

本书各版全球总销量达1300万册  
为新的C++11标准重新撰写

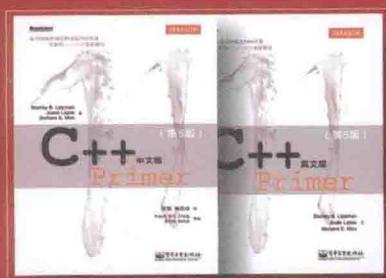
# C++ Primer 习题集

(第5版)

Stanley B. Lippman

[美] Josée Lajoie 著  
Barbara E. Moo

王刚 杨巨峰 李忠伟 改编



电子工业出版社  
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY  
<http://www.phei.com.cn>

( 第5版 )

# C++ Primer

## 习题集

Stanley B. Lippman  
[美] Josée Lajoie 著  
Barbara E. Moo  
王刚 杨巨峰 李忠伟 改编

## 内 容 简 介

*C++ Primer (Fifth Edition)* 由三位 C++ 大师 Stanley B. Lippman、Josée Lajoie 和 Barbara E. Moo 合作完成，其中文译本《C++ Primer 中文版（第 5 版）》也已出版。与上一版相比，这一版基于 C++11 标准进行了全面而彻底的内容更新；既是初学者的最佳学习指南，也是中高级程序员不可或缺的参考书。本书作为这部久负盛名的 C++ 经典教程的配套习题解答，提供了 *C++ Primer (Fifth Edition)* 中英文版图书中所有习题的参考答案。

本书对使用 *C++ Primer (Fifth Edition)* 学习 C++ 程序设计语言的读者来说是一本非常理想的参考书。

Authorized translation from the English language edition, entitled *C++ Primer, 5E*, 9780321714114 by STANLEY B. LIPPMAN; JOSEE LAJOIE; BARBARA E. MOO, published by Pearson Education, Inc., Copyright©2013 Pearson Education, Inc.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

Chinese simplified language edition published by PEARSON EDUCATION ASIA LTD., and PUBLISHING HOUSE OF ELECTRONICS INDUSTRY Copyright © 2015.

本书翻译改编专有版权由 Pearson Education 培生教育出版亚洲有限公司授予电子工业出版社。未经出版者预先书面许可，不得以任何方式复制或抄袭本书的任何部分。

本书翻译改编版贴有 Pearson Education 培生教育出版集团激光防伪标签，无标签者不得销售。

版权贸易合同登记号 图字：01-2013-5743

## 图书在版编目（CIP）数据

C++ Primer 习题集：第 5 版 / (美) 李普曼 (Lippman,S.B.), (美) 拉乔伊 (Lajoie,J.), (美) 默 (Moo,B.E.) 著；王刚，杨巨峰，李忠伟改编。—北京：电子工业出版社，2015.3

书名原文：C++ Primer, 5E

ISBN 978-7-121-25229-7

I. ①C… II. ①李… ②拉… ③默… ④王… ⑤杨… ⑥李… III. ①C 语言—程序设计—习题集  
IV. ①TP312-44

中国版本图书馆 CIP 数据核字（2014）第 302751 号

策划编辑：张春雨

责任编辑：刘 脍

印 刷：北京京科印刷有限公司

装 订：三河市皇庄路通装订厂

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱

邮编：100036

开 本：787×1092 1/16

印张：33

字数：740 千字

版 次：2015 年 3 月第 1 版

印 次：2015 年 3 月第 1 次印刷

定 价：89.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：(010) 88254888。

质量投诉请发邮件至 [zlts@phei.com.cn](mailto:zlts@phei.com.cn)，盗版侵权举报请发邮件至 [dbqq@phei.com.cn](mailto:dbqq@phei.com.cn)。

服务热线：(010) 88258888。

# 前言

C++语言是应用极为广泛的一门程序设计语言，难以计数的程序员已经通过旧版的 *C++ Primer* 学会了 C++ 语言。

2011 年，C++ 标准委员会发布了 ISO C++ 标准的一个重要修订版。此修订版是 C++ 进化过程中的最新一步，其目标是使得 C++ 语言更统一、更简单、更安全、更高效。为此，三位 C++ 大师 Stanley B. Lippman、Josée Lajoie 和 Barbara E. Moo 编著完成了 *C++ Primer (Fifth Edition)*，在旧版基础上基于 C++11 标准进行了全面而彻底的内容更新，重点讲解了 C++11 新特性以及这些新特性是如何影响 C++ 语言的。*C++ Primer (Fifth Edition)* 的中文译本《C++ Primer 中文版（第 5 版）》也已由电子工业出版社出版。

本书是 *C++ Primer (Fifth Edition)* 的配套书籍，提供了该书所有习题的参考答案。在设计本书的行文格式时我们做了如下考虑：

- 在原书和习题解答之间建立良好衔接。
- 授之以鱼，不如授之以渔，要让读者明白怎么做以及为什么。
- 与 C++11 的目标保持一致，即简单高效。

在每章的开始，我们设计了“导读”模块，目的是言简意赅地把本章的重要知识点串成一个整体，帮助读者梳理在本章应该学到哪些内容。之后的每个题目分为【出题思路】和【解答】两部分：其中【出题思路】站在出题者的角度阐述题目的考查角度和考查目的，【解答】则负责给出满足题目要求的代码及注释；为了开拓读者的思路，有的题目我们给出了不止一种解决方案。

衷心希望本书能对使用 *C++ Primer (Fifth Edition)* 学习 C++ 的读者有所帮助。由于编者水平有限，书中的不当之处恳请读者批评指正。

编 者  
2014 年 11 月  
于南开园

# 目录

第 1 章 开始 .....	1
练习 1.1~练习 1.25	
第 2 章 变量和基本类型 .....	12
练习 2.1~练习 2.42	
第 3 章 字符串、向量和数组 .....	37
练习 3.1~练习 3.45	
第 4 章 表达式 .....	80
练习 4.1~练习 4.38	
第 5 章 语句 .....	99
练习 5.1~练习 5.25	
第 6 章 函数 .....	120
练习 6.1~练习 6.56	
第 7 章 类 .....	152
练习 7.1~练习 7.58	
第 8 章 IO 库 .....	183
练习 8.1~练习 8.14	
第 9 章 顺序容器 .....	193
练习 9.1~练习 9.52	
第 10 章 泛型算法 .....	234
练习 10.1~练习 10.42	

第 11 章 关联容器 .....	273
练习 11.1~练习 11.38	
第 12 章 动态内存 .....	297
练习 12.1~练习 12.33	
第 13 章 拷贝控制 .....	331
练习 13.1~练习 13.58	
第 14 章 重载运算与类型转换 .....	368
练习 14.1~练习 14.53	
第 15 章 面向对象程序设计 .....	399
练习 15.1~练习 15.42	
第 16 章 模板与泛型编程 .....	424
练习 16.1~练习 16.67	
第 17 章 标准库特殊设施 .....	458
练习 17.1~练习 17.39	
第 18 章 用于大型程序的工具 .....	483
练习 18.1~练习 18.30	
第 19 章 特殊工具与技术 .....	502
练习 19.1~练习 19.26	

# 第 1 章

## 开始

### 导读

在本章，我们初识 C++ 语言，学习了如何编译、运行简单的 C++ 程序。因此，本章习题的一个主要训练内容是熟悉编译工具和集成开发环境。本章还介绍了 C++ 程序的基本结构，如何定义变量，如何进行输入输出，以及如何编写 if、for 和 while 语句，用这些知识来练习编写简单的 C++ 程序是本章练习的另一个重要内容。本章最后简要介绍了如何使用其他人定义的类，因此，本章最后一部分习题是利用书中的 Sales-item 类进行类对象的创建、使用方面的练习。

**练习 1.1：**查阅你使用的编译器的文档，确定它所使用的命名约定。编译并运行第 2 页的 main 程序。

#### 【出题思路】

熟悉编译工具和集成开发环境。

#### 【解答】

首先利用编辑器（如 Linux 系统中的 vim 或 Windows 系统中 Visual Studio 自带的编辑器）输入 main 程序，保存为 .cpp 或 .cc 后缀的源程序文件。然后按书中说明运行 GNU 或微软编译器，将源文件编译为可执行文件。最后执行程序，观察执行结果。

尝试使用命令行方式编译程序，尝试不同编译选项，可以帮助我们更好地了解编译过程，掌握生成所需目标程序的方法。

**练习 1.2：**改写程序，让它返回-1。返回值-1 通常被当作程序错误的标识。重新编译并运行你的程序，观察你的系统如何处理 main 返回的错误标识。

### 【出题思路】

了解 C++ 程序与操作系统间的交互。

### 【解答】

Windows 7 操作系统并不处理或报告程序返回的错误标识，直观上，返回-1 的程序与返回 0 的程序在执行效果上并无不同。但环境变量 ERRORLEVEL 记录了上一个程序的返回值。因此，在控制台窗口执行修改后的程序，接着执行 echo %ERRORLEVEL%，会输出-1。在 Linux 系统中，执行 echo \$? 有类似效果。

**练习 1.3：**编写程序，在标准输出上打印 Hello, World。

### 【出题思路】

C++ 程序基本结构和简单输出的练习。

### 【解答】

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

**练习 1.4：**我们的程序使用加法运算符+来将两个数相加。编写程序使用乘法运算符\*，来打印两个数的积。

### 【出题思路】

简单运算和简单输入输出的练习。

### 【解答】

```
#include <iostream>

int main()
{
    std::cout << "请输入两个数" << std::endl;
    int v1, v2;
    std::cin >> v1 >> v2;
    std::cout << v1 << "和" << v2 << "的积为"
        << v1 * v2 << std::endl;
    return 0;
}
```

**练习 1.5：**我们将所有输出操作放在一条很长的语句中。重写程序，将每个运算对象的打印操作放在一条独立的语句中。

### 【出题思路】

对输出语句进行不同形式的练习，同时让读者体会良好的程序格式。

### 【解答】

```
#include <iostream>

int main()
{
    std::cout << "请输入两个数";
    std::cout << std::endl;
    int v1, v2;
    std::cin >> v1 >> v2;
    std::cout << v1 << "和" << v2 << "的积为"
        << v1 * v2 << std::endl;
    return 0;
}
```

**练习 1.6：**解释下面程序片段是否合法。

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

如果程序是合法的，它输出什么？如果程序不合法，原因何在？应该如何修正？

### 【出题思路】

延续练习 1.5，让读者体会输出语句分段形式容易出现的错误。

### 【解答】

这段代码不合法。

前两行的末尾有分号，表示语句结束，第 2、3 两行为两条新的语句。而这两条语句在“<<”之前缺少了输出流，应在“<<”之前加上“std::cout”，即得到正确的程序。

**练习 1.7：**编译一个包含不正确的嵌套注释的程序，观察编译器返回的错误信息。

### 【出题思路】

一方面了解非法嵌套注释，另一方面体会编译器对较为复杂的错误如何给出错误信息，程序员应如何利用这些信息迅速定位、修改错误。

### 【解答】

对不正确的嵌套注释，不同编译器给出的错误信息可能是不同的，而且通常很难理解。例如，用 tdm-gcc 4.8.1 编译器编译 1.3 节中错误嵌套的程序：

```
#include <iostream>

/*
 * 注释对/* */不能嵌套。
 * “不能嵌套”几个字会被认为是源码,
 * 像剩余程序一样处理
 */

int main()
```

```
{
    return 0;
}
```

编译器会报告：

```
4   error: stray '\262' in program
4   error: stray '\273' in program
```

(篇幅所限，后续错误信息略。)

原因是编译器将第一个“\*/”看作注释结束，之后的中文文字看作下一条语句，从而给出非法字符的错误信息。如果“\*/”之后是英文文字，或是使用其他编译器进行编译，给出的可能是完全不同的错误信息。而且这些错误信息都很难直接与注释错误嵌套挂上钩，程序员需要有一定的经验才能快速定位错误，确定错误原因。

### 练习 1.8：指出下列哪些输出语句是合法的（如果有的话）：

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

预测编译这些语句会产生什么样的结果，实际编译这些语句来验证你的答案（编写一个小程序，每次将上述一条语句作为其主体），改正每个编译错误。

#### 【出题思路】

进一步熟悉更复杂的正确和不正确的注释语句。

#### 【解答】

第一条和第二条语句显然是合法的。

在第三条语句中，第一个双引号被注释掉了，因此<<运算符后真正被编译的内容是“ \*/”，编译器认为这是一个不完整的字符串，所以会报告：

```
7   error: missing terminating " character
```

即，缺少结束的双引号。在分号前补上一个双引号，这条语句就变为正确的了。

第四条看起来很混乱，但它是正确的。第一个双引号被注释掉了，第四个双引号也被注释掉了，第二个双引号和第三个双引号之间的“ /\* ”被认为是字符串的文字内容。但是，这样的程序风格显然是不好的。

### 练习 1.9：编写程序，使用 while 循环将 50 到 100 的整数相加。

#### 【解答】

```
#include <iostream>

int main()
{
    int sum = 0;
    int i = 50;
    while (i <= 100) {
        sum += i;
        i++;
    }
}
```

```

    std::cout << "50 到 100 之间的整数之和为"
    << sum << std::endl;
    return 0;
}

```

**练习 1.10:**除了++运算符将运算对象的值增加 1 以外，还有一个递减运算符( -- )实现将值减少 1。编写程序，使用递减运算符在循环中按递减序打印出 10 到 0 之间的整数。

### 【出题思路】

递减循环较之递增循环用得较少，应有意进行这方面的练习，对提高编程能力是有益的。

### 【解答】

```

#include <iostream>

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

```

**练习 1.11:** 编写程序，提示用户输入两个整数，打印出这两个整数所指定的范围内的所有整数。

### 【出题思路】

编写一个简单但完整的依据用户输入进行处理的实例。让读者体会：用户的输入可能有各种各样的情况，我们编写的程序必须全面地考虑各种情况，避免由于考虑不周使得程序在某些用户输入下产生错误结果甚至更严重的后果。

### 【解答】

```

#include <iostream>

int main()
{
    std::cout << "请输入两个数";
    std::cout << std::endl;
    int v1, v2;
    std::cin >> v1 >> v2;
    if (v1 > v2) // 由大至小打印
        while (v1 >= v2) {
            std::cout << v1 << " ";
            v1--;
        }
}

```

```
else // 由小至大打印
    while (v1 <= v2) {
        std::cout << v1 << " ";
        v1++;
    }
    std::cout << std::endl;
    return 0;
}
```

