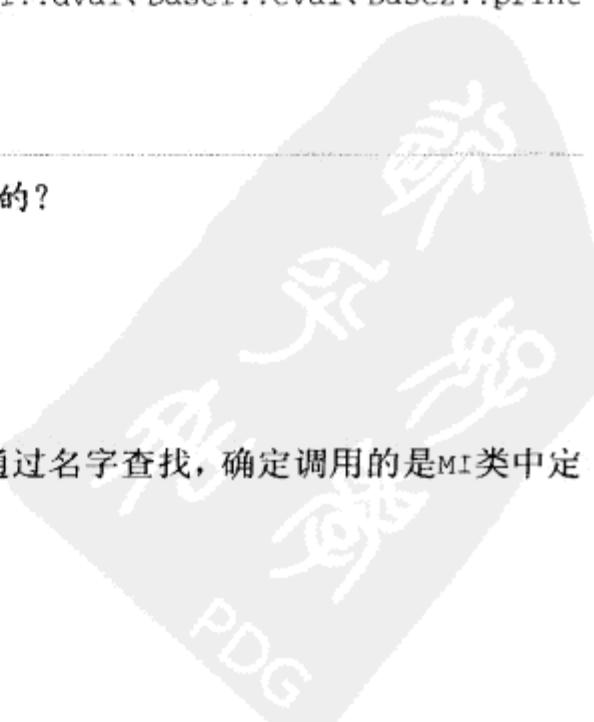
给定前面给出的类层次，为什么下面这个print调用是错误的？

```
MI mi;
mi.print(42);
```

修改MI，使得这个print调用可以正确编译和执行。

#### 【解答】

因为mi.print(42);通过MI类对象调用print函数，编译器通过名字查找，确定调用的是MI类中定



义的print，但MI类中定义的print需要std::vector<double>类型的参数，所以该调用是错误的。

将MI中print的声明改为void print(int);，该print调用即可正确编译和执行。

### 习题17.31

使用前面给出的类层次，如果下面的赋值中有错误的，识别哪些是错误的：

```
void MI::bar() {
    int sval;
    // exercise questions occur here ...
}

(a) dval = 3.14159;      (b) cval = 'a';    (c) id = 1;
(d) fval = 0;            (e) sval = *ival;
```

### 【解答】

错误的有：

- dval = 3.14159; MI的基类 Derived 和 Base1 中都定义了成员 dval，此处无法确定使用哪个 dval。
- id = 1; 使用的是 Base1 中定义的成员 id，但该成员为 private 成员，不能在 MI 中使用，并且，对指针 id 赋以 int 型值 2 也是错误的。

### 习题17.32

使用前面给出的类层次以及下面的MI::foobar成员函数框架

```
void MI::foobar(double cval)
{
    int dval;
    // exercise questions occur here ...
}
```

- (1) 将Base1的dval成员与Derived的dval成员的和赋给dval的局部实例。
- (2) 将MI::dvec中最后一个元素赋给Base2::fval。
- (3) 将Base1的cval赋给Derived中sval的第一个字符。

### 【解答】

- (1) dval = Base1::dval + Derived::dval;
- (2) fval = dvec.back();
- (3) sval[0] = cval;

注意，fval、dvec、sval和cval均只在一个类中定义，所以可以不用限定名访问。

### 习题17.33

给定下面的类层次，从VMI类内部可以不加限定地访问哪些继承成员？哪些继承成员需要限定？解释你的推理。

```
class Base {
public:
    bar(int);
protected:
```

```

        int ival;
    };
    class Derived1 : virtual public Base {
public:
    bar(char);
    foo(char);
protected:
    char cval;
};
class Derived2 : virtual public Base {
public:
    foo(int);
protected:
    int ival;
    char cval;
};
class VMI : public Derived1, public Derived2 {};

```

**【解答】**

从VMI类内部可以不加限定地访问继承成员bar和ival：bar在共享基类Base和派生类Derived1中都存在，但特定派生类实例的优先级高于共享基类实例，所以在VMI类内部不加限定地访问bar，则访问到的是Derived1中的bar实例。ival在共享基类Base和派生类Derived2中都存在；同理，在VMI类内部不加限定地访问ival，访问到的是Derived2中的ival实例。

继承成员foo和cval需要限定：二者在Derived1和Derived2中都存在，Derived1和Derived2均为Base的派生类，访问优先级相同，所以，如果在VMI类内不加限定地访问foo和cval，则出现二义性。

**习题17.34**

有一种情况下派生类不必为虚基类提供初始化式，这种情况是什么？

**【解答】**

派生类不必为虚基类提供初始化式的情况是：虚基类具有默认构造函数（显式提供或由编译器合成）。

**习题17.35**

给定下面类层次，

```

class Class { ... };
class Base : public Class { ... };
class Derived1 : virtual public Base { ... };
class Derived2 : virtual public Base { ... };
class MI : public Derived1,
           public Derived2 { ... };
class Final : public MI, public Class { ... };

```

(a) 对于Final对象的定义，构造函数和析构函数的次序是什么？

(b) 一个Final对象中有几个Base子对象？有几个Class子对象？

(c) 下面哪个赋值在编译时有错？

```

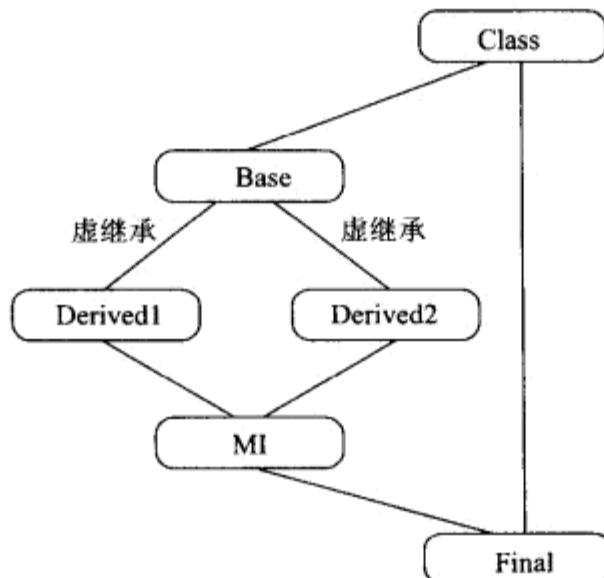
Base *pb; Class *pc;
MI *pmi; Derived2 *pd2;
(i) pb = new Class;      (iii) pmi = pb;
(ii) pc = new Final;     (iv) pd2 = pmi;