**练习 1.12:** 下面的 `for` 循环完成了什么功能? `sum` 的终值是多少?

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

**【解答】**

此循环将-100 到 100 之间（包含-100 和 100）的整数相加，`sum` 的终值是 0。

**练习 1.13:** 使用 `for` 循环重做 1.4.1 节中的所有练习（第 11 页）。

**【出题思路】**

让读者体会：对同样的目标，C++语言提供的不同解决方法。

**【解答】**

练习 1.9 的循环版本：

```
#include <iostream>

int main()
{
    int sum = 0;
    for (int i = 50; i <= 100; i++)
        sum += i;
    std::cout << "50 到 100 之间的整数之和为"
        << sum << std::endl;
    return 0;
}
```

练习 1.10 的循环版本：

```
#include <iostream>

int main()
{
    for (int i = 10; i >= 0; i--)
        std::cout << i << " ";
    std::cout << std::endl;
    return 0;
}
```

练习 1.11 的循环版本：

```
#include <iostream>

int main()
```

```

{
    std::cout << "请输入两个数";
    std::cout << std::endl;
    int v1, v2;
    std::cin >> v1 >> v2;
    if (v1 > v2) // 由大至小打印
        for (; v1 >= v2; v1--)
            std::cout << v1 << " ";
    else // 由小至大打印
        for (; v1 <= v2; v1++)
            std::cout << v1 << " ";
    std::cout << std::endl;
    return 0;
}

```

**练习 1.14:** 对比 `for` 循环和 `while` 循环，两种形式的优缺点各是什么？

**【解答】**

在循环次数已知的情况下，`for` 循环的形式显然更为简洁。

而循环次数无法预知时，用 `while` 循环实现更适合。用特定条件控制循环是否执行，循环体中执行的语句可能导致循环判定条件发生变化

**练习 1.15:** 编写程序，包含第 14 页“再探编译”中讨论的常见错误。熟悉编译器生成的错误信息。

**【出题思路】**

继续熟悉编译器对不同错误给出的信息。

**【解答】**

对于复杂程序中的错误，编译器给出的错误信息很可能无法对应到真正的错误位置并给出准确的错误原因。这是很正常的，因为某些时候我们人类都无法准确判断程序员到底犯了什么错误，在当前人工智能技术发展水平下，要求编译器有超越人类的智能是不现实的。

而且，不同的编译器对同一个程序给出的错误信息有可能是有很大差别的。一方面是因为如前所述，很多时候并不存在“唯一正确”的错误原因，编译器（甚至我们人类也是）只能给出它认为最有可能的错误原因；另一方面，不同编译器对同样的错误原因也可能有自己不同的解释方式。

因此，使用几种不同的编译器，编译一些错误的程序，观察编译器给出的错误信息。对今后在大型软件中查找、修改编译错误是很有帮助的。

**练习 1.16:** 编写程序，从 `cin` 读取一组数，输出其和。

**【出题思路】**

练习不定次数的循环，以及输入流结束判断。

**【解答】**

```
#include <iostream>
int main()
{
    int sum = 0, value = 0;
    std::cout << "请输入一些数, 按 Ctrl+Z 表示结束"
        << std::endl;
    for (; std::cin >> value; )
        sum += value;
    std::cout << "读入的数之和为" <<
        sum << std::endl;
    return 0;
}
```

显然, 对于循环次数无法预知的情况, for 循环比 while 循环稍累赘一些。

**练习 1.17:** 如果输入的所有值都是相等的, 本节的程序会输出什么? 如果没有重复值, 输出又会是怎样的?

**【出题思路】**

练习程序分析和手工执行程序。

**【解答】**

如果输入的所有值都相等, 则 while 循环中的 else 分支永远不会执行, 直到输入结束, while 循环退出, 循环后的输出语句打印这唯一的一个值和它出现的次数。

若没有重复值, 则 while 循环中的 if 语句的真值分支永远不会执行, 每读入一个值, 都会进入 else 分支, 打印它的值和出现次数 1。输入结束后, while 循环退出, 循环后的输出语句打印最后一个值和出现次数 1。

**练习 1.18:** 编译并运行本节的程序, 给它输入全都相等的值。再次运行程序, 输入没有重复的值。

**【解答】**

输入:

1 1 1 1 1

程序输出:

1 occurs 5 times

输入:

1 2 3 4 5

程序输出:

1 occurs 1 times

2 occurs 1 times

3 occurs 1 times

4 occurs 1 times

5 occurs 1 times

注意，不要忘了用 Ctrl+Z 表示输入结束。

**练习 1.19：**修改你为 1.4.1 节练习 1.10（第 11 页）所编写的程序（打印一个范围内的数），使其能处理用户输入的第一个数比第二个数小的情况。

#### 【解答】

练习 1.10 的解答已经包含了此功能。

**练习 1.20：**在网站 <http://www.informit.com/title/0321714113> 上，第 1 章的代码目录中包含了头文件 Sales\_item.h。将它拷贝到你自己的工作目录中。用它编写一个程序，读取一组书籍销售记录，将每条记录打印到标准输出上。

#### 【出题思路】

练习如何使用其他人定义的类来创建、使用对象。

#### 【解答】

```
#include <iostream>
#include "Sales_item.h"
int main()
{
    Sales_item book;
    std::cout << "请输入销售记录: "
    << std::endl;
    while (std::cin >> book) {
        std::cout << "ISBN、售出本数、销售额和平均售价为 "
        << book << std::endl;
    }
    return 0;
}
```

**练习 1.21：**编写程序，读取两个 ISBN 相同的 Sales\_item 对象，输出它们的和。

#### 【出题思路】

类对象的更复杂的使用，利用类接口进行运算。

#### 【解答】

```
#include <iostream>
#include "Sales_item.h"
int main()
{
    Sales_item trans1, trans2;
    std::cout << "请输入两条 ISBN 相同的销售记录: "
    << std::endl;
    std::cin >> trans1 >> trans2;
    if (compareIsbn(trans1, trans2))
        std::cout << "汇总信息: ISBN、售出本数、销售额和平均售价为 "
        << trans1 + trans2 << std::endl;
    else
```

```

    std::cout << "两条销售记录的 ISBN 不同" << std::endl;
    return 0;
}

```

**练习 1.22:** 编写程序，读取多个具有相同 ISBN 的销售记录，输出所有记录的和。

### 【出题思路】

练习在处理数据流的过程中“状态”（是否是相同的 ISBN）的保存和变迁。

### 【解答】

```

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

int main()
{
    Sales_item total, trans;
    std::cout << "请输入几条 ISBN 相同的销售记录: "
    << std::endl;
    if (std::cin >> total) {
        while (std::cin >> trans)
            if (compareIsbn(total, trans)) // ISBN 相同
                total = total + trans;
            else { // ISBN 不同
                std::cout << "ISBN 不同" << std::endl;
                return -1;
            }
        std::cout << "汇总信息: ISBN、售出本数、销售额和平均售价为 "
        << total << std::endl;
    }
    else {
        std::cout << "没有数据" << std::endl;
        return -1;
    }
    return 0;
}

```

**练习 1.23:** 编写程序，读取多条销售记录，并统计每个 ISBN（每本书）有几条销售记录。

### 【解答】

```

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

int main()
{
    Sales_item trans1, trans2;
    int num = 1;
    std::cout << "请输入若干销售记录: "
    << std::endl;
    if (std::cin >> trans1) {
        while (std::cin >> trans2)

```

```
if (compareIsbn(trans1, trans2)) // ISBN 相同
    num++;
else { // ISBN 不同
    std::cout << trans1.isbn() << "共有"
        << num << "条销售记录" << std::endl;
    trans1 = trans2;
    num = 1;
}
std::cout << trans1.isbn() << "共有"
        << num << "条销售记录" << std::endl;
}
else {
    std::cout << "没有数据" << std::endl;
    return -1;
}
return 0;
}
```

**练习 1.24：**输入表示多个 ISBN 的多条销售记录来测试上一个程序，每个 ISBN 的记录应该聚在一起。

#### 【解答】

在网站 <http://www.informit.com/title/0321714113> 上，第 1 章的代码目录中包含了一些数据文件，可以将这些文件重定向到此程序进行测试，也可自己创建销售记录文件进行测试。

**练习 1.25：**借助网站上的 Sales\_item.h 头文件，编译并运行本节给出的书店程序。

#### 【出题思路】

练习编译、运行稍大规模的程序。

#### 【解答】

在网站 <http://www.informit.com/title/0321714113> 上下载 Sales\_item.h 头文件，编译书店程序即可。

# 第 2 章

## 变量和基本类型

### 导读

本章介绍了 C++ 的几种典型数据类型，它们分别是：

- 基本内置类型
- 复合类型
- 自定义数据结构

其中，char、int、long、float、double、bool 是最常见的基本内置类型；引用和指针是两种最重要的复合类型；struct 关键字和 class 关键字则常用于声明用户自定义的数据结构。

除此之外，本章的重要知识点还包括：变量的声明、定义和作用域，顶层 const 和底层 const。C++11 新标准引入了 auto 关键字和 decltype 关键字，利用这两个关键字，程序员就可以不必指定变量的数据类型，而把这项任务交给编译器自动执行。

**练习 2.1：**类型 int、long、long long 和 short 的区别是什么？无符号类型和带符号类型的区别是什么？float 和 double 的区别是什么？

#### 【出题思路】

本题旨在考查 C++ 语言中几种主要算术类型的区别以及符号的表示方法和意义，读者需要重点理解几种算术类型在内存中的存储方式。

#### 【解答】

在 C++ 语言中，int、long、long long 和 short 都属于整型，区别是 C++ 标准规定的尺寸的最小值（即该类型在内存中所占的比特数）不同。其中，short 是短整型，占 16 位；int 是整型，占 16 位；long 和 long long 均为长整型，分

别占 32 位和 64 位。C++ 标准允许不同的编译器赋予这些类型更大的尺寸。某一类型占的比特数不同，它所能表示的数据范围也不一样。

大多数整型都可以划分为无符号类型和带符号类型，在无符号类型中所有比特都用来存储数值，但是仅能表示大于等于 0 的值；带符号类型则可以表示正数、负数或 0。

`float` 和 `double` 分别是单精度浮点数和双精度浮点数，区别主要是在内存中所占的比特数不同，以及默认规定的有效位数不同。

**练习 2.2：**计算按揭贷款时，对于利率、本金和付款分别应选择何种数据类型？说明你的理由。

#### 【出题思路】

本题旨在考查 C++ 语言中选择数据类型的方法。

#### 【解答】

在实际应用中，利率、本金和付款既有可能是整数，也有可能是普通的实数。因此应该选择一种浮点类型来表示。在三种可供选择的浮点类型 `float`、`double` 和 `long double` 中，`double` 和 `float` 的计算代价比较接近且表示范围更广，`long double` 的计算代价则相对较大，一般情况下没有选择的必要。综合以上分析，选择 `double` 是比较恰当的。

**练习 2.3：**读程序写结果。

```
unsigned u=10, u2=42;
std::cout << u2-u << std::endl;
std::cout << u-u2 << std::endl;

int i=10, i2=42;
std::cout << i2-i << std::endl;
std::cout << i-i2 << std::endl;
std::cout << i-u << std::endl;
std::cout << u-i << std::endl;
```

#### 【出题思路】

本题考查的知识点包括：无符号数的计算、带符号数的计算以及混合计算。

#### 【解答】

程序的运行结果是：

```
32
4294967264
32
-32
0
0
```

`u` 和 `u2` 都是无符号整数，因此 `u2-u` 得到了正确的结果 ( $42-10=32$ )；`u-u2` 也能正确计算，但是因为直接计算的结果是 -32，所以在表示为无符号整数时自动加上

了模，在作者的编译环境中 `int` 占 32 位，因此加模的结果是 4294967264。

`i` 和 `i2` 都是带符号整数，因此中间两个式子的结果比较直观， $42-10=32$ ， $10-42=-32$ 。

在最后两个式子中，`u` 和 `i` 分别是无符号整数和带符号整数，计算时编译器先把带符号数转换为无符号数，幸运的是，`i` 本身是一个正数，因此转换后不会出现异常情况，两个式子的计算结果都是 0。

不过需要提醒读者注意的是，一般情况下请不要在同一个表达式中混合使用无符号类型和带符号类型。因为计算前带符号类型会自动转换成无符号类型，当带符号类型取值为负时就会出现异常结果。

**练习 2.4：**编写程序检查你的估计是否正确，如果不正确，请仔细研读本节直到弄明白问题所在。

#### 【出题思路】

本题旨在考查同时含有无符号类型和带符号类型的表达式的计算问题。

#### 【解答】

对于读程序写结果的题目，读者首先应该根据本节知识点独立思考，在脑海中勾勒出题目的考查角度和关键点，进而写出结果并在编程环境中加以验证。

通过练习本题，尤其是观察 `u-u2` 的异常结果，进一步理解无符号类型和带符号类型的区别。读者不妨思考，你是否能用上述 4 个变量组合出其他产生异常结果的式子？应该如何避免这种情况？

**练习 2.5：**指出下述字面值的数据类型并说明每一组内几种字面值的区别：

- (a) 'a', L'a', "a", L"a"
- (b) 10, 10u, 10L, 10uL, 012, 0xC
- (c) 3.14, 3.14f, 3.14L
- (d) 10, 10u, 10., 10e-2

#### 【出题思路】

本题考查的知识点是利用特殊的前缀和后缀指定字面值的类型。

#### 【解答】

(a) 'a' 表示字符 `a`，`L'a'` 表示宽字符型字面值 `a` 且类型是 `wchar_t`，"a" 表示字符串 `a`，`L "a"` 表示宽字符型字符串 `a`。

(b) 10 是一个普通的整数类型字面值，`10u` 表示一个无符号数，`10L` 表示一个长整型数，`10uL` 表示一个无符号长整型数，`012` 是一个八进制数（对应的十进制数是 10），`0xC` 是一个十六进制数（对应的十进制数是 12）。

(c) `3.14` 是一个普通的浮点类型字面值，`3.14f` 表示一个 `float` 类型的单精度浮点数，`3.14L` 表示一个 `long double` 类型的扩展精度浮点数。

(d) 10 是一个整数，`10u` 是一个无符号整数，`10.` 是一个浮点数，`10e-2` 是一

个科学计数法表示的浮点数，大小为  $10*10^{-2}=0.1$ 。

**练习 2.6：**下面两组定义是否有区别，如果有，请叙述之：

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

**【出题思路】**

本题旨在考查十进制数字与八进制数字的表示方法。

**【解答】**

第一组定义是正确的，定义了两个十进制数 9 和 7。

第二组定义是错误的，编译时将报错。因为以 0 开头的数是八进制数，而数字 9 显然超出了八进制数能表示的范围，所以第二组定义无法被编译通过。

**练习 2.7：**下述字面值表示何种含义？它们各自的数据类型是什么？

- (a) "Who goes with F\145rgus?\012"
- (b) 3.14e1L        (c) 1024f        (d) 3.14L

**【出题思路】**

本题考查的知识点是转义字符及利用特殊的后缀指定字面值的类型。

**【解答】**

(a)是一个字符串，包含两个转义字符，其中\145 表示字符 e，\012 表示一个换行符，因此该字符串的输出结果是 Who goes with Fergus?

(b)是一个科学计数法表示的扩展精度浮点数，大小为  $3.14*10^1=31.4$ 。

(c)试图表示一个单精度浮点数，但是该形式在某些编译器中将报错，因为后缀 f 直接跟在了整数 1024 后面；改写成 1024.f 就可以了。

(d)是一个扩展精度浮点数，类型是 long double，大小为 3.14。

**练习 2.8：**请利用转义字符编写一段程序，要求先输出 2M，然后转到新一行。

修改程序使其先输出 2，然后输出制表符，再输出 M，最后转到新一行。

**【出题思路】**

本题旨在考查转义字符的使用。

**【解答】**

```
#include <iostream>

int main ()
{
    std::cout << "2\x4d\012";           // 输出 2M, 然后换行
    std::cout << "2\tM\n";             // 输出 2、制表符、M, 然后换行
    return 0;
}
```

主函数的前两行分别实现了题目中要求的两种输出形式。

其中，字符串"2\x4d\012"先输出字符 2，紧接着利用转义字符\x4d 输出字符 M，最后利用转义字符\012 转到新一行。

字符串"2\tM\n"先输出字符 2，然后利用转义字符\t 输出一个制表符，接着输出字符 M，最后利用转义字符\n 转到新一行。

读者可以发现，输出同一个字符有多种方式可供选择。例如，可以直接输出字符 M，也可以通过转义字符\x4d 输出字符 M；可以用转义字符\012 换行，也可以用转义字符\n 换行。

### 练习 2.9：解释下列定义的含义。对于非法的定义，请说明错在何处并将其改正。

- (a) std::cin >> int input\_value;
- (b) int i = { 3.14 };
- (c) double salary = wage = 9999.99;
- (d) int i = 3.14;

#### 【出题思路】

本题旨在考查变量定义与初始化，其中列表初始化是重点也是难点。

#### 【解答】

(a)是错误的，输入运算符的右侧需要一个明确的变量名称，而非定义变量的语句，改正后的结果是：

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

(b)引发警告，该语句定义了一个整型变量 i，但是试图通过列表初始化的方式把浮点数 3.14 赋值给 i，这样做将造成小数部分丢失，是一种不被建议的窄化操作。

(c)是错误的，该语句试图将 9999.99 分别赋值给 salary 和 wage，但是在声明语句中声明多个变量时需要用逗号将变量名隔开，而不能直接用赋值运算符连接，改正后的结果是：

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

(d)引发警告，该语句定义了一个整型变量 i，但是试图把浮点数 3.14 赋值给 i，这样做将造成小数部分丢失，与(b)一样是不被建议的窄化操作。

### 练习 2.10：下列变量的初值分别是什么？

```
std::string global_str;
int global_int;
int main()
{
    int local_int;
    std::string local_str;
}
```

#### 【出题思路】

本题旨在考查默认初始化的几种不同情况，如全局变量和局部变量的区别、内置类型和复合类型的区别。

#### 【解答】

对于 `string` 类型的变量来说，因为 `string` 类型本身接受无参数的初始化方式，所以不论变量定义在函数内还是函数外都被默认初始化为空串。

对于内置类型 `int` 来说，变量 `global_int` 定义在所有函数体之外，根据 C++ 的规定，`global_int` 默认初始化为 0；而变量 `local_int` 定义在 `main` 函数的内部，将不被初始化，如果程序试图拷贝或输出未初始化的变量，将遇到一个未定义的奇异值。

**练习 2.11：**指出下面的语句是声明还是定义：

- (a) `extern int ix = 1024;`
- (b) `int iy;`
- (c) `extern int iz;`

**【出题思路】**

本题旨在考查变量声明和定义的关系。

**【解答】**

声明与定义的关系是：声明使得名字为程序所知，而定义负责创建与名字关联的实体。(a)定义了变量 `ix`，(b)声明并定义了变量 `iy`，(c)声明了变量 `iz`。

**练习 2.12：**请指出下面的名字哪个是非法的？

- |  |                                  |
|--|----------------------------------|
| (a) <code>int double = 3.14;</code>    | (b) <code>int _;</code>          |
| (c) <code>int catch-22;</code>         | (d) <code>int 1_or_2 = 1;</code> |
| (e) <code>double Double = 3.14;</code> |                                  |

**【出题思路】**

本题旨在考查 C++ 标识符的命名规则。

**【解答】**

(a) 是非法的，因为 `double` 是 C++ 关键字，代表一种数据类型，不能作为变量的名字。

(c) 是非法的，在标识符中只能出现字母、数字和下画线，不能出现符号-，如果改成“`int catch_22;`”就是合法的了。

(d) 是非法的，因为标识符必须以字母或下画线开头，不能以数字开头。

(b) 和 (e) 是合法的命名。

**练习 2.13：**下面程序中 `j` 的值是多少？

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

**【出题思路】**

本题旨在考查全局作用域与局部作用域的关系。

**【解答】**

`j` 的值是 100。C++ 允许内层作用域重新定义外层作用域中已有的名字，在本题中，`int i=42;` 位于外层作用域，但是变量 `i` 在内层作用域被重新定义了，因此真正赋予 `j` 的值是定义在内层作用域中的 `i` 的值，即 100。

**练习 2.14：**下面的程序合法吗？如果合法，它将输出什么？

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

**【出题思路】**

本题旨在考查嵌套作用域中变量的定义和使用。

**【解答】**

该程序是合法的，输出结果是 100 45。

该程序存在嵌套的作用域，其中 `for` 循环之外是外层作用域，`for` 循环内部是内层作用域。首先在外层作用域中定义了 `i` 和 `sum`，但是在 `for` 循环内部 `i` 被重新定义了，因此 `for` 循环实际上是从 `i=0` 循环到了 `i=9`，内层作用域中没有重新定义 `sum`，因此 `sum` 的初始值是 0 并在此基础上依次累加。最后一句输出语句位于外层作用域中，此时在 `for` 循环内部重新定义的 `i` 已经失效，因此实际输出的仍然是外层作用域的 `i`，值为 100；而 `sum` 经由循环累加，值变为了 45。

**练习 2.15：**下面的哪个定义是不合法的？为什么？

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

**【出题思路】**

本题旨在考查引用的含义，应该明确引用与对象的关系。

**【解答】**

(b) 是非法的，引用必须指向一个实际存在的对象而非字面值常量。

(d) 是非法的，因为我们无法令引用重新绑定到另外一个对象，所以引用必须初始化。

(a) 和 (c) 是合法的。

**练习 2.16：**考查下面的所有赋值然后回答：哪些赋值是不合法的？为什么？哪些赋值是合法的？它们执行了什么样的操作？

```
int i = 0, &r1 = i; double d = 0, &r2 = d;
(a) r2 = 3.14159;           (b) r2 = r1;
(c) i = r2;                 (d) r1 = d;
```

**【出题思路】**

本题旨在考查引用的含义及用法，应该明确引用与对象的关系。

**【解答】**

(a)是合法的，为引用赋值实际上是把值赋给了与引用绑定的对象，在这里是把 3.14159 赋给了变量 d。

(b)是合法的，以引用作为初始值实际上是以引用绑定的对象作为初始值，在这里是把 i 的值赋给了变量 d。

(c)是合法的，把 d 的值赋给了变量 i，因为 d 是双精度浮点数而 i 是整数，所以该语句实际上执行了窄化操作。

(d)是合法的，把 d 的值赋给了变量 i，与上一条语句一样执行了窄化操作。

**练习 2.17：** 执行下面的代码段将输出什么结果？

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

**【出题思路】**

本题旨在考查引用的含义及用法，应该明确引用与对象的关系。

**【解答】**

程序的输出结果是 10 10。

引用不是对象，它只是为已经存在的对象起了另外一个名字，因此 ri 实际上是 i 的别名。在上述程序中，首先将 i 赋值为 5，然后把这个值更新为 10。因为 ri 是 i 的引用，所以它们的输出结果是一样的。

**练习 2.18：** 编写代码分别更改指针的值以及指针所指对象的值。**【出题思路】**

本题旨在考查指针的含义，以及指针本身的值与指针所指对象值的区别，重点掌握解引用运算符的使用方法。

**【解答】**

一个满足要求的简单示例如下所示：

```
#include <iostream>

int main ()
{
    int i = 5, j = 10;
    int *p = &i;
    std::cout << p << " " << *p << std::endl;
    p=&j;
    std::cout << p << " " << *p << std::endl;
    *p=20;
    std::cout << p << " " << *p << std::endl;
    j=30;
    std::cout << p << " " << *p << std::endl;
    return 0;
}
```

程序的输出结果是：

```
0x28fef8 5
0x28fef4 10
0x28fef4 20
0x28fef4 30
```

在上述示例中，首先定义了两个整型变量 *i* 和 *j* 以及一个整型指针 *p*，初始情况下令指针 *p* 指向变量 *i*，此时分别输出 *p* 的值（即 *p* 所指对象的内存地址）以及 *p* 所指对象的值，得到 0x28fef8 和 5。

随后依次更改指针的值以及指针所指对象的值。*p=&j;* 改变了指针的值，令指针 *p* 指向另外一个整数对象 *j*。*\*p=20;* 和 *j=30* 是两种更改指针所指对象值的方式，前者显式地更改指针 *p* 所指的内容，后者则通过更改变量 *j* 的值实现同样的目的。

### 练习 2.19：说明指针和引用的主要区别。

#### 【出题思路】

指针和引用同为复合类型，都与内存中实际存在的对象有联系。本题旨在考查二者的主要区别。

#### 【解答】

指针“指向”内存中的某个对象，而引用“绑定到”内存中的某个对象，它们都实现了对其他对象的间接访问，二者的区别主要有两方面：

第一，指针本身就是一个对象，允许对指针赋值和拷贝，而且在指针的生命周期内它可以指向几个不同的对象；引用不是一个对象，无法令引用重新绑定到另外一个对象。

第二，指针无须在定义时赋初值，和其他内置类型一样，在块作用域内定义的指针如果没有被初始化，也将拥有一个不确定的值；引用则必须在定义时赋初值。

### 练习 2.20：请叙述下面这段代码的作用。

```
int i = 42;
int *p1 = &i;
*p1 = *p1 * *p1;
```

#### 【出题思路】

本题旨在考查运算符\*的两种含义：一是指针声明符，二是解引用运算符。

#### 【解答】

这段代码首先定义了一个整型变量 *i* 并设其初值为 42；接着定义了一个整型指针 *p1*，令其指向变量 *i*；最后取出 *p1* 所指的当前值，计算平方后重新赋给 *p1* 所指的变量 *i*。

第二行的\*表示声明一个指针，第三行的\*表示解引用运算，即取出指针 *p1* 所指对象的值。

**练习 2.21:** 请解释下述定义。在这些定义中有非法的吗？如果有，为什么？

```
int i = 0;
(a) double* dp = &i; (b) int *ip = i; (c) int *p = &i;
```

### 【出题思路】

本题旨在考查指针的声明和定义。

### 【解答】

- (a) 是非法的，dp 是一个 double 指针，而 i 是一个 int 变量，类型不匹配。
- (b) 是非法的，不能直接把 int 变量赋给 int 指针，正确的做法是通过取地址运算 `&i` 得到变量 i 在内存中的地址，然后再将该地址赋给指针。
- (c) 是合法的。

**练习 2.22:** 假设 p 是一个 int 型指针，请说明下述代码的含义。

```
if (p)    // ...
if (*p)  // ...
```

### 【出题思路】

本题旨在考查指针的值与指针所指对象的值的区别，巧妙之处是把 p 和 \*p 作为 if 语句的条件，通过判断其是否为真帮助读者加深理解。

### 【解答】

指针 p 作为 if 语句的条件时，实际检验的是指针本身的值，即指针所指的地址值。如果指针指向一个真实存在的变量，则其值必不为 0，此时条件为真；如果指针没有指向任何对象或者是无效指针，则对 p 的使用将引发不可预计的结果。

解引用运算符 \*p 作为 if 语句的条件时，实际检验的是指针所指的对象内容，在上面的示例中是指针 p 所指的 int 值。如果该 int 值为 0，则条件为假；否则，如果该 int 值不为 0，对应条件为真。

一个简单的示例如下所示：

```
#include <iostream>

int main ()
{
    int i = 0;
    int *p1 = nullptr;
    int *p = &i;
    if (p1) // 检验指针的值（即指针所指对象的地址）
        std::cout << "p1 pass" << std::endl;
    if (p)   // 检验指针的值（即指针所指对象的地址）
        std::cout << "p pass" << std::endl;
    if (*p) // 检验指针所指对象的值
        std::cout << "i pass" << std::endl;
    return 0;
}
```

在这段程序中，p 和 p1 是两个整型指针，其中 p1 被定义为空指针(nullptr)，p 则指向整数 i。在 3 个判断条件中，p1 指向为空，意即指向地址为 0，条件不满

足；`p` 指向 `i`，在内存中有一个实际的地址值且不为 0，因此该条件满足；`*p` 表示 `p` 所指对象的内容，即整数 `i` 的值，因为程序开始处 `i` 被赋予了初值 0，所以这个条件不满足。

综合以上分析，程序的输出结果是 `p pass.`

**练习 2.23：**给定指针 `p`，你能知道它是否指向了一个合法的对象吗？如果能，叙述判断的思路；如果不能，说明原因。

#### 【出题思路】

本题旨在考查指针初始化，读者应该熟悉并掌握 C++11 的新语法特征 `nullptr`。

#### 【解答】

在 C++ 程序中，应该尽量初始化所有指针，并且尽可能等定义了对象之后再定义指向它的指针。如果实在不清楚指针应该指向何处，就把它初始化为 `nullptr` 或者 0，这样程序就能检测并知道它有没有指向一个具体的对象了。其中，`nullptr` 是 C++11 新标准刚刚引入的一个特殊字面值，它可以转换成任意其他的指针类型。在此前提下，判断 `p` 是否指向合法的对象，只需把 `p` 作为 `if` 语句的条件即可，如果 `p` 的值是 `nullptr`，则条件为假；反之，条件为真。