```

**【解答】**

(a) 该类层次可表示如下：





注意，此处采用了与*C++ Primer*（第4版）一致的表示方式。如果采用UML类图表示，则有所不同。如果定义Final对象，则创建该对象时按如下次序调用构造函数：

```

Class();
Base();
Derived1();
Derived2();
MI();
Class();
Final();
  
```

注意，首先调用虚基类Base的构造函数（导致调用Class构造函数），然后按声明次序调用非虚基类的构造函数：首先是MI()，它导致调用Derived1()和Derived2()，然后是Class()，最后调用Final类的构造函数。

撤销Final对象时调用析构函数的次序与调用构造函数的次序相反，为

```

~Final();
~Class();
~MI();
~Derived2();
~Derived1();
~Base();
~Class();
  
```

(b) 一个Final对象中只有一个Base子对象，有两个Class子对象。

(c) 错误的有：

- (i) pb = new Class; // 不能用派生类指针指向基类对象
- (iii) pmi = pb; // 不能用指向基类的指针对指向派生类的指针进行赋值

### 习题17.36

给定前面的层次，并假定Base定义了下面三个构造函数，定义从Base派生的类。给每个类同样的三个构造函数，每个构造函数使用实参初始化其Base部分。

```

struct Base {
    Base();
    Base(std::string);
  
```

```

        Base(const Base&);
protected:
    std::string name;
};

```

**【解答】**

各个类的概略定义如下：

```

class Class {
    //...
};

class Base : public Class {
    //...
public:
    Base() : name("Base") { }
    Base(std::string s) : name(s) { }
    Base(const Base& b) : name(b.name) { }
protected:
    std::string name;
};

class Derived1 : virtual public Base {
    //...
public:
    Derived1() : Base("Derived1") { }
    Derived1(std::string s) : Base(s) { }
    Derived1(const Derived1& d) : Base(d) { }
};

class Derived2 : virtual public Base {
    //...
public:
    Derived2() : Base("Derived2") { }
    Derived2(std::string s) : Base(s) { }
    Derived2(const Derived2& d) : Base(d) { }
};

class MI : public Derived1,
public Derived2 {
    //...
public:
    MI() : Base("MI") { }
    MI(std::string s) : Base(s), Derived1(s), Derived2(s) { }
    MI(const MI& m) : Base(m), Derived1(m), Derived2(m) { }
};

class Final : public MI, public Class {
    //...
public:
    Final() : Base("Final") { }
    Final(std::string s) : Base(s), MI(s) { }
    Final(const Final& f) : Base(f), MI(f) { }
};

```

注意，此处的 Class 类具有默认构造函数；每个类需要为其直接基类提供初始化式，除此之外，任何直接或间接继承虚基类 Base 的类必须为 Base 类提供初始化式，否则将无法创建相应类的独立对象。

## 特殊工具与技术

### 习题18.1

实现自己的Vector类的版本，包括vector成员reserve（9.4节）、resize（9.3.5节）以及const和非const下标操作符（14.5节）。

#### 【解答】

Vector类的头文件如下：

```
// Vector.hpp
// Vector类的头文件
// 实现自己的Vector类的版本，包括vector成员reserve、resize、size、
// capacity以及const和非const下标操作符
#ifndef MYVECTOR_H
#define MYVECTOR_H
#include <memory>
#include <cstddef>
using namespace std;

template <class T> class Vector {
public:
    Vector(): elements(0), first_free(0), end(0) { }
    void push_back(const T&);
    void reserve(const size_t capa);

    // 调整Vector大小，使其能容纳n个元素：
    // 如果n小于Vector当前大小，则删除多余元素；
    // 否则，添加采用值初始化的新元素
    void resize(const size_t n);

    // 调整Vector大小，使其能容纳n个元素：所有新添加的元素值都为t
    void resize(const size_t n, const T& t);

    // 下标操作符
    T& operator[](const size_t);
    const T& operator[](const size_t) const;

    // 返回Vector大小
    size_t size()
    {   return first_free - elements; }

    // 返回Vector容量
    size_t capacity()
    {   return end - elements; }
private:
    static std::allocator<T> alloc; // 用于获取未构造内存的对象
    void reallocate(); // 获取更多空间并复制现有元素
```

```

T* elements;           // 指向第一个元素的指针
T* first_free;        // 指向第一个自由元素的指针
T* end;               // 指向数组末端的下一元素位置的指针
};

#include "Vector.cpp"    // 引入Vector类的实现文件
#endif

```

Vector类的实现文件如下：

```

// Vector.cpp
// Vector类的实现文件(源文件)
template <class T> allocator<T> Vector<T>::alloc;

template <class T> void Vector<T>::push_back(const T& t)
{
    if (first_free == end)
        // 已用完所分配的空间
        reallocate(); // 获取更多空间并复制现有元素至新分配空间
    alloc.construct(first_free, t);
    ++first_free;
}

template <class T> void Vector<T>::reallocate()
{
    // 计算当前大小并分配两倍于当前元素数的空间
    ptrdiff_t size = first_free - elements;
    ptrdiff_t newcapacity = 2 * max(size, 1);

    // 分配空间以保存newcapacity个T类型的元素
    T* newelements = alloc.allocate(newcapacity);

    // 在新空间中构造现有元素的副本
    uninitialized_copy(elements, first_free, newelements);

    // 逆序撤销旧元素
    for (T *p = first_free; p != elements; /* empty */)
        alloc.destroy(--p);

    // 不能用0值指针调用deallocate
    if (elements)
        // 释放保存元素的内存
        alloc.deallocate(elements, end - elements);

    // 使数据结构指向新元素
    elements = newelements;
    first_free = elements + size;
    end = elements + newcapacity;
}

template <class T>
void Vector<T>::reserve(const size_t capa)
{
    // 计算当前数组的大小
    size_t size = first_free - elements;

    // 分配可保存capa个T类型元素的空间
    T* newelements = alloc.allocate(capa);

    // 在新分配的空间中构造现有元素的副本
    if (size <= capa)
        uninitialized_copy(elements, first_free, newelements);
}

```

```

else
    // 如果capa小于原来数组的size，则去掉多余的元素
    uninitialized_copy(elements, elements + capa, newelements);

    // 逆序撤销旧元素
    for (T *p = first_free; p != elements; /* 表达式为空 */)
        alloc.destroy(--p);

    if (elements)
        // 释放旧元素占用的内存
        alloc.deallocate(elements, end - elements);

    // 使数据结构指向新元素
    elements = newelements;
    first_free = elements + min(size, capa);
    end = elements + capa;
}

template <class T>
void Vector<T>::resize(const size_t n)
{
    // 计算当前大小及容量
    size_t size = first_free - elements;
    size_t capacity = end - elements;

    if (n > capacity) {
        reallocate(); // 获取更多空间并复制现有元素
        // 添加采用值初始化的新元素
        uninitialized_fill(elements + size, elements + n, T());
    } else if (n > size)
        // 添加采用值初始化的新元素
        uninitialized_fill(elements + size, elements + n, T());
    else
        // 逆序撤销多余元素
        for (T *p = first_free; p != elements + n; /* 表达式为空 */)
            alloc.destroy(--p);

    // 使数据结构指向新元素
    first_free = elements + n;
}

template <class T>
void Vector<T>::resize(const size_t n, const T& t)
{
    // 计算当前大小及容量
    size_t size = first_free - elements;
    size_t capacity = end - elements;

    if (n > capacity) {
        reallocate(); // 获取更多空间并复制现有元素
        // 添加值为t的新元素
        uninitialized_fill(elements + size, elements + n, t);
    } else if (n > size)
        // 添加值为t的新元素
        uninitialized_fill(elements + size, elements + n, t);
    else
        // 逆序撤销多余元素
        for (T *p = first_free; p != elements + n; /* 表达式为空 */)
            alloc.destroy(--p);

    // 使数据结构指向新元素
    first_free = elements + n;
}

```

```

template <class T>
T& Vector<T>::operator[] (const size_t index)
{
    return elements[index];
}

template <class T>
const T& Vector<T>::operator[] (const size_t index) const
{
    return elements[index];
}

```

注意, Microsoft Visual C++ .NET 2003 支持模板的包含编译模型, 但不要将模板类的实现文件显式地加入到project中, 否则会引起编译错误。

### 习题18.2

定义一个类型别名, 使用对应指针类型作为Vector的iterator。

#### 【解答】

在Vector类的定义体中定义如下类型别名:

```
typedef T* iterator;
```

则Vector类的定义中用到T\*的地方都可用iterator代替。

如果将该类型别名定义在类的public部分, 同时在类中提供如下两个公有成员函数:

```

// 返回指向第一个元素的迭代器
iterator begin()
{   return elements; }

// 返回指向最后一个元素的下一个位置的迭代器
iterator last()
{   return first_free; }

```

则可以类似于标准库中的vector类通过Vector对象使用iterator, 例如:

```

Vector<int> ivec;
//...
// 输出Vector对象的元素
for (Vector<int>::iterator iter = ivec.begin();
     iter != ivec.last(); ++iter)
    cout << *iter << "\t";

```

### 习题18.3

为了测试你的Vector类, 重新实现前面用vector编写的程序, 用Vector代替vector。

#### 【解答】

前面用vector编写的程序很多, 因篇幅所限, 此处不一一改编。下面给出一个使用Vector类的实例程序:

```

// 18-3.cpp
// 使用元素类型为int和string的Vector, 以测试Vector类
#include "Vector.hpp" // 引入习题18.1解答中定义的Vector类
#include <iostream>
#include <string>
using namespace std;

```

```

int main()
{
    // 使用元素为int型的Vector
    Vector<int> ivec;
    for (size_t i = 0; i != 8; ++i){
        ivec.push_back(i);
        cout << ivec[i] << "\t";
    }
    cout << endl;

    // 改变Vector容量及size并输出相关数据
    cout << ivec.size() << "\t" << ivec.capacity() << endl;
    ivec.reserve(50);
    cout << ivec.size() << "\t" << ivec.capacity() << endl;
    ivec.resize(20);
    for (size_t i = 0; i != 20; ++i)
        cout << ivec[i] << "\t";
    cout << endl;

    cout << ivec.size() << "\t" << ivec.capacity() << endl;
    cout << ivec[2] << "\t" << ivec[10] << endl;

    // 使用元素为string型的Vector
    Vector<string> svec;
    string s;

    // 读入Vector元素
    cout << "Enter some strings(Ctrl+Z to end):" << endl;
    while (cin >> s)
        svec.push_back(s);

    // 输出Vector元素
    for (size_t i = 0; i != svec.size(); ++i)
        cout << svec[i] << "\t";
    cout << endl;

    // 改变Vector容量及size并输出相关数据
    cout << svec.size() << "\t" << svec.capacity() << endl;
    svec.reserve(50);
    cout << svec.size() << "\t" << svec.capacity() << endl;
    svec.resize(20);
    for (size_t i = 0; i != 20; ++i)
        cout << svec[i] << "\t";
    cout << endl;

    cout << svec.size() << "\t" << svec.capacity() << endl;
    cout << svec[2] << "\t" << svec[10] << endl;

    return 0;
}