如果不注意初始化所有指针而贸然判断指针的值，则有可能引发不可预知的结果。一种处理的办法是把 `if(p)` 置于 `try` 结构中，当程序块顺利执行时，表示 `p` 指向了合法的对象；当程序块出错跳转到 `catch` 语句时，表示 `p` 没有指向合法的对象。

**练习 2.24：**在下面这段代码中为什么 `p` 合法而 `lp` 非法？

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

#### 【出题思路】

本题主要考查 `void*` 指针的含义和用法。

#### 【解答】

`p` 是合法的，因为 `void*` 是一种特殊的指针类型，可用于存放任意对象的地址。

`lp` 是非法的，因为 `lp` 是一个长整型指针，而 `i` 只是一个普通整型数，二者的类型不匹配。

**练习 2.25：**说明下列变量的类型和值。

(a) `int* ip, i, &r = i;` (b) `int i, *ip = 0;` (c) `int* ip, ip2;`

#### 【出题思路】

本题旨在考查指针和引用这两种复合类型的声明。

#### 【解答】

(a) `ip` 是一个整型指针，指向一个整型数，它的值是所指整型数在内存中的地址；`i` 是一个整型数；`r` 是一个引用，它绑定了 `i`，可以看作是 `i` 的别名，`r` 的值就是 `i` 的值。

(b)i 是一个整型数; ip 是一个整型指针, 但是它不指向任何具体的对象, 它的值被初始化为 0。

(c)ip 是一个整型指针, 指向一个整型数, 它的值是所指整型数在内存中的地址; ip2 是一个整型数。

**练习 2.26:** 下面哪些句子是合法的? 如果有不合法的句子, 请说明为什么?

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

**【出题思路】**

本题旨在考查 `const` 限定符的用法, 尤其是 `const` 对象的定义、初始化和运算。

**【解答】**

本题的所有语句应该被看作是顺序执行的, 即形如:

```
const int buf;
int cnt = 0;
const int sz = cnt;
++cnt;
++sz;
```

(a) 是非法的, `const` 对象一旦创建后其值就不能改变, 所以 `const` 对象必须初始化。该句应修改为 `const int buf = 10`。

(b) 和 (c) 是合法的。

(d) 是非法的, `sz` 是一个 `const` 对象, 其值不能被改变, 当然不能执行自增操作。

**练习 2.27:** 下面的哪些初始化是合法的? 请说明原因。

- |                               |                                |
|-------------------------------|--------------------------------|
| (a) int i = -1, &r = 0;       | (b) int *const p2 = &i2;       |
| (c) const int i = -1, &r = 0; | (d) const int *const p3 = &i2; |
| (e) const int *p1 = &i2;      | (f) const int &const r2;       |
| (g) const int i2 = i, &r = i; |                                |

**【出题思路】**

本题旨在考查常量引用、常量指针和指向常量的指针的初始化方法。

**【解答】**

(a) 是非法的, 非常量引用 `r` 不能引用字面值常量 0。

(b) 是合法的, `p2` 是一个常量指针, `p2` 的值永不改变, 即 `p2` 永远指向变量 `i2`。

(c) 是合法的, `i` 是一个常量, `r` 是一个常量引用, 此时 `r` 可以绑定到字面值常量 0。

(d) 是合法的, `p3` 是一个常量指针, `p3` 的值永不改变, 即 `p3` 永远指向变量 `i2`; 同时 `p3` 指向的是常量, 即我们不能通过 `p3` 改变所指对象的值。

(e) 是合法的, `p1` 指向一个常量, 即我们不能通过 `p1` 改变所指对象的值。

(f) 是非法的, 引用本身不是对象, 因此不能让引用恒定不变。

(g) 是合法的, `i2` 是一个常量, `r` 是一个常量引用。

**练习 2.28:** 说明下面的这些定义是什么意思，挑出其中不合法的。

- |                            |                          |
|----------------------------|--------------------------|
| (a) int i, *const cp;      | (b) int *p1, *const p2;  |
| (c) const int ic, &r = ic; | (d) const int *const p3; |
| (e) const int *p;          |                          |

#### 【出题思路】

本题旨在考查常量引用、常量指针和指向常量的指针的定义及区别。

#### 【解答】

- (a) 是非法的，cp 是一个常量指针，因其值不能被改变，所以必须初始化。
- (b) 是非法的，cp2 是一个常量指针，因其值不能被改变，所以必须初始化。
- (c) 是非法的，ic 是一个常量，因其值不能被改变，所以必须初始化。
- (d) 是非法的，p3 是一个常量指针，因其值不能被改变，所以必须初始化；同时 p3 指向的是常量，即我们不能通过 p3 改变所指对象的值。
- (e) 是合法的，但是 p 没有指向任何实际的对象。

**练习 2.29:** 假设已有上一个练习中定义的那些变量，则下面的哪些语句是合法的？请说明原因。

- |               |               |
|---------------|---------------|
| (a) i = ic;   | (b) p1 = p3;  |
| (c) p1 = &ic; | (d) p3 = &ic; |
| (e) p2 = p1;  | (f) ic = *p3; |

#### 【出题思路】

本题旨在考查常量引用、常量指针和指向常量的指针的赋值方法。

#### 【解答】

- (a) 是合法的，常量 ic 的值赋给了非常量 i。
- (b) 是非法的，普通指针 p1 指向了一个常量，从语法上说，p1 的值可以随意改变，显然是不合理的。
- (c) 是非法的，普通指针 p1 指向了一个常量，错误情况与上一条类似。
- (d) 是非法的，p3 是一个常量指针，不能被赋值。
- (e) 是非法的，p2 是一个常量指针，不能被赋值。
- (f) 是非法的，ic 是一个常量，不能被赋值。

**练习 2.30:** 对于下面的这些语句，请说明对象被声明成了顶层 const 还是底层 const？

```
const int v2 = 0;    int v1 = v2;
int *p1 = &v1, &r1 = v1;
const int *p2 = &v2, *const p3 = &i, &r2 = v2;
```

#### 【出题思路】

本题旨在考查顶层 const 和底层 const 的区别，读者应明确顶层 const 表示任意的对象是常量，而底层 const 与指针和引用等复合类型的基本类型部分有关。

#### 【解答】

v2 和 p3 是顶层 const，分别表示一个整型常量和一个整型常量指针；p2 和

r2 是底层 const，分别表示它们所指（所引用）的对象是常量。

**练习 2.31：**假设已有上一个练习中所做的那些声明，则下面的哪些语句是合法的？请说明顶层 const 和底层 const 在每个例子中有何体现。

```
r1 = v2;
p1 = p2; p2 = p1;
p1 = p3; p2 = p3;
```

#### 【出题思路】

本题旨在考查顶层 const 和底层 const 对于拷贝操作的影响。

#### 【解答】

在执行拷贝操作时，顶层 const 和底层 const 区别明显。其中，顶层 const 不受影响，这是因为拷贝操作并不会改变被拷贝对象的值。底层 const 的限制则不容忽视，拷入和拷出的对象必须具有相同的底层 const 资格，或者两个对象的数据类型必须能够转换。一般来说，非常量可以转换成常量，反之则不行。

`r1=v2;` 是合法的，`r1` 是一个非常量引用，`v2` 是一个常量（顶层 const），把 `v2` 的值拷贝给 `r1` 不会对 `v2` 有任何影响。

`p1=p2;` 是非法的，`p1` 是普通指针，指向的对象可以是任意值，`p2` 是指向常量的指针（底层 const），令 `p1` 指向 `p2` 所指的内容，有可能错误地改变常量的值。

`p2=p1;` 是合法的，与上一条语句相反，`p2` 可以指向一个非常量，只不过我们不会通过 `p2` 更改它所指的值。

`p1=p3;` 是非法的，`p3` 包含底层 const 定义（`p3` 所指的对象是常量），不能把 `p3` 的值赋给普通指针。

`p2=p3;` 是合法的，`p2` 和 `p3` 包含相同的底层 const，`p3` 的顶层 const 则可以忽略不计。

**练习 2.32：**下面的代码是否合法？如果非法，请设法将其修改正确。

```
int null = 0, *p = null;
```

#### 【出题思路】

本题旨在考查指针的使用。

#### 【解答】

上述代码是非法的，`null` 是一个 int 变量，`p` 是一个 int 指针，二者不能直接绑定。仅从语法角度来说，可以将代码修改为：

```
int null = 0, *p = &null;
```

显然，这种改法与代码的原意不一定相符。另一种改法是使用 `nullptr`：

```
int null = 0, *p = nullptr;
```

**练习 2.33：**利用本节定义的变量，判断下列语句的运行结果。

```
a = 42; b = 42; c = 42;
```

```
d = 42; e = 42; g = 42;
```

### 【出题思路】

本题旨在考查 `auto` 说明符与复合类型、常量混合使用时的各种情形。首先，使用引用其实是使用引用的对象，所以当引用被用作初始值时，真正参与初始化的其实是引用对象的值，编译器以引用对象的类型作为 `auto` 的推断类型。其次，`auto` 一般会忽略掉顶层 `const`，同时保留底层 `const`。

### 【解答】

前3条赋值语句是合法的，原因如下：

`r` 是 `i` 的别名，而 `i` 是一个整数，所以 `a` 的类型推断结果是一个整数；`ci` 是一个整型常量，在类型推断时顶层 `const` 被忽略掉了，所以 `b` 是一个整数；`cr` 是 `ci` 的别名，而 `ci` 是一个整型常量，所以 `c` 的类型推断结果是一个整数。因为 `a`、`b`、`c` 都是整数，所以为其赋值 42 是合法的。

后3条赋值语句是非法的，原因如下：

`i` 是一个整数，`&i` 是 `i` 的地址，所以 `d` 的类型推断结果是一个整型指针；`ci` 是一个整型常量，`&ci` 是一个整型常量的地址，所以 `e` 的类型推断结果是一个指向整型常量的指针；`ci` 是一个整型常量，所以 `g` 的类型推断结果是一个整型常量引用。因为 `d` 和 `e` 都是指针，所以不能直接用字面值常量为其赋值；`g` 绑定到了整型常量，所以不能修改它的值。

**练习 2.34：** 基于上一个练习中的变量和语句编写一段程序，输出赋值前后变量的内容，你刚才的推断正确吗？如果不对，请反复研读本节的示例直到你明白错在何处为止。

### 【出题思路】

本题旨在考查 `auto` 说明符与复合类型、常量混合使用时的各种情形。

### 【解答】

基于上一个练习中的变量和语句编写的程序如下所示：

```
#include <iostream>

int main()
{
    int i = 0, &r = i;
    auto a = r;          // a 是一个整数 (r 是 i 的别名，而 i 是个整数)
    const int ci = i, &cr = ci;
    auto b = ci;         // b 是一个整数 (ci 的顶层 const 特性被忽略掉了)
    auto c = cr;         // c 是一个整数 (cr 是 ci 的别名，ci 本身是一个顶层 const)
    auto d = &i;          // d 是一个整型指针 (整数的地址就是指向整数的指针)
    auto e = &ci;         // e 是一个指向整型常量的指针(对常量对象取地址是一种底层 const)
    auto &g = ci;         // g 是一个整型常量引用，绑定到 ci
    std::cout << a << " " << b << " " << c << " " << d << " " << e <<
    " " << g << std::endl;
```

```

a = 42;
b = 42;
c = 42;
d = 42;           // 错误: d 是一个指针, 赋值非法
e = 42;           // 错误: e 是一个指针, 赋值非法
g = 42;           // 错误: g 是一个常量引用, 赋值非法
std::cout << a << " " << b << " " << c << " " << d << " " << e <<
" " << g << std::endl;

return 0;
}

```

程序的输出结果是：

```

0    0    0    0x28fedc    0x28fed8    0
42   42   42   0x28fedc    0x28fed8    0

```

**练习 2.35：**判断下列定义推断出的类型是什么，然后编写程序进行验证。

```

const int i = 42;
auto j = i; const auto &k = i; auto *p = &i;
const auto j2 = i, &k2 = i;

```

### 【出题思路】

本题旨在考查 `auto` 说明符与复合类型、常量混合使用时的各种情形。

### 【解答】

由题意可知，`i` 是一个整型常量，`j` 的类型推断结果是整数，`k` 的类型推断结果是整型常量，`p` 的类型推断结果是指向整型常量的指针，`j2` 的类型推断结果是整数，`k2` 的类型推断结果是整数。

用于验证的程序是：

```

#include <iostream>
#include<typeinfo>

int main ()
{
    const int i = 42;
    auto j = i;
    const auto &k = i;
    auto *p = &i;
    const auto j2 = i, &k2 = i;
    std::cout << typeid(i).name() << std::endl;
    std::cout << typeid(j).name() << std::endl;
    std::cout << typeid(k).name() << std::endl;
    std::cout << typeid(p).name() << std::endl;
    std::cout << typeid(j2).name() << std::endl;
    std::cout << typeid(k2).name() << std::endl;

    return 0;
}

```

**练习 2.36:** 关于下面的代码, 请指出每一个变量的类型以及程序结束时它们各自的值。

```
int a = 3, b = 4;
decltype(a) c = a;
decltype((b)) d = a;
++c;
++d;
```

### 【出题思路】

本题旨在考查 decltype 与引用的关系。对于 decltype 所用的表达式来说, 如果变量名加上一对括号, 得到的类型与不加括号时会有不同。具体来说, 如果 decltype 使用的是一个不加括号的变量, 则得到的结果就是该变量的类型; 如果给变量加上了一层或多层括号, 编译器就会把它当成是一个表达式, 从而推断得到引用类型。

### 【解答】

在本题的程序中, 初始情况下 a 的值是 3、b 的值是 4。decltype(a) c=a; 使用的是一个不加括号的变量, 因此 c 的类型就是 a 的类型, 即该语句等同于 int c=a;, 此时 c 是一个新整型变量, 值为 3。decltype((b)) d=a; 使用的是一个加了括号的变量, 因此 d 的类型是引用, 即该语句等同于 int &d=a;, 此时 d 是变量 a 的别名。

执行 ++c; ++d; 时, 变量 c 的值自增为 4, 因为 d 是 a 的别名, 所以 d 自增 1 意味着 a 的值变成了 4。当程序结束时, a、b、c、d 的值都是 4。

**练习 2.37:** 赋值是会产生引用的一类典型表达式, 引用的类型就是左值的类型。也就是说, 如果 i 是 int, 则表达式 i=x 的类型是 int&。根据这一特点, 请指出下面的代码中每一个变量的类型和值。

```
int a = 3, b = 4;
decltype(a) c = a;
decltype(a = b) d = a;
```