```

#### 习题18.4

你认为为什么限制construct函数只能使用元素类型的复制构造函数？

#### 【解答】

因为allocator类提供的是可感知类型的内存分配，限制construct函数只能使用元素类型的复制构造函数，可以获得更高的类型安全性。

习题18.5

为什么定位new表达式更灵活？

### 【解答】

因为定位new表达式初始化一个对象时可以使用任何构造函数，并直接建立对象，而construct函数总是使用复制构造函数。对于某些特定类，因复制构造函数为私有的而不能使用，或考虑到性能而需要避免使用复制构造函数，则使用定位new表达式可以满足需要。

习题18.6

重新实现Vector类，使用operator new、operator delete和定位new表达式，并直接调用析构函数。

### 【解答】

对习题 18.1 解答中给出的 Vector 类进行修改，得到如下 Vector 类的定义：

```

template <class T> std::allocator<T> Vector<T>::alloc;

template <class T> void Vector<T>::push_back(const T& t)
{
    if (first_free == end)
        // 已用完所分配的空间
        reallocate(); // 索取更多空间并复制现有元素至新分配空间

    // 使用定位new表达式在first_free所指向的内存中构造t的副本
    new (first_free) T(t);
    ++first_free;
}

template <class T> void Vector<T>::reallocate()
{
    // 计算当前大小并分配两倍于当前元素数的空间
    ptrdiff_t size = first_free - elements;
    ptrdiff_t newcapacity = 2 * max(size, 1);

    // 使用operator new分配未构造内存以保存newcapacity个T类型的元素
    T* newelements = static_cast<T*>
        (operator new[](newcapacity * sizeof(T)));

    // 在新空间中构造现有元素的副本
    uninitialized_copy(elements, first_free, newelements);
    // 显式调用析构函数逆序撤销旧元素
    for (T *p = first_free; p != elements; /* empty */)
        (--p) -> ~T();

    if (elements)
        // 使用operator delete释放保存元素的内存
        operator delete[](elements);

    // 使数据结构指向新元素
    elements = newelements;
    first_free = elements + size;
    end = elements + newcapacity;
}

template <class T>
void Vector<T>::reserve(const size_t capa)
{
    // 计算当前数组的大小
    size_t size = first_free - elements;

    // 使用operator new分配未构造内存以保存capa个T类型元素
    T* newelements = static_cast<T*>
        (operator new[](capa * sizeof(T)));

    // 在新分配的空间中构造现有元素的副本
    if (size <= capa)
        uninitialized_copy(elements, first_free, newelements);
    else
        // 如果capa小于原来数组的size，则去掉多余的元素
        uninitialized_copy(elements, elements + capa, newelements);

    // 显式调用析构函数逆序撤销旧元素
    for (T *p = first_free; p != elements; /* 表达式为空 */)
        (--p) -> ~T();
}

```

```

if (elements)
    // 使用operator delete释放旧元素占用的内存
    operator delete[](elements);

// 使数据结构指向新元素
elements = newelements;
first_free = elements + min(size, capa);
end = elements + capa;
}

template <class T>
void Vector<T>::resize(const size_t n)
{
    // 计算当前大小及容量
    size_t size = first_free - elements;
    size_t capacity = end - elements;

    if (n > capacity) {
        reallocate(); // 获取更多空间并复制现有元素
        // 添加采用值初始化的新元素
        uninitialized_fill(elements + size, elements + n, T());
    } else if (n > size)
        // 添加采用值初始化的新元素
        uninitialized_fill(elements + size, elements + n, T());
    else
        // 显式调用析构函数逆序撤销多余元素
        for (T *p = first_free; p != elements + n; /* 表达式为空 */)
            (*--p) ~T();

    // 使数据结构指向新元素
    first_free = elements + n;
}

template <class T>
void Vector<T>::resize(const size_t n, const T& t)
{
    // 计算当前大小及容量
    size_t size = first_free - elements;
    size_t capacity = end - elements;

    if (n > capacity) {
        reallocate(); // 获取更多空间并复制现有元素
        // 添加值为t的新元素
        uninitialized_fill(elements + size, elements + n, t);
    } else if (n > size)
        // 添加值为t的新元素
        uninitialized_fill(elements + size, elements + n, t);
    else
        // 显式调用析构函数逆序撤销多余元素
        for (T *p = first_free; p != elements + n; /* 表达式为空 */)
            (*--p) ~T();

    // 使数据结构指向新元素
    first_free = elements + n;
}

template <class T>
T& Vector<T>::operator[] (const size_t index)
{
    return elements[index];
}

```

```

template <class T>
const T& Vector<T>::operator[] (const size_t index) const
{
    return elements[index];
}

#endif

```

所进行的修改包括：

- 分配新空间时不用 `allocator::allocate` 而改用 `operator new`。
- 释放旧空间时不用 `allocator::deallocate` 而改用 `operator delete`。
- 在特定内存位置构造对象不用 `allocator::construct` 而改用定位 `new` 表达式。
- 清除元素时不用 `allocator::destroy` 而改为直接调用元素所属类型的析构函数。

注意，因为笔者所使用的编译器支持模板的包含编译模型，所以为节省篇幅起见，未区分该类的头文件和实现文件。

### 习题 18.7

运行为原来的 `Vector` 实现而运行的程序，测试你的新版本。

#### 【解答】

因为 `vector` 的新版本中类的接口保持不变，所以，使用 `vector` 的新版本，习题 18.3 解答中给出的示例程序可不加修改地运行。

### 习题 18.8

你认为哪个版本更好？为什么？

#### 【解答】

使用 `allocator` 类管理内存的版本（习题 18.1 的解答中给出）更好。

该版本的代码相比较而言要简单些（如分配空间时无需进行指针类型的强制转换），更重要的是 `allocator` 类提供可感知类型的内存管理，更加安全也更加灵活。但是，也需要注意到，在构造对象时使用定位 `new` 表达式比 `allocator::construct` 要灵活一些（见习题 18.5 的解答）。

### 习题 18.9

为 `QueueItem` 类声明成员 `new` 和 `delete`。

#### 【解答】

可定义如下 `QueueItem` 类：

```

template <class Type> class QueueItem {
    friend class Queue<Type>;
    QueueItem(const Type &t): item(t), next(0) { }
    Type item;
    QueueItem *next;
    // 声明成员new和delete
    void* operator new(size_t);
    void operator delete(void*, size_t);
};

```



**习题18.10**

解释下面每个初始化，指出是否有错误的；如果有，为什么错。

```
class iStack {
public:
    iStack(int capacity): stack(capacity), top(0) { }
private:
    int top;
    vector<int> stack;
};
```

- (a) iStack \*ps = new iStack(20);
- (b) iStack \*ps2 = new const iStack(15);
- (c) iStack \*ps3 = new iStack[ 100 ];

**【解答】**

(a) new表达式创建一个iStack对象，其top成员初始化为0，stack成员初始化为包含20个0值int型元素的vector，并返回指向该iStack对象的指针。用new表达式的返回值对指针ps进行初始化（使ps指向该iStack对象）。

(b) new表达式创建一个const iStack对象，其top成员初始化为0，stack成员初始化为包含15个0值int型元素的vector，返回指向该iStack对象的指针，并试图用new表达式的返回值对指针ps2进行初始化。其中有错误：new表达式返回一个const iStack型指针，不能用该指针对iStack型指针ps2进行初始化。

(c) 试图使用new表达式分配一个包含100个iStack对象的数组，并用该数组的首地址对指针ps3进行初始化。其中有错误：使用new表达式动态分配数组时，如果数组元素具有类类型，将使用该类的默认构造函数实现初始化（见4.3.1节），因此分配iStack对象数组的new表达式需要使用iStack类的默认构造函数对数组元素进行初始化，但iStack类没有提供默认构造函数，因而出错。

**习题18.11**

解释下面的new和delete表达式中发生什么。

```
struct Exercise {
    Exercise();
    ~Exercise();
};
Exercise *pe = new Exercise[20];
delete[] pe;
```

**【解答】**

new表达式动态分配包含20个Exercise对象元素的数组，并调用Exercise类的默认构造函数对数组元素进行初始化。

delete表达式调用Exercise类的析构函数清除由new表达式动态分配的数组中的每个对象，并释放该数组所占用的内存。

**习题18.12**

为Queue类或你选择的其他类实现一个类特定的内存分配器。测量性能的改变，看看到底有多大

帮助。

### 【解答】

首先需要实现18.1.7节提出的内存分配器基类CachedObj:

```
// CachedObj.hpp(for 18-12)
#ifndef CACHEDOBJ_H
#define CACHEDOBJ_H
#include <cstddef>
#include <memory>
#include <stdexcept>

// 内存分配类: 预分配对象并维持一个自由列表
// 释放对象时, 将其放入自由列表中
// 程序退出时才将内存返还给系统
template <class T> class CachedObj {
public:
    void *operator new(std::size_t);
    void operator delete(void *, std::size_t);
    virtual ~CachedObj() { }
protected:
    T *next;
private:
    static void add_to_freelist(T*);
    static std::allocator<T> alloc_mem;
    static T *freeStore;
    static const std::size_t chunk;
};

// 定义静态数据成员
template <class T> std::allocator<T> CachedObj<T>::alloc_mem;
template <class T> T *CachedObj<T>::freeStore = 0;
template <class T> const std::size_t CachedObj<T>::chunk = 24;

template <class T>
void *CachedObj<T>::operator new(std::size_t sz)
{
    // new只创建T类对象, 不能用于创建T的派生类的对象:
    // 因此需要检查所要求分配内存的数量
    if (sz != sizeof(T))
        throw std::runtime_error
            ("CachedObj: wrong size object in operator new");

    if (!freeStore) { // 自由列表为空:
        // 使用allocate对象分配chunk个T类型的未构造对象
        T * array = alloc_mem.allocate(chunk);

        // 设置新分配空间中每个对象的next指针
        for (std::size_t i = 0; i != chunk; ++i)
            add_to_freelist(&array[i]);
    }

    T *p = freeStore;
    freeStore = freeStore->CachedObj<T>::next;

    return p; // T的构造函数将构造对象的T类部分
}

template <class T>
void CachedObj<T>::operator delete(void *p, std::size_t)
{
```

```

if (p != 0)
    // 将被delete的对象放在自由列表的头端
    add_to_freelist(static_cast<T*>(p));
}

// 将对象放在自由列表的头端
template <class T>
void CachedObj<T>::add_to_freelist(T *p)
{
    p->CachedObj<T>::next = freeStore;
    freeStore = p;
}

#endif

```

然后，将QueueItem类定义为CachedObj类的派生类，派生的形式如下：

```
class QueueItem: public CachedObj< QueueItem<Type> >
```