### 【出题思路】

decltype 的参数既可以是普通变量, 也可以是一个表达式。当参数是普通变量时, 推断出的类型就是该变量的类型; 当参数是表达式时, 推断出的类型是引用。

### 【解答】

根据 decltype 的上述性质可知, c 的类型是 int, 值为 3; 表达式 a=b 作为 decltype 的参数, 编译器分析表达式并得到它的类型作为 d 的推断类型, 但是不实际计算该表达式, 所以 a 的值不发生改变, 仍然是 3; d 的类型是 int&, d 是 a 的别名, 值是 3; b 的值一直没有发生改变, 为 4。

**练习 2.38:** 说明由 decltype 指定类型和由 auto 指定类型有何区别。请举出一个例子, decltype 指定的类型与 auto 指定的类型一样; 再举一个例子, decltype 指定的类型与 auto 指定的类型不一样。

**【出题思路】**

`auto` 和 `decltype` 是两种类型推断的方式，本题旨在考查二者的区别和联系。

**【解答】**

`auto` 和 `decltype` 的区别主要有三个方面：

第一，`auto` 类型说明符用编译器计算变量的初始值来推断其类型，而 `decltype` 虽然也让编译器分析表达式并得到它的类型，但是不实际计算表达式的值。

第二，编译器推断出来的 `auto` 类型有时候和初始值的类型并不完全一样，编译器会适当地改变结果类型使其更符合初始化规则。例如，`auto` 一般会忽略掉顶层 `const`，而把底层 `const` 保留下。与之相反，`decltype` 会保留变量的顶层 `const`。

第三，与 `auto` 不同，`decltype` 的结果类型与表达式形式密切相关，如果变量名加上了一对括号，则得到的类型与不加括号时会有不同。如果 `decltype` 使用的是一个不加括号的变量，则得到的结果就是该变量的类型；如果给变量加上了一层或多层括号，则编译器将推断得到引用类型。

一个用以说明的示例如下所示：

```
#include <iostream>
#include <typeinfo>

int main ()
{
    int a = 3;
    auto c1 = a;
    decltype(a) c2 = a;
    decltype((a)) c3 = a;

    const int d = 5;
    auto f1 = d;
    decltype(d) f2 = d;

    std::cout << typeid(c1).name() << std::endl;
    std::cout << typeid(c2).name() << std::endl;
    std::cout << typeid(c3).name() << std::endl;
    std::cout << typeid(f1).name() << std::endl;
    std::cout << typeid(f2).name() << std::endl;

    c1++;
    c2++;
    c3++;
    f1++;
    f2++;           // 错误：f2 是整型常量，不能执行自增操作
    std::cout << a << " " << c1 << " " << c2 << " " << c3 << " " << f1
    << " " << f2 << std::endl;
    return 0;
}
```

对于第一组类型推断来说，`a` 是一个非常量整数，`c1` 的推断结果是整数，`c2` 的推断结果也是整数，`c3` 的推断结果由于变量 `a` 额外加了一对括号所以是整数引用。`c1`、`c2`、`c3` 依次执行自增操作，因为 `c3` 是变量 `a` 的别名，所以 `c3` 自增等同于 `a`

自增，最终 `a`、`c1`、`c2`、`c3` 的值都变为 4。

对于第二组类型推断来说，`d` 是一个常量整数，含有顶层 `const`，使用 `auto` 推断类型自动忽略掉顶层 `const`，因此 `f1` 的推断结果是整数；`decltype` 则保留顶层 `const`，所以 `f2` 的推断结果是整数常量。`f1` 可以正常执行自增操作，而常量 `f2` 的值不能被改变，所以无法自增。

**练习 2.39：**编译下面的程序观察其运行结果，注意，如果忘记写类定义体后面的分号会发生什么情况？记录下相关信息，以后可能会有用。

```
struct Foo { /* 此处为空 */ }      // 注意：没有分号
int main()
{
    return 0;
}
```

#### 【出题思路】

本题旨在考查类定义的语法规范，尤其要注意类体结束之后的分号必不可少。

#### 【解答】

该程序无法编译通过，原因是缺少了一个分号。因为类体后面可以紧跟变量名以示对该类型对象的定义，所以在类体右侧表示结束的花括号之后必须写一个分号。稍作修改，该程序就可以编译通过了。

```
struct Foo { /* 此处为空 */ };
int main ()
{
    return 0;
}
```

**练习 2.40：**根据自己的理解写出 `Sales_data` 类，最好与书中的例子有所区别。

#### 【出题思路】

类的设计源于实际应用，设计 `Sales_data` 类的关键是理解在销售过程中应该包含哪些数据元素，同时为每个元素设定合理的数据类型。

#### 【解答】

原书中的程序包含 3 个数据成员，分别是 `bookNo`（书籍编号）、`units_sold`（销售量）、`revenue`（销售收入），新设计的 `Sales_data` 类细化了销售收入的计算方式，在保留 `bookNo` 和 `units_sold` 的基础上，新增了 `sellingprice`（零售价、原价）、`saleprice`（实售价、折扣价）、`discount`（折扣），其中 `discount=saleprice/sellingprice`。

```
struct Sales_data {
    std::string bookNo;           // 书籍编号
    unsigned units_sold = 0;       // 销售量
    double sellingprice = 0.0;     // 零售价
    double saleprice = 0.0;        // 实售价
    double discount = 0.0;         // 折扣
```

```
    double discount = 0.0          // 折扣
};


```

**练习 2.41:** 使用你自己的 Sales\_data 类重写 1.5.1 节(第 20 页)、1.5.2 节(第 21 页)和 1.6 节(第 22 页)的练习。眼下先把 Sales\_data 类的定义和 main 函数放在同一个文件里。

### 【出题思路】

本题旨在考查如何利用自定义类创建对象并执行基本操作。

### 【解答】

```
#include <iostream>
#include <string>

using namespace std;

class Sales_data {
// 友元函数
friend std::istream& operator >> (std::istream&, Sales_data&);
// 友元函数
friend std::ostream& operator << (std::ostream&, const Sales_data&);
// 友元函数
friend bool operator < (const Sales_data&, const Sales_data&);
// 友元函数
friend bool operator == (const Sales_data&, const Sales_data&);
public:      // 构造函数的 3 种形式
    Sales_data() = default;
    Sales_data(const std::string &book): bookNo(book) { }
    Sales_data(std::istream &is) { is >> *this; }
public:
    Sales_data& operator += (const Sales_data&);
    std::string isbn() const { return bookNo; }
private:
    std::string bookNo;           // 书籍编号, 隐式初始化为空串
    unsigned units_sold = 0;     // 销售量, 显式初始化为 0
    double sellingprice = 0.0;   // 原始价格, 显式初始化为 0.0
    double saleprice = 0.0;      // 实售价格, 显式初始化为 0.0
    double discount = 0.0;       // 折扣, 显式初始化为 0.0
};

inline bool compareIsbn(const Sales_data &lhs, const Sales_data &rhs)
{ return lhs.isbn() == rhs.isbn(); }

Sales_data operator + (const Sales_data&, const Sales_data&);

inline bool operator == (const Sales_data &lhs, const Sales_data &rhs)
{
    return lhs.units_sold == rhs.units_sold &&
           lhs.sellingprice == rhs.sellingprice &&
```

```

        lhs.saleprice == rhs.saleprice &&
        lhs.isbn() == rhs.isbn();
    }

inline bool operator != (const Sales_data &lhs, const Sales_data &rhs)
{
    return !(lhs == rhs); // 基于运算符==给出!=的定义
}

Sales_data& Sales_data::operator += (const Sales_data& rhs)
{
    units_sold += rhs.units_sold;
    saleprice = (rhs.saleprice * rhs.units_sold + saleprice * units_sold)
        / (rhs.units_sold + units_sold);
    if(sellingprice != 0)
        discount = saleprice / sellingprice;
    return *this;
}

Sales_data operator + (const Sales_data& lhs, const Sales_data& rhs)
{
    Sales_data ret(lhs); // 把 lhs 的内容拷贝到临时变量 ret 中, 这种做法便于运算
    ret += rhs;           // 把 rhs 的内容加入其中
    return ret;           // 返回 ret
}

std::istream& operator>>(std::istream& in, Sales_data& s)
{
    in >> s.bookNo >> s.units_sold >> s.sellingprice >> s.saleprice;
    if (in && s.sellingprice != 0)
        s.discount = s.saleprice / s.sellingprice;
    else
        s = Sales_data(); // 输入错误, 重置输入的数据
    return in;
}

std::ostream& operator << (std::ostream& out, const Sales_data& s)
{
    out << s.isbn() << " " << s.units_sold << " "
        << s.sellingprice << " " << s.saleprice << " " << s.discount;
    return out;
}

int main()
{
    Sales_data book;
    std::cout << "请输入销售记录: " << std::endl;
    while (std::cin >> book) {
        std::cout << " ISBN、售出本数、原始价格、实售价格、折扣为" << book <<
        std::endl;
    }
    Sales_data transl, trans2;
}

```

```

std::cout << "请输入两条 ISBN 相同的销售记录: " << std::endl;
std::cin >> transl >> trans2;
if (compareIsbn(transl, trans2))
    std::cout << "汇总信息: ISBN、售出本数、原始价格、实售价格、折扣为 "
    << transl + trans2 << std::endl;
else
    std::cout << "两条销售记录的 ISBN 不同" << std::endl;

Sales_data total, trans;
std::cout << "请输入几条 ISBN 相同的销售记录: " << std::endl;
if (std::cin >> total) {
    while (std::cin >> trans)
        if (compareIsbn(total, trans)) // ISBN 相同
            total = total + trans;
        else { // ISBN 不同
            std::cout << "当前书籍 ISBN 不同" << std::endl;
            break;
        }
    std::cout << "有效汇总信息: ISBN、售出本数、原始价格、实售价格、折扣为"
    << total << std::endl;
}
else {
    std::cout << "没有数据" << std::endl;
    return -1;
}

int num = 1;      // 记录当前书籍的销售记录总数
std::cout << "请输入若干销售记录: " << std::endl;
if (std::cin >> transl) {
    while (std::cin >> trans2)
        if (compareIsbn(transl, trans2)) // ISBN 相同
            num++;
        else { // ISBN 不同
            std::cout << transl.isbn() << "共有"
            << num << "条销售记录" << std::endl;
            transl = trans2;
            num = 1;
        }
    std::cout << transl.isbn() << "共有"
    << num << "条销售记录" << std::endl;
}
else {
    std::cout << "没有数据" << std::endl;
    return -1;
}

return 0;
}

```

**练习 2.42：**根据你自己的理解重写一个 `Sales_data.h` 头文件，并以此为基础重做 2.6.2 节（第 67 页）的练习。

### 【出题思路】

本题旨在考查自定义头文件并基于头文件编写程序的方法。

### 【解答】

将上题答案中类的定义放置于 `Sales_data.h` 头文件中，具体的使用方法与第 1 章练习 1.20~练习 1.25 类似。

```
Sales_data.h 头文件的内容是：
#ifndef SALES_DATA_H_INCLUDED
#define SALES_DATA_H_INCLUDED

#include <iostream>
#include <string>

class Sales_data {
    // 友元函数
    friend std::istream& operator >> (std::istream&, Sales_data&);
    // 友元函数
    friend std::ostream& operator << (std::ostream&, const Sales_data&);
    // 友元函数
    friend bool operator < (const Sales_data&, const Sales_data&);
    // 友元函数
    friend bool operator == (const Sales_data&, const Sales_data&);

public:    // 构造函数的 3 种形式
    Sales_data() = default;
    Sales_data(const std::string &book) : bookNo(book) { }
    Sales_data(std::istream &is) { is >> *this; }

public:
    Sales_data& operator += (const Sales_data&);
    std::string isbn() const { return bookNo; }

private:
    std::string bookNo;           // 书籍编号，隐式初始化为空串
    unsigned units_sold = 0;      // 销售量，显式初始化为 0
    double sellingprice = 0.0;    // 原始价格，显式初始化为 0.0
    double saleprice = 0.0;       // 实售价格，显式初始化为 0.0
    double discount = 0.0;        // 折扣，显式初始化为 0.0
};

inline bool compareIsbn(const Sales_data &lhs, const Sales_data &rhs)
{ return lhs.isbn() == rhs.isbn(); }

Sales_data operator + (const Sales_data&, const Sales_data&);

inline bool operator == (const Sales_data &lhs, const Sales_data &rhs)
{
    return lhs.units_sold == rhs.units_sold &&
           lhs.sellingprice == rhs.sellingprice &&
           lhs.saleprice == rhs.saleprice &&
```

```

        lhs.isbn() == rhs.isbn();
    }

inline bool operator != (const Sales_data &lhs, const Sales_data &rhs)
{
    return !(lhs == rhs);      // 基于运算符==给出!=的定义
}

Sales_data& Sales_data::operator += (const Sales_data& rhs)
{
    units_sold += rhs.units_sold;
    saleprice = (rhs.saleprice * rhs.units_sold + saleprice * units_sold)
        / (rhs.units_sold + units_sold);
    if(sellingprice != 0)
        discount = saleprice / sellingprice;
    return *this;
}
Sales_data operator + (const Sales_data& lhs, const Sales_data& rhs)
{
    Sales_data ret(lhs); // 把lhs 的内容拷贝到临时变量ret 中, 这种做法便于运算
    ret += rhs;          // 把rhs 的内容加入其中
    return ret;          // 返回ret
}

std::istream& operator>>(std::istream& in, Sales_data& s)
{
    in >> s.bookNo >> s.units_sold >> s.sellingprice >> s.saleprice;
    if (in && s.sellingprice != 0)
        s.discount = s.saleprice / s.sellingprice;
    else
        s = Sales_data();      // 输入错误, 重置输入的数据
    return in;
}

std::ostream& operator << (std::ostream& out, const Sales_data& s)
{
    out << s.isbn() << " " << s.units_sold << " "
        << s.sellingprice << " " << s.saleprice << " " << s.discount;
    return out;
}

#endif // SALES_DATA_H_INCLUDED

```

main.cpp 源文件内容是:

```

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

int main()
{
    Sales_data book;
    std::cout << "请输入销售记录: " << std::endl;
    while (std::cin >> book) {
        std::cout << " ISBN、售出本数、原始价格、实售价格、折扣为" << book <<