QueueItem类的其余部分以及Queue类无需改变。性能改变的幅度与所使用机器的软硬件环境相关。

### 习题18.13

给定下面的类层次，其中每个类都定义了public默认构造函数和虚析构函数：

```

class A { /* ... */ };
class B : public A { /* ... */ };
class C : public B { /* ... */ };
class D : public B, public A { /* ... */ };

```

如果有，下面哪些dynamic\_cast失败？

- (a) A \*pa = new C;  
B \*pb = dynamic\_cast< B\*>(pa);
- (b) B \*pb = new B;  
C \*pc = dynamic\_cast< C\*>(pb);
- (c) A \*pa = new D;  
B \*pb = dynamic\_cast< B\*>(pa);

### 【解答】

使用dynamic\_cast操作符时，如果运行时实际绑定到引用或指针的对象不是目标类型的对象（或其派生类的对象），则dynamic\_cast失败。

(b) dynamic\_cast失败。因为目标类型为c，但pb实际指向的不是c类对象，而是一个b类（c的基类）对象。

注意，要使用RTTI，一般需要在编译器中设置相应编译选项。例如，在Microsoft Visual C++ .NET 2003中，在project菜单中选择properties菜单项，在configuration properties -> C/C++ -> Language中打开RTTI选项。

### 习题18.14

如果D和B都以A为虚基类，习题18.13最后一个转换中将发生什么？

### 【解答】

没什么区别。此处可能是原书作者的疏忽。如果最后一个转换为

```
B *pa = new D;
A *pb = dynamic_cast< A*>(pa);
```

则两种情况下是有区别的：

- 如果 A 不是虚基类，则该转换因有二义性而失败：D 对象中有两个 A 类子对象（一个通过 B 间接继承 A 而来，一个直接继承 A 而来），转换后不知指针 pb 应指向哪个子对象。
- 如果 A 是虚基类，则该转换成功：D 对象中只有一个 A 类子对象，转换后指针 pb 即指向该子对象。

### 习题18.15

使用上面习题中定义的类层次，重写下面代码片段，以便执行dynamic\_cast将表达式\*pa转换为C&类型：

```
if (C *pc = dynamic_cast< C*>(pa))
    // use C's members
} else {
    // use A's members
}
```

### 【解答】

使用dynamic\_cast将基类引用转换为派生类引用时，如果转换失败，会抛出一个std::bad\_cast 异常，因此可以这样重写上述代码片段：

```
try {
    C &c = dynamic_cast< C&>(*pa)
    // use C's members
}
catch (std::bad_cast& bc) {
    // use A's members
}
```

### 习题18.16

解释什么时候可以使用dynamic\_cast代替虚函数。

### 【解答】

如果我们需要在派生类中增加新的成员函数（假设为函数f），但又无法取得基类的源代码，因而无法在基类中增加相应的虚函数，这时，可以在派生类中增加非虚成员函数。但这样一来，就无法用基类指针来调用f。如果在程序中需要通过基类指针（如使用该继承层次的某个类中所包含的指向基类对象的指针数据成员p）来调用f，则必须使用dynamic\_cast将p转换为指向派生类的指针，才能调用f。也就是说，如果无法为基类增加虚函数，就可以使用dynamic\_cast代替虚函数。

### 习题18.17

编写一个表达式，动态地将Query\_base对象的指针强制转换为AndQuery对象的指针。通过使用AndQuery和其他查询类型的对象测试该转换。显示一个语句指出强制转换是否工作，并确信输出与你的表达式匹配。

### 【解答】

假设指针qb的类型为Query\_base\*，则表达式dynamic\_cast<AndQuery\*>(qb)可动态地将Query\_base对象的指针强制转换为AndQuery对象的指针。如果强制转换失败，则转换结果为0。因此，

下面的代码段可测试该转换并输出提示指出强制转换是否工作：

```
if (dynamic_cast<AndQuery*> (qb))
    cout << "success" << endl;
else
    cout << "failure" << endl;
```

注意，第15章中所设计的Query\_base类层次主要是通过句柄类Query使用，各个类的构造函数均为私有的，因此不能直接创建各类的对象。如果要测试18.17~18.19题的解答中给出的代码段，需要为Query\_base类层次中的各个类设计公有的构造函数。

### 习题18.18

编写相同的强制转换，但将Query\_base对象转换为AndQuery的引用。重复测试以确信你的转换正确工作。

#### 【解答】

假设指针qb的类型为Query\_base\*，则表达式dynamic\_cast<AndQuery&> (\*qb)可动态地将Query\_base对象强制转换为AndQuery的引用，下面的代码段可测试该转换并输出提示指出强制转换是否工作：

```
try {
    dynamic_cast<AndQuery&> (*qb);
    cout << "success" << endl;
}
catch(std::bad_cast) {
    cout << "failure" << endl;
}
```

### 习题18.19

编写一个typeid表达式，看两个Query\_base指针是否指向相同的类型，然后检查该类型是否为AndQuery。

#### 【解答】

假设指针qb1和qb2的类型为Query\_base\*，则判断两个Query\_base指针是否指向相同的类型的 typeid表达式如下：

```
typeid(*qb1) == typeid(*qb2)
```

判断该类型是否为AndQuery的 typeid表达式如下：

```
typeid(*qb1) == typeid(AndQuery)
```

### 习题18.20

给定下面的类层次，其中每个类都定义了public默认构造函数及虚析构函数。下面的语句显示哪些类型名？

```
class A { /* ... */ };
class B : public A { /* ... */ };
class C : public B { /* ... */ };
```

- (a) A \*pa = new C;  
cout << typeid(pa).name() << endl;
- (b) C cobj;  
A& ra = cobj;  
cout << typeid(&ra).name() << endl;
- (c) B \*px = new B;  
A& ra = \*px;  
cout << typeid(ra).name() << endl;