```

```

    std::endl;
}

Sales_data trans1, trans2;
std::cout << "请输入两条 ISBN 相同的销售记录: " << std::endl;
std::cin >> trans1 >> trans2;
if (compareIsbn(trans1, trans2))
    std::cout << "汇总信息: ISBN、售出本数、原始价格、实售价格、折扣为 "
    << trans1 + trans2 << std::endl;
else
    std::cout << "两条销售记录的 ISBN 不同" << std::endl;

Sales_data total, trans;
std::cout << "请输入几条 ISBN 相同的销售记录: " << std::endl;
if (std::cin >> total) {
    while (std::cin >> trans)
        if (compareIsbn(total, trans)) // ISBN 相同
            total = total + trans;
        else { // ISBN 不同
            std::cout << "当前书籍 ISBN 不同" << std::endl;
            break;
        }
    std::cout << "有效汇总信息: ISBN、售出本数、原始价格、实售价格、折扣为"
    << total << std::endl;
}
else {
    std::cout << "没有数据" << std::endl;
    return -1;
}

int num = 1; // 记录当前书籍的销售记录总数
std::cout << "请输入若干销售记录: " << std::endl;
if (std::cin >> trans1) {
    while (std::cin >> trans2)
        if (compareIsbn(trans1, trans2)) // ISBN 相同
            num++;
        else { // ISBN 不同
            std::cout << trans1.isbn() << "共有"
            << num << "条销售记录" << std::endl;
            trans1 = trans2;
            num = 1;
        }
    std::cout << trans1.isbn() << "共有"
    << num << "条销售记录" << std::endl;
}
else {
    std::cout << "没有数据" << std::endl;
    return -1;
}

return 0;
}

```

# 第3章

# 字符串、向量和数组

## 导读

本章介绍了两种重要的标准库类型，它们分别是 `string` 和 `vector`。

`string` 表示可变长字符串，它的初始化方式分为两种：直接初始化和拷贝初始化。如果要处理 `string` 中的单个字符，我们可以使用 C++11 新定义的范围 `for` 语句，也可以使用下标运算符执行随机访问。

`vector` 表示对象的集合，其中所有对象的类型相同。如果要向 `vector` 对象中添加元素，必须使用 `push_back()` 函数，不允许使用下标形式添加元素。

除此之外，本章还介绍了迭代器和数组。

迭代器提供与指针功能类似的间接访问操作，迭代器可以执行解引用、与整数相加、比较、两个整数相减等操作，但是不能直接把两个迭代器相加。

与 `vector` 一样，数组也表示同类型对象的集合，但是 `vector` 的长度不固定，而数组的容量是固定不变的。在 C++11 新标准下，我们可以使用与迭代器类似的 `begin` 和 `end` 函数确定数组的边界。

**练习 3.1：** 使用恰当的 `using` 声明重做 1.4.1 节（第 11 页）和 2.6.2 节（第 67 页）的练习。

### 【出题思路】

使用作用域操作符指定名字作用域的方式比较烦琐，使用 `using` 声明是一种更好的选择。本题旨在引导读者去除程序中的作用域操作符，而以 `using` 声明作为替代，从而使得程序更加简洁直观。

### 【解答】

使用 `using` 声明重做 1.4.1 节中练习 1.9 的程序如下所示：

```
#include <iostream>
// 使用 using 声明使得 cout 和 endl 在程序中可见
using std::cout;
using std::endl;

int main()
{
    int sum = 0;
    int i = 50;
    while (i <= 100) {
        sum += i;
        i++;
    }
    cout << "50 到 100 之间的整数之和为" << sum << endl;
    return 0;
}
```

使用 using 声明重做 1.4.1 节练习 1.10 的程序如下所示：

```
#include <iostream>
// 使用 using 声明使得 cout 和 endl 在程序中可见
using std::cout;
using std::endl;

int main()
{
    int i = 10;
    while (i >= 0) {
        cout << i << " ";
        i--;
    }
    cout << endl;
    return 0;
}
```

使用 using 声明重做 1.4.1 节练习 1.11 的程序如下所示：

```
#include <iostream>
// 使用 using 声明使得 cin、cout 和 endl 在程序中可见
using std::cin;
using std::cout;
using std::endl;

int main()
{
    cout << "请输入两个数";
    cout << endl;
    int v1, v2;
    cin >> v1 >> v2;
    if (v1 > v2)           // 由大至小打印
        while (v1 >= v2) {
            cout << v1 << " ";
            v1--;
        }
    else                   // 由小至大打印
```

```

    while (v1 <= v2) {
        cout << v1 << " ";
        v1++;
    }
    cout << endl;
    return 0;
}

```

使用 using 声明重做 2.6.2 节练习的程序如下所示：

```

#include <iostream>
#include <string>
// 使用 using 声明使得以下的名字在程序中可见
using std::cin;
using std::cout;
using std::endl;
using std::istream;
using std::ostream;
using std::string;

class Sales_data {
friend istream& operator >> (istream&, Sales_data&);           // 友元函数
friend ostream& operator << (ostream&, const Sales_data&); // 友元函数
// 友元函数
friend bool operator < (const Sales_data&, const Sales_data&);
// 友元函数
friend bool operator == (const Sales_data&, const Sales_data&);
public:      // 构造函数的 3 种形式
    Sales_data() = default;
    Sales_data(const string &book): bookNo(book) { }
    Sales_data(istream &is) { is >> *this; }
public:
    Sales_data& operator += (const Sales_data&);
    string isbn() const { return bookNo; }
private:
    string bookNo;           // 书籍编号，隐式初始化为空串
    unsigned units_sold = 0;  // 销售量，显式初始化为 0
    double sellingprice = 0.0; // 原始价格，显式初始化为 0.0
    double saleprice = 0.0;   // 实售价格，显式初始化为 0.0
    double discount = 0.0;    // 折扣，显式初始化为 0.0
};

inline bool compareIsbn(const Sales_data &lhs, const Sales_data &rhs)
{ return lhs.isbn() == rhs.isbn(); }

Sales_data operator + (const Sales_data&, const Sales_data&);

inline bool operator == (const Sales_data &lhs, const Sales_data &rhs)
{
    return lhs.units_sold == rhs.units_sold &&
           lhs.sellingprice == rhs.sellingprice &&
           lhs.saleprice == rhs.saleprice &&

```

```

        lhs.isbn() == rhs.isbn();
    }

inline bool operator != (const Sales_data &lhs, const Sales_data &rhs)
{
    return !(lhs == rhs); // 基于运算符==给出!=的定义
}

Sales_data& Sales_data::operator += (const Sales_data& rhs)
{
    units_sold += rhs.units_sold;
    saleprice = (rhs.saleprice * rhs.units_sold + saleprice * units_sold)
        / (rhs.units_sold + units_sold);
    if(sellingprice != 0)
        discount = saleprice / sellingprice;
    return *this;
}

Sales_data operator + (const Sales_data& lhs, const Sales_data& rhs)
{
    Sales_data ret(lhs); // 把lhs的内容拷贝到临时变量ret中，这种做法便于运算
    ret += rhs;           // 把rhs的内容加入其中
    return ret;           // 返回ret
}

istream& operator>>(istream& in, Sales_data& s)
{
    in >> s.bookNo >> s.units_sold >> s.sellingprice >> s.saleprice;
    if (in && s.sellingprice != 0)
        s.discount = s.saleprice / s.sellingprice;
    else
        s = Sales_data(); // 输入错误，重置输入的数据
    return in;
}

ostream& operator << (ostream& out, const Sales_data& s)
{
    out << s.isbn() << " " << s.units_sold << " "
        << s.sellingprice << " " << s.saleprice << " " << s.discount;
    return out;
}

int main()
{
    Sales_data book;
    cout << "请输入销售记录: " << endl;
    while (cin >> book) {
        cout << " ISBN、售出本数、原始价格、实售价格、折扣为"
            << book << endl;
    }

    Sales_data transl, trans2;
}

```

```

cout << "请输入两条 ISBN 相同的销售记录: " << endl;
cin >> trans1 >> trans2;
if (compareIsbn(trans1, trans2))
    cout << "汇总信息: ISBN、售出本数、实售价格、折扣为 "
    << trans1 + trans2 << endl;
else
    cout << "两条销售记录的 ISBN 不同" << endl;

Sales_data total, trans;
cout << "请输入几条 ISBN 相同的销售记录: " << endl;
if (cin >> total) {
    while (cin >> trans)
        if (compareIsbn(total, trans)) // ISBN 相同
            total = total + trans;
        else { // ISBN 不同
            cout << "当前书籍 ISBN 不同" << endl;
            break;
        }
    cout << "有效汇总信息: ISBN、售出本数、原始价格、实售价格、折扣为"
    << total << endl;
}
else {
    cout << "没有数据" << endl;
    return -1;
}

int num = 1;      // 记录当前书籍的销售记录总数
cout << "请输入若干销售记录: " << endl;
if (cin >> trans1) {
    while (cin >> trans2)
        if (compareIsbn(trans1, trans2)) // ISBN 相同
            num++;
        else { // ISBN 不同
            cout << trans1.isbn() << "共有"
            << num << "条销售记录" << endl;
            trans1 = trans2;
            num = 1;
        }
    cout << trans1.isbn() << "共有"
    << num << "条销售记录" << endl;
}
else {
    cout << "没有数据" << endl;
    return -1;
}

return 0;
}

```

**练习 3.2：**编写一段程序从标准输入中一次读入一整行，然后修改该程序使其一次读入一个词。

### 【出题思路】

string 头文件包含了很多用于字符串操作的函数，其中常用的字符串读取方式有两种：一种是使用 `getline` 函数一次读入一整行；另一种是使用 `cin` 一次读入一个词，遇空白停止。

### 【解答】

第一种方式是使用 `getline` 函数一次读入一整行，行的结束标识是回车符。如果一开始输入的就是回车符，则 `getline` 直接结束本次读取，所得的结果是一个空字符串。

满足题意的程序如下所示：

```
#include <iostream>
#include <string>

using namespace std;

int main()      // 使用 getline 一次读入一整行，遇回车结束
{
    string line;
    // 循环读取，每次读入一行，直至文件结束或遇到异常输入
    cout << "请输入您的字符串，可以包含空格：" << endl;
    while (getline(cin, line))
        cout << line << endl;

    return 0;
}
```

第二种方式是使用 `cin` 一次读入一个单词，遇到空白停止。满足题意的程序如下所示：

```
#include <iostream>
#include <string>

using namespace std;

int main()      // 使用 cin 一次读入一个词，遇空白结束
{
    string word;
    // 循环读取，每次读入一个词，直至文件结束或遇到异常输入
    cout << "请输入您的单词，不可以包含空格：" << endl;
    while (cin >> word)
        cout << word << endl;    // 为了便于观察，输出每个单词后换行

    return 0;
}
```

**练习 3.3：**请说明 `string` 类的输入运算符和 `getline` 函数分别是如何处理空白字符的。

#### 【出题思路】

本题考查的知识点是两种字符串读取方式对空白字符处理的差异。

#### 【解答】

标准库 `string` 的输入运算符自动忽略字符串开头的空白（包括空格符、换行符、制表符等），从第一个真正的字符开始读起，直到遇见下一处空白为止。

如果希望在最终的字符串中保留输入时的空白符，应该使用 `getline` 函数代替原来的`>>`运算符，`getline` 从给定的输入流中读取数据，直到遇到换行符为止，此时换行符也被读取进来，但是并不存储在最后的字符串中。

一个典型的示例如下所示：

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    string word, line;
    cout << "请选择读取字符串的方式：1 表示逐词读取，2 表示整行读取" << endl;
    char ch;
    cin >> ch;
    if(ch == '1')
    {
        cout << "请输入字符串： Welcome to C++ family! " << endl;
        cin >> word;
        cout << "系统读取的有效字符串是：" << endl;
        cout << word << endl;
        return 0;
    }
    // 清空输入缓冲区
    cin.clear();
    cin.sync();
    if(ch == '2')
    {
        cout << "请输入字符串： Welcome to C++ family! " << endl;
        getline(cin, line);
        cout << "系统读取的有效字符串是：" << endl;
        cout << line << endl;
        return 0;
    }
    cout << "您的输入有误！";

    return -1;
}
```

在上述程序中，如果用户输入 1，则输出是 `Welcome`，此时字符串开头的空格和第一个单词之后的所有内容都被忽略掉了；如果用户输入 2，则输出是 `Welcome`

to C++ family!，字符串开头、中间、结尾的空格都保留了下来。

**练习 3.4：**编写一段程序读入两个字符串，比较其是否相等并输出结果。如果不相等，输出较大的那个字符串。改写上述程序，比较输入的两个字符串是否等长，如果不等长，输出长度较大的那个字符串。

### 【出题思路】

本题考查的知识点是 `string` 对象的比较以及 `string` 对象长度的计算，在 C++11 新标准中，可以使用 `auto` 推断 `string` 对象长度的实际类型。

### 【解答】

比较字符串大小的程序如下所示：

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    string s1, s2;
    cout << "请输入两个字符串：" << endl;
    cin >> s1 >> s2;
    if(s1 == s2)
        cout << "两个字符串相等" << endl;
    else if(s1 > s2)
        cout << s1 << " 大于 " << s2 << endl;
    else
        cout << s2 << " 大于 " << s1 << endl;

    return 0;
}
```

比较字符串长度的程序如下所示：

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    string s1, s2;
    cout << "请输入两个字符串：" << endl;
    cin >> s1 >> s2;
    auto len1 = s1.size();
    auto len2 = s2.size();
    if(len1 == len2)
        cout << s1 << " 和 " << s2 << " 的长度都是 " << len1 << endl;
    else if(len1 > len2)
        cout << s1 << " 比 " << s2 << " 的长度多 " << len1 - len2 << endl;
    else
        cout << s1 << " 比 " << s2 << " 的长度小 " << len2 - len1 << endl;
```

```

    return 0;
}

```

**练习 3.5：**编写一段程序从标准输入中读入多个字符串并将它们连接在一起，输出连接成的大字符串。然后修改上述程序，用空格把输入的多个字符串分隔开来。

#### 【出题思路】

本题旨在考查 `string` 对象的连接操作，需要注意程序逻辑的严谨和友好，确保程序按照用户期望的方式进行。

#### 【解答】

连接多个字符串的程序如下所示：

```

#include <iostream>
#include <string>

using namespace std;

int main()
{
    char cont = 'y';
    string s, result;
    cout << "请输入第一个字符串：" << endl;
    while(cin >> s)
    {
        result += s;
        cout << "是否继续(y or n)?" << endl;
        cin >> cont;
        if(cont == 'y' || cont == 'Y')
            cout << "请输入下一个字符串：" << endl;
        else
            break;
    }
    cout << "拼接后的字符串是：" << result << endl;

    return 0;
}

```

连接多个字符串并以空格分隔的程序如下所示：

```

#include <iostream>
#include <string>

using namespace std;

int main()
{
    char cont = 'y';
    string s, result;
    cout << "请输入第一个字符串：" << endl;
    while(cin >> s)
    {
        if(!result.size())      // 第一个拼接的字符串之前不加空格

```