**【解答】**

`typeid`操作符的结果是`type_info`类对象，`type_info`类的成员函数`name`返回一个C风格字符串，代表`type_info`对象所表示的类型的名字。返回的C风格字符串的格式和值根据编译器而定。在Microsoft Visual C++ .NET 2003中，上述语句显示的类型名分别为：

- (a) class A \*
- 因为`pa`是指向`A`类对象的指针，其类型为`A*`。
- (b) class A \*
- 因为`ra`是`A`类对象的引用，表达式`&ra`求得`ra`的地址，该地址的类型为`A*`。
- (c) class B
- 因为`ra`是`A`类对象的引用，其当前所绑定的对象是指针`px`所指向的对象，是一个`B`类对象。

**习题18.21**

普通数据指针或函数指针与数据成员指针或函数成员指针之间的区别是什么？

**【解答】**

区别在于：指定成员指针（数据成员指针及函数成员指针）的类型时，除了给出成员本身的类型之外，还必须给出所属类的类型（函数成员指针还要指明成员是否为`const`）。例如：指向`int`型数据的普通数据指针的类型为`int*`，而指向`C`类的`int`型数据成员的成员指针的类型为`int C::*`；指向“不带参数并返回`int`型值的函数”的普通函数指针的类型为`int (*)()`，而指向“`C`类的不带参数并返回`int`型值的`const`成员函数”的函数成员指针的类型为`int (C::*)() const`。

**习题18.22**

定义可以表示`Sales_item`类的`isbn`成员的指针的类型。

**【解答】**

`Sales_item`类的`isbn`成员是一个数据成员，其类型为`std::string`。可以表示`Sales_item`类的`isbn`成员的指针的类型为：

```
std::string Sales_item::*
```

**习题18.23**

定义可以指向`same_isbn`成员的指针。

**【解答】**

`Sales_item`类的`same_isbn`成员是一个函数成员，其原型为：

```
bool same_isbn(const Sales_item&) const;
```

可以指向same\_isbn成员的指针是一个函数成员指针，可如下定义p为指向same\_isbn成员的指针：

```
bool (Sales_item::*p)(const Sales_item&) const
```

### 习题18.24

编写类型别名，作为可指向Sales\_item的avg\_price成员的指针的同义词。

#### 【解答】

Sales\_item的avg\_price成员是一个函数成员，其原型为：

```
double avg_price() const;
```

定义如下类型别名Pt，作为可指向Sales\_item的avg\_price成员的指针的同义词：

```
typedef double (Sales_item::*Pt)() const;
```

### 习题18.25

Screen类的成员cursor<sup>1</sup>的类型是什么？

#### 【解答】

Screen类的成员cursor的类型是Screen::index，即std::string::size\_type。

### 习题18.26

定义一个可以指向Screen类cursor成员的成员指针，通过该指针获取Screen::cursor的值。

#### 【解答】

指向Screen类cursor成员的成员指针pm可定义如下：

```
Screen::index Screen::*pm = &Screen::cursor;
```

可使用成员指针解引用操作符（.\*）从对象或引用获取成员，使用成员指针箭头操作符（->\*）通过对象的指针获取成员。

假设有如下对象定义：

```
Screen myScreen;
```

则可以这样通过成员指针pm获取Screen::cursor的值：

```
myScreen.*pm
```

假设有如下对象指针定义：

```
Screen *pScreen;
```

则可以这样通过成员指针pm获取Screen::cursor的值：

```
pScreen->*pm
```

### 习题18.27

为Screen类成员函数的每个可区分类型定义类型别名。

1. 此处英文原文有误：screen不是Screen类的成员。

**【解答】**

在18.3节中给出的Screen类版本中，成员函数有4个可区分类型，2个get函数为2个不同类型，home等5个光标移动函数为一个类型，move函数为一个类型。因此可定义4个类型别名Pmf1、Pmf2、Pmf3和Pmf4：

```
typedef char (Screen::*Pmf1)() const;
typedef char (Screen::*Pmf2)(Screen::index, Screen::index) const;
typedef Screen& (Screen::*Pmf3)();
typedef Screen& (Screen::*Pmf4)(Screen::Directions);
```

**习题18.28**

也可以将成员指针声明为类的数据成员。修改Screen类的定义，以便包含与home<sup>1</sup>类型相同的Screen成员函数的指针。

**【解答】**

```
class Screen {
public:
    // 成员函数指针的类型别名
    typedef Screen& (Screen::*PmfType)();

    typedef std::string::size_type index;

    char get() const;
    char get(index ht, index wd) const;
    // 光标移动函数
    Screen& home();
    Screen& forward();
    Screen& back();
    Screen& up();
    Screen& down();
private:
    std::string contents;
    index cursor;
    index height, width;
public:
    // 成员函数指针
    PmfType pmf;
};
```

**习题18.29**

编写一个Screen构造函数，该构造函数接受指向Screen成员函数的指针类型形参，成员函数的形参表和返回类型与成员函数home<sup>2</sup>的相同。

**【解答】**

该构造函数接受指向Screen成员函数的指针类型形参，并对成员函数指针进行初始化：

```
Screen(PmfType p) : pmf(p) {}
```

1. 此处英文原文有误：end不是Screen类的成员。

2. 此处英文原文有误：end不是Screen类的成员。

**习题18.30**

为该形参提供默认实参，使用该形参对上题中引入的数据成员进行初始化。

**【解答】**

可将成员函数home作为默认实参，构造函数如下：

```
Screen(PmfType p = &Screen::home) : pmf(p) {}
```

**习题18.31**

提供一个Screen成员函数来设置这个成员。

**【解答】**

提供成员函数setPmf来设置成员函数指针成员：

```
void setPmf(PmfType p)
{
    pmf = p;
}
```

Screen类的完整定义体如下：

```
class Screen {
public:
    // 成员函数指针的类型别名
    typedef Screen& (Screen::*PmfType) ();

    typedef std::string::size_type index;

    // 接受函数指针形参的构造函数
    Screen(PmfType p = &Screen::home) : pmf(p) {}

    // 设置成员函数指针的函数
    void setPmf(PmfType p)
    {
        pmf = p;
    }

    char get() const;
    char get(index ht, index wd) const;
    // 光标移动函数
    Screen& home();
    Screen& forward();
    Screen& back();
    Screen& up();
    Screen& down();
private:
    std::string contents;
    index cursor;
    index height, width;
public:
    // 成员函数指针
    PmfType pmf;
};

// ...其他成员函数的实现(略)
```

注意，如果将成员函数指针定义为public数据成员，则可按如下方式使用该指针：

```
Screen myScreen;
myScreen.setPmf(&Screen::down);
```

```
(myScreen.*(myScreen.pmf))(); // 等价于myScreen.down();
```

如果将成员函数指针定义为private数据成员，则应该提供一个public成员函数（如getPmf）来获取该指针，因此可按如下方式使用该指针：

```
Screen myScreen;
myScreen.setPmf(&Screen::down);
(myScreen.*(myScreen.getPmf()))(); // 等价于myScreen.down();
```

getPmf函数可定义如下：

```
PmfType getPmf()
{
    return pmf;
}
```

### 习题18.32

重新实现第16章的Queue类和QueueItem类，使QueueItem成为Queue内部的嵌套类。

#### 【解答】

主要修改：在Queue类内部的private部分定义嵌套类QueueItem，并将类中所有对QueueItem<Type>的使用改为QueueItem。

程序如下：

```
template <class Type> class Queue {
public:
    // 构造空Queue
    Queue(): head(0), tail(0) {}

    // 复制控制成员
    Queue(const Queue &Q): head(0), tail(0)
        { copy_elems(Q); }
    Queue& operator=(const Queue&);

    ~Queue() { destroy(); }

    // 返回Queue中的队头元素
    // 不作检查：对空Queue执行此操作是未定义的行为
    Type& front()
    {
        return head->item;
    }
    const Type &front() const
    {
        return head->item;
    }

    // 在Queue的尾部增加元素
    void push(const Type &);

    // 删除队头元素
    void pop();

    // 如果Queue中没有元素，则empty返回true
    bool empty () const
    {
        return head == 0;
    }
}
```



```

private:
    struct QueueItem {
        QueueItem(const Type& );
        Type item;
        QueueItem *next;
    };
    QueueItem *head;           // 指向Queue中第一个元素的指针
    QueueItem *tail;          // 指向Queue中最后一个元素的指针

    // 复制控制成员所使用的辅助函数
    void destroy();           // 删除所有元素
    void copy_elems(const Queue&); // 从形参复制元素
};

template <class Type> void Queue<Type>::destroy()
{
    while (!empty())
        pop();
}

template <class Type> void Queue<Type>::pop()
{
    // 不做检查：在空Queue上执行pop操作的行为未定义
    QueueItem *p = head;      // 保存head指针以便于删除它所指向的元素
    head = head->next;       // 修改head指针指向下一个元素
    delete p;                // 删除原队头元素
}

template <class Type> void Queue<Type>::push(const Type &val)
{
    // 创建新的QueueItem对象
    QueueItem *pt = new QueueItem(val);

    // 将QueueItem对象放入队列
    if (empty())
        head = tail = pt; // 队列中只有一个元素
    else {
        tail->next = pt; // 新元素放在队尾
        tail = pt;
    }
}

template <class Type>
void Queue<Type>::copy_elems(const Queue &orig)
{
    // 从orig复制元素至该Queue
    // 到达orig.tail时, pt == 0, 循环结束
    for (QueueItem *pt = orig.head; pt; pt = pt->next)
        push(pt->item); // 复制元素
}

template <class Type>
Queue<Type>& Queue<Type>::operator = (const Queue &orig)
{
    head = tail = 0;
    copy_elems(orig);

    return *this;
}

// 定义Queue类中的嵌套类QueueItem的构造函数

```

```
template<class Type>
Queue<Type>::QueueItem::QueueItem(const Type &t):
    item(t), next(0) { }
```

**习题18.33**

解释Queue设计的原来版本和嵌套类版本的优缺点。

**【解答】**

原来的版本中，将QueueItem类设计为Queue类的伙伴执行类，因此QueueItem类是私有类（它只有private成员）。普通用户代码不能使用QueueItem类的对象：它的所有成员（包括构造函数）均为private。但是QueueItem类是在全局作用域中定义的，名字QueueItem是全局可见的，因而不能将标识符QueueItem用作其他自有类型或其他实体的名字。

嵌套类版本中，将QueueItem类设计为Queue类的私有成员，从而，Queue类（及其友元）可以使用QueueItem。QueueItem类类型对普通用户代码不可见，因此普通用户代码不能使用QueueItem类，但可以将标识符QueueItem用作其他自有类型或其他实体的名字。

比较而言，嵌套类版本是一个更好的设计。

**习题18.34**

解释下面这些声明，并指出它们是否合法：

```
extern "C" int compute(int *, int);
extern "C" double compute(double *, double);
```

**【解答】**

第一个声明指出：compute是一个用C语言编写的函数，该函数接受一个int\*类型及一个int类型的形参，返回int型值。

第二个声明指出：compute是一个用C语言编写的函数，该函数接受一个double\*类型及一个double类型的形参，返回double型值。

如果这两个声明单独出现，则是合法的；如果二者同时出现，则是不合法的，因为这两个compute函数构成了函数重载，而C语言是不支持函数重载的。