```

        result += s;
    else           // 之后拼接的每个字符串之前加一个空格
        result = result + " " + s;
    cout << "是否继续(y or n)? " << endl;
    cin >> cont;
    if(cont == 'y' || cont == 'Y')
        cout << "请输入下一个字符串: " << endl;
    else
        break;
}
cout << "拼接后的字符串是: " << result << endl;

return 0;
}

```

**练习 3.6:** 编写一段程序，使用范围 for 语句将字符串内的所有字符用 x 替换。

#### 【出题思路】

范围 for 语句是 C++11 新定义的语法形式，当我们处理字符串中的每一个字符时使用范围 for 语句非常方便。

#### 【解答】

将字符串的所有字符替换为 x 的程序如下所示：

```

#include <iostream>
#include <string>

using namespace std;

int main()
{
    string s;
    cout << "请输入一个字符串，可以包含空格: " << endl;
    getline(cin, s);      // 读取整行，遇回车符结束
    for(auto &c : s)     // 依次处理字符串中的每一个字符
    {
        c = 'X';
    }
    cout << s << endl;

    return 0;
}

```

有两点值得注意：一是利用 auto 关键字推断字符串中每一个元素的类型；二是 c 必须定义为引用类型，否则无法修改字符串内容。

**练习 3.7:** 就上一题完成的程序而言，如果将循环控制变量的类型设为 char 将发生什么？先估计一下结果，然后实际编程进行验证。

#### 【出题思路】

本题旨在考查 `auto` 自动推断变量类型与直接指定类型的异同。

### 【解答】

修改后的程序如下所示：

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    string s;
    cout << "请输入一个字符串，可以包含空格：" << endl;
    getline(cin, s);
    for(char &c : s)
    {
        c = 'X';
    }
    cout << s << endl;

    return 0;
}
```

就本题而言，将循环控制变量的类型设为 `char` 不会对程序运行结果造成影响，因为我们使用 `auto` 自动推断字符串 `s` 的元素类型，结果同样是 `char`。

**练习 3.8：** 分别用 `while` 循环和传统的 `for` 循环重写练习 3.6 的程序，你觉得哪种形式更好呢？为什么？

### 【出题思路】

本题旨在引导读者对比 C++11 新引入的范围 `for` 循环与传统的 `while` 循环和 `for` 循环之间的异同，如果准备处理范围内的每一个元素，则一般而言使用范围 `for` 循环更好。

### 【解答】

使用 `while` 循环实现的程序如下所示：

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    string s;
    cout << "请输入一个字符串，可以包含空格：" << endl;
    getline(cin, s);
    int i = 0;
    while(s[i] != '\0')
    {
        s[i] = 'X';
        ++i;
    }
}
```

```

    }
    cout << s << endl;

    return 0;
}

```

使用传统 for 循环实现的程序如下所示：

```

#include <iostream>
#include <string>

using namespace std;

int main()
{
    string s;
    cout << "请输入一个字符串，可以包含空格：" << endl;
    getline(cin, s);
    for(unsigned int i = 0; i < s.size(); i++)
    {
        s[i] = 'X';
    }
    cout << s << endl;

    return 0;
}

```

在本例中，我们希望处理字符串中的每一个字符，且无须在意字符的处理顺序，因此与传统的 while 循环和 for 循环相比，使用范围 for 循环更简洁直观。

**练习 3.9：**下面的程序有何作用？它合法吗？如果不合法，为什么？

```

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

```

#### 【出题思路】

本题旨在提醒读者注意，利用下标访问字符串内容时，必须确保下标在合法范围内。

#### 【解答】

该程序的原意是输出字符串 s 的首字符，但程序是错误的。因为初始状态下没有给 s 赋任何初值，所以字符串 s 的内容为空，当然也就不存在首字符，下标 0 是非法的。

但是在某些编译器环境中，上述语句并不会引发编译错误。

**练习 3.10：**编写一段程序，读入一个包含标点符号的字符串，将标点符号去除后输出字符串剩余的部分。

#### 【出题思路】

字符串遍历和修改有几种不同的方法，本题旨在考查读者是否对该项技能熟练掌握并能够灵活运用。

**【解答】**

解题思路一，利用范围 for 语句遍历字符串，逐个输出非标点字符：

```
#include <iostream>
#include <string>
#include <cctype>

using namespace std;

int main()
{
    string s;
    cout << "请输入一个字符串，最好含有某些标点符号：" << endl;
    getline(cin, s);
    for(auto c : s)
    {
        if(!ispunct(c))
            cout << c;
    }
    cout << endl;

    return 0;
}
```

解题思路二，利用普通 for 循环遍历字符串，通过下标执行随机访问，把非标点字符拼接成一个新串后输出：

```
#include <iostream>
#include <string>
#include <cctype>

using namespace std;

int main()
{
    string s, result;
    cout << "请输入一个字符串，最好含有某些标点符号：" << endl;
    getline(cin, s);
    for(decltype(s.size()) i = 0; i < s.size(); i++)
    {
        if(!ispunct(s[i]))
            result += s[i];
    }
    cout << result << endl;

    return 0;
}
```

**练习 3.11：**下面的范围 for 语句合法吗？如果合法，c 的类型是什么？

```
const string s = "Keep out!";
for (auto &c : s) { /* ... */ }
```

**【出题思路】**

本题旨在考查范围 for 循环中循环变量的类型与 auto 自动推断的关系。

**【解答】**

该程序段从语法上来说是合法的，`s` 是一个常量字符串，则 `c` 的推断类型是常量引用，即 `c` 所绑定的对象值不能改变。为了证明这一点，读者不妨尝试下面的程序：

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    const string s = "Keep out!";
    for (auto &c : s)
    {
        c='X';
        // 其他对 c 的操作
    }

    return 0;
}
```

该段程序不能正确编译，因为 `c` 是绑定到常量的引用，其值不能改变。作者所用的 GCC 编译器给出的错误信息是：

```
||== Build: Debug in 11 (compiler: GNU GCC Compiler) ==|
G:\chapter3\11\main.cpp||In function 'int main()':|
G:\chapter3\11\main.cpp|11|error: assignment of read-only reference
'c'||
||== Build failed: 1 error(s), 0 warning(s) (0 minute(s), 0 second(s))
==|
```

**练习 3.12：**下列 `vector` 对象的定义有不正确的吗？如果有，请指出来。对于正确的，描述其执行结果；对于不正确的，说明其错误的原因。

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

**【出题思路】**

本题考查的知识点是 `vector` 对象的定义和初始化。

**【解答】**

(a) 是正确的，定义了一个名为 `ivec` 的 `vector` 对象，其中的每个元素都是 `vector<int>` 对象。

(b) 是错误的，`svec` 的元素类型是 `string`，而 `ivec` 的元素类型是 `int`，因此不能使用 `ivec` 初始化 `svec`。

(c) 是正确的，定义了一个名为 `svec` 的 `vector` 对象，其中含有 10 个元素，每个元素都是字符串 `null`。

**练习 3.13:** 下列的 `vector` 对象各包含多少个元素？这些元素的值分别是多少？

- |   |   |
|---|---|
| (a) <code>vector&lt;int&gt; v1;</code>              | (b) <code>vector&lt;int&gt; v2(10);</code>    |
| (c) <code>vector&lt;int&gt; v3(10, 42);</code>      | (d) <code>vector&lt;int&gt; v4{10};</code>    |
| (e) <code>vector&lt;int&gt; v5{10, 42};</code>      | (f) <code>vector&lt;string&gt; v6{10};</code> |
| (g) <code>vector&lt;string&gt; v7{10, "hi"};</code> |   |

#### 【出题思路】

初始化 `vector` 对象的方法有很多，其中一些形式上比较类似，本题旨在考查读者对这些初始化方法的掌握情况。

#### 【解答】

- (a) 的元素数量为 0。
- (b) 的元素数量为 10，每一个元素都被初始化为 0。
- (c) 的元素数量为 10，每一个元素都被初始化为 42。
- (d) 的元素数量为 1，元素的值为 10。
- (e) 的元素数量为 2，两个元素的值分别是 10 和 42。
- (f) 的元素数量为 10，每一个元素都被初始化为空串。
- (g) 的元素数量为 10，每一个元素都被初始化为 " hi "。

**练习 3.14:** 编写一段程序，用 `cin` 读入一组整数并把它们存入一个 `vector` 对象。

#### 【出题思路】

对于 `vector` 对象来说，直接初始化的方式适用于三种情况：一是初始值已知且数量较少；二是初始值是另一个 `vector` 对象的副本；三是所有元素的初始值都一样。然而一般情况下，上述条件很难满足，这时就需要利用 `push_back` 函数向 `vector` 对象中逐个添加元素。

#### 【解答】

满足题意的程序如下所示：

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector<int> vInt;           // 元素类型为 int 的 vector 对象
    int i;                      // 记录用户的输入值
    char cont = 'y';            // 与用户交互，决定是否继续输入
    while(cin >> i)
    {
        vInt.push_back(i);     // 向 vector 对象中添加元素
        cout << "您要继续吗 (y or n)? " << endl;
        cin >> cont;
        if(cont != 'y' && cont != 'Y')
            break;
    }
}
```

```

    for(auto mem : vInt)      // 使用范围 for 循环语句遍历 vInt 中的每个元素
        cout << mem << " ";
    cout << endl;
    return 0;
}

```

**练习 3.15：**改写上题的程序，不过这次读入的是字符串。

#### 【出题思路】

与上题的区别是，`vector` 对象的元素类型从 `int` 变为了 `string`。

#### 【解答】

满足题意的程序如下所示：

```

#include <iostream>
#include <vector>
#include <string>

using namespace std;

int main()
{
    vector<string> vString;          // 元素类型为 string 的 vector 对象
    string s;                      // 记录用户的输入值
    char cont = 'y';                // 与用户交互，决定是否继续输入
    while(cin >> s)
    {
        vString.push_back(s);       // 向 vector 对象中添加元素
        cout << "您要继续吗 (y or n) ? " << endl;
        cin >> cont;
        if(cont != 'y' && cont != 'Y')
            break;
    }
    for(auto mem : vString)         // 使用范围 for 循环语句遍历 vString 中的每个元素
        cout << mem << " ";
    cout << endl;
    return 0;
}

```

**练习 3.16：**编写一段程序，把练习 3.13 中 `vector` 对象的容量和具体内容输出出来。检验你之前的回答是否正确，如果不对，回过头重新学习第 3.3.1 节（第 87 页），直到弄明白错在何处为止。

#### 【出题思路】

利用 `size` 函数得到 `vector` 对象的容量（元素个数），利用范围 `for` 循环遍历 `vector` 对象的每一个元素，然后逐个输出。

#### 【解答】

用于检验的程序如下所示：

```
#include <iostream>
```

```

#include <vector>
#include <string>

using namespace std;

int main()
{
    vector<int> v1;
    vector<int> v2(10);
    vector<int> v3(10, 42);
    vector<int> v4{10};
    vector<int> v5{10, 42};
    vector<string> v6{10};
    vector<string> v7{10, "hi"};

    cout << "v1 的元素个数是: " << v1.size() << endl;
    if(v1.size() > 0)          // 当 vector 含有元素时逐个输出
    {
        cout << "v1 的元素分别是: " << endl;
        for(auto e : v1)        // 使用范围 for 语句遍历每一个元素
            cout << e << " ";
        cout << endl;
    }

    cout << "v2 的元素个数是: " << v2.size() << endl;
    if(v2.size() > 0)          // 当 vector 含有元素时逐个输出
    {
        cout << "v2 的元素分别是: " << endl;
        for(auto e : v2)        // 使用范围 for 语句遍历每一个元素
            cout << e << " ";
        cout << endl;
    }

    cout << "v3 的元素个数是: " << v3.size() << endl;
    if(v3.size() > 0)          // 当 vector 含有元素时逐个输出
    {
        cout << "v3 的元素分别是: " << endl;
        for(auto e : v3)        // 使用范围 for 语句遍历每一个元素
            cout << e << " ";
        cout << endl;
    }

    cout << "v4 的元素个数是: " << v4.size() << endl;
    if(v4.size() > 0)          // 当 vector 含有元素时逐个输出
    {
        cout << "v4 的元素分别是: " << endl;
        for(auto e : v4)        // 使用范围 for 语句遍历每一个元素
            cout << e << " ";
        cout << endl;
    }

    cout << "v5 的元素个数是: " << v5.size() << endl;

```

```

if(v5.size() > 0)      // 当 vector 含有元素时逐个输出
{
    cout << "v5 的元素分别是：" << endl;
    for(auto e : v5)    // 使用范围 for 语句遍历每一个元素
        cout << e << " ";
    cout << endl;
}

cout << "v6 的元素个数是：" << v6.size() << endl;
if(v6.size() > 0)      // 当 vector 含有元素时逐个输出
{
    cout << "v6 的元素分别是：" << endl;
    for(auto e : v6)    // 使用范围 for 语句遍历每一个元素
        cout << e << " ";
    cout << endl;
}

cout << "v7 的元素个数是：" << v7.size() << endl;
if(v7.size() > 0)      // 当 vector 含有元素时逐个输出
{
    cout << "v7 的元素分别是：" << endl;
    for(auto e : v7)    // 使用范围 for 语句遍历每一个元素
        cout << e << " ";
    cout << endl;
}

return 0;
}

```

**练习 3.17：**从 cin 读入一组词并把它们存入一个 vector 对象，然后设法把所有词都改写为大写形式。输出改变后的结果，每个词占一行。

### 【出题思路】

本题的知识点包括：向 vector 对象中添加元素、范围 for 循环语句、利用 toupper 函数将词改写为大写形式。

### 【解答】

满足题意的程序如下所示：

```

#include <iostream>
#include <vector>
#include <string>

using namespace std;

int main()
{
    vector<string> vString;      // 元素类型为 string 的 vector 对象
    string s;                    // 记录用户的输入值
    char cont = 'y';            // 与用户交互，决定是否继续输入
    cout << "请输入第一个词：" << endl;

```

```

while(cin >> s)
{
    vString.push_back(s); // 向vector对象中添加元素
    cout << "您要继续吗(y or n)? " << endl;
    cin >> cont;
    if(cont != 'y' && cont != 'Y')
        break;
    cout << "请输入下一个词: " << endl;
}
cout << "转换后的结果是: " << endl;
for(auto &mem : vString) // 使用范围for循环语句遍历vString中的每个元素
{
    for(auto &c : mem) // 使用范围for循环语句遍历mem中的每个字符
        c = toupper(c); // 改写为大写字母形式
    cout << mem << endl;
}

return 0;
}

```

**练习 3.18:** 下面的程序合法吗？如果不合法，你准备如何修改？

```

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

```

### 【出题思路】

本题旨在考查利用下标运算符访问 vector 元素的问题，注意：vector 对象的下标运算符只能用于访问已经存在的元素，而不能用于添加元素。

### 【解答】

该程序是非法的，因为 ivec 目前没有任何元素，因此 ivec[0] 的形式是错误的，程序试图访问的元素根本不存在。要想向 vector 对象中添加新元素，需要使用 push\_back 函数。

下面是修改后的程序代码如下所示：

```

vector<int> ivec;
ivec.push_back(42);

```

**练习 3.19:** 如果想定义一个含有 10 个元素的 vector 对象，所有元素的值都是 42，请列举出三种不同的实现方法。哪种方法更好呢？为什么？

### 【出题思路】

本题旨在考查定义及初始化 vector 对象的方法，不同的初始化方法有各自适应的场景。

### 【解答】

解决思路一：先定义一个空 vector 对象，然后添加元素。

```

vector<int> vInt;
for(int i = 0; i < 10; i++)
    vInt.push_back(42);

```

解决思路二：列表初始化，罗列出全部10个元素的值。

```
vector<int> vInt = {42, 42, 42, 42, 42, 42, 42, 42, 42, 42};
```

解决思路三：用括号给出所有元素的值，效果类似于解决思路二。

```
vector<int> vInt{42, 42, 42, 42, 42, 42, 42, 42, 42, 42};
```

解决思路四：定义的时候使用参数指定元素个数及重复的值。

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

解决思路五：先指定元素个数，再利用范围 for 循环依次为元素赋值。

```
vector<int> vInt(10);
for(auto &i : vInt)
    i = 42;
```

显然，思路四采用的初始化方式形式上最简洁直观，当 vector 对象的元素数量较多且取值重复时是最好的选择；而思路一在开始的时候不限定元素的个数，比较灵活。

**练习 3.20：**读入一组整数并把它们存入一个 vector 对象，将每对相邻整数的和输出出来。改写你的程序，这次要求先输出第 1 个和最后 1 个元素的和，接着输出第 2 个和倒数第 2 个元素的和，以此类推。

### 【出题思路】

本题旨在考查 vector 对象的初始化及利用下标访问 vector 对象的元素的方法。

### 【解答】

求相邻元素和的程序如下所示：

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector<int> vInt;
    int iVal;
    cout << "请输入一组数字：" << endl;
    while(cin >> iVal)
        vInt.push_back(iVal);
    if(vInt.size() == 0)
    {
        cout << "没有任何元素" << endl;
        return -1;
    }
    cout << "相邻两项的和依次是：" << endl;
    // 利用 decltype 推断 i 的类型
    for(decltype(vInt.size()) i = 0; i < vInt.size() - 1; i += 2)
    {
        // 求相邻两项的和
        cout << vInt[i] + vInt[i+1] << " ";
        // 每行输出 5 个数字
    }
}
```

```

        if((i + 2) % 10 == 0)
            cout << endl;
    }
    // 如果元素数是奇数，单独处理最后一个元素
    if(vInt.size() % 2 != 0)
        cout << vInt[vInt.size() - 1];

    return 0;
}

```

求首尾元素和的程序如下所示：

```

#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector<int> vInt;
    int iVal;
    cout << "请输入一组数字：" << endl;
    while(cin >> iVal)
        vInt.push_back(iVal);
    if(vInt.size() == 0)
    {
        cout << "没有任何元素" << endl;
        return -1;
    }
    cout << "首尾两项的和依次是：" << endl;
    // 利用 decltype 推断 i 的类型
    for(decltype(vInt.size()) i = 0; i < vInt.size() / 2; i++)
    {
        // 求首尾两项的和
        cout << vInt[i] + vInt[vInt.size() - i - 1] << " ";
        // 每行输出 5 个数字
        if((i + 1) % 5 == 0)
            cout << endl;
    }
    // 如果元素数是奇数，单独处理最后一个元素
    if(vInt.size() % 2 != 0)
        cout << vInt[vInt.size() / 2];

    return 0;
}

```

**练习 3.21：**请使用迭代器重做 3.3.3 节（第 94 页）的第一个练习。

### 【出题思路】

迭代器是一种访问容器元素的通用机制，与指针类型类似，迭代器也提供了对对象的间接访问。使用迭代器可以访问某个元素，迭代器也能从一个元素移动到另外一个元素。因为本题只需输出 `vector` 对象的内容而无须对其进行更改，所以使

用的迭代器应该是 `cbegin` 和 `cend`，而非 `begin` 和 `end`。

### 【解答】

利用迭代器改写练习 3.16 所得的程序如下所示：

```
#include <iostream>
#include <vector>
#include <string>

using namespace std;

int main()
{
    vector<int> v1;
    vector<int> v2(10);
    vector<int> v3(10, 42);
    vector<int> v4{10};
    vector<int> v5{10, 42};
    vector<string> v6{10};
    vector<string> v7{10, "hi"};

    cout << "v1 的元素个数是: " << v1.size() << endl;
    if(v1.cbegin() != v1.cend()) // 当 vector 含有元素时逐个输出
    {
        cout << "v1 的元素分别是: " << endl;
        // 使用范围 for 语句遍历每一个元素
        for(auto it = v1.cbegin(); it != v1.cend(); it++)
            cout << *it << " ";
        cout << endl;
    }

    cout << "v2 的元素个数是: " << v2.size() << endl;
    if(v2.cbegin() != v2.cend()) // 当 vector 含有元素时逐个输出
    {
        cout << "v2 的元素分别是: " << endl;
        // 使用范围 for 语句遍历每一个元素
        for(auto it = v2.cbegin(); it != v2.cend(); it++)
            cout << *it << " ";
        cout << endl;
    }

    cout << "v3 的元素个数是: " << v3.size() << endl;
    if(v3.cbegin() != v3.cend()) // 当 vector 含有元素时逐个输出
    {
        cout << "v3 的元素分别是: " << endl;
        // 使用范围 for 语句遍历每一个元素
        for(auto it = v3.cbegin(); it != v3.cend(); it++)
            cout << *it << " ";
        cout << endl;
    }

    cout << "v4 的元素个数是: " << v4.size() << endl;
    if(v4.cbegin() != v4.cend()) // 当 vector 含有元素时逐个输出
```

```

{
    cout << "v4 的元素分别是: " << endl;
    // 使用范围 for 语句遍历每一个元素
    for(auto it = v4.cbegin(); it != v4.cend(); it++)
        cout << *it << " ";
    cout << endl;
}

cout << "v5 的元素个数是: " << v5.size() << endl;
if(v5.cbegin() != v5.cend())           // 当 vector 蕴含元素时逐个输出
{
    cout << "v5 的元素分别是: " << endl;
    // 使用范围 for 语句遍历每一个元素
    for(auto it = v5.cbegin(); it != v5.cend(); it++)
        cout << *it << " ";
    cout << endl;
}

cout << "v6 的元素个数是: " << v6.size() << endl;
if(v6.cbegin() != v6.cend())           // 当 vector 蕴含元素时逐个输出
{
    cout << "v6 的元素分别是: " << endl;
    // 使用范围 for 语句遍历每一个元素
    for(auto it = v6.cbegin(); it != v6.cend(); it++)
        cout << *it << " ";
    cout << endl;
}

cout << "v7 的元素个数是: " << v7.size() << endl;
if(v7.cbegin() != v7.cend())           // 当 vector 蕴含元素时逐个输出
{
    cout << "v7 的元素分别是: " << endl;
    // 使用范围 for 语句遍历每一个元素
    for(auto it = v7.cbegin(); it != v7.cend(); it++)
        cout << *it << " ";
    cout << endl;
}

return 0;
}

```

**练习 3.22：**修改之前那个输出 text 第一段的程序，首先把 text 的第一段全都改成大写形式，然后再输出它。

### 【出题思路】

与原书的示例程序相比，需要将第一段（vector 对象第一个空串元素之前的所有元素）改写成大写字母的形式再输出。因为需要更改 vector 对象的内容，所以使用的迭代器应该是 begin 和 end，而非 cbegin 和 cend。

### 【解答】

满足题意的程序如下所示：

```
#include <iostream>
#include <vector>
#include <string>

using namespace std;

int main()
{
    vector<string> text;
    string s;
    // 利用 getline 读取一句话，直接回车产生一个空串，表示段落结束
    while(getline(cin, s))
        text.push_back(s);           // 逐个添加到 text 中
    // 利用迭代器遍历全部字符串，遇空串停止循环
    for(auto it = text.begin(); it != text.end() && !it->empty(); it++)
    {
        // 利用迭代器遍历当前字符串
        for(auto it2 = it -> begin(); it2 != it -> end(); it2++)
            *it2 = toupper(*it2);   // 利用 toupper 改写成大写形式
        cout << *it << endl;       // 输出当前字符串
    }

    return 0;
}
```

**练习 3.23：**编写一段程序，创建一个含有 10 个整数的 vector 对象，然后使用迭代器将所有元素的值都变成原来的两倍。输出 vector 对象的内容，检验程序是否正确。

### 【出题思路】

本题与之前题目的区别是由程序自动生成随机数并添加到 vector 对象中，当输出原始数据时，只需读取而无须更改，所以迭代器选用 cbegin 和 cend；当执行翻倍计算时，需要读写元素内容，所以迭代器选用 begin 和 end。

### 【解答】

满足题意的程序如下所示：

```
#include <iostream>
#include <vector>
#include <ctime>
#include <cstdlib>

using namespace std;

int main()
{
    vector<int> vInt;
    srand((unsigned)time(NULL));    // 生成随机数种子
    for(int i = 0; i < 10; i++)    // 循环 10 次
    {
```

```

    // 每次循环生成一个 1000 以内的随机数并添加到 vInt 中
    vInt.push_back( rand() % 1000 );
}

cout << "随机生成的 10 个数字是: " << endl;
// 利用常量迭代器读取原始数据
for(auto it = vInt.cbegin(); it != vInt.cend(); it++)
{
    cout << *it << " ";           // 输出当前数字
}
cout << endl;
cout << "翻倍后的 10 个数字是: " << endl;
// 利用非常量迭代器修改 vInt 内容并输出
for(auto it = vInt.begin(); it != vInt.end(); it++)
{
    *it *= 2;
    cout << *it << " ";           // 输出当前数字
}
cout << endl;

return 0;
}

```

**练习 3.24:** 请使用迭代器重做 3.3.3 节（第 94 页）的最后一个练习。

#### 【出题思路】

本题旨在考查如何使用迭代器向 vector 向量中添加元素以及如何访问已有的元素。

#### 【解答】

求相邻元素和的程序如下所示：

```

#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector<int> vInt;
    int iVal;
    cout << "请输入一组数字: " << endl;
    while(cin >> iVal)
        vInt.push_back(iVal);
    if(vInt.cbegin() == vInt.cend())
    {
        cout << "没有任何元素" << endl;
        return -1;
    }
    cout << "相邻两项的和依次是: " << endl;
    // 利用 auto 推断 it 的类型
    for(auto it = vInt.cbegin(); it != vInt.cend() - 1; it++)
    {

```

```

    // 求相邻两项的和
    cout << (*it + *(++it)) << " ";
    // 每行输出 5 个数字
    if((it - vInt.cbegin() + 1) % 10 == 0)
        cout << endl;
    }
    // 如果元素个数是奇数，单独处理最后一个元素
    if(vInt.size() % 2 != 0)
        cout << *(vInt.cend() - 1);

    return 0;
}

```

求首尾元素和的程序如下所示：

```

#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector<int> vInt;
    int iVal;
    cout << "请输入一组数字：" << endl;
    while(cin >> iVal)
        vInt.push_back(iVal);
    if(vInt.cbegin() == vInt.cend())
    {
        cout << "没有任何元素" << endl;
        return -1;
    }
    cout << "首尾两项的和依次是：" << endl;
    auto beg = vInt.begin();
    auto end = vInt.end();
    // 利用 auto 推断 it 的类型
    for(auto it = beg; it != beg + (end - beg) / 2; it++)
    {
        // 求首尾两项的和
        cout << (*it + *(beg + (end - it) - 1)) << " ";
        // 每行输出 5 个数字
        if((it - beg + 1) % 5 == 0)
            cout << endl;
    }
    // 如果元素个数是奇数，单独处理最后一个元素
    if(vInt.size() % 2 != 0)
        cout << *(beg + (end - beg) / 2);

    return 0;
}

```

求解本题时应该特别注意迭代器 `begin` 和 `end` 的含义，其中 `begin` 指向容器的首元素而 `end` 指向容器的最后一个元素的下一位位置。只有熟悉上述定义才能精确推断迭代器的当前位置在哪里。

**练习 3.25:** 3.3.3 节（第 93 页）划分分数段的程序是使用下标运算符实现的，请利用迭代器改写该程序并实现完全相同的功能。

#### 【出题思路】

与指针类似，C++提供了迭代器的算术运算操作，使得迭代器可以在元素间移动；同时我们也可以通过解引用迭代器来获取它所指示的元素，前提是确保迭代器合法。

#### 【解答】

满足题意的程序如下所示：

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    // 该 vector 对象记录各分数段的人数，初始值均为 0
    vector<unsigned> vUS(11);
    auto it = vUS.begin();
    int iVal;
    cout << "请输入一组成绩 (0~100): " << endl;
    while(cin >> iVal)
        if(iVal < 101)                      // 成绩应在合理范围之内
            ++*(it + iVal / 10);           // 利用迭代器定位到对应的元素，加 1

    cout << "您总计输入了 " << vUS.size() << " 个成绩" << endl;
    cout << "各分数段的人数分布是 (成绩从低到高): " << endl;
    // 利用迭代器遍历 vUS 的元素并逐个输出
    for(it = vUS.begin(); it != vUS.end(); it++)
    {
        cout << *it << " ";
    }
    cout << endl;

    return 0;
}
```

**练习 3.26:** 在 100 页的二分搜索程序中，为什么用的是  $mid = beg + (end - beg) / 2;$ ，而非  $mid = (beg + end) / 2;$ ？

#### 【出题思路】

本题旨在考查迭代器的运算类型及含义。

#### 【解答】

C++并没有定义两个迭代器的加法运算，实际上直接把两个迭代器加起来是没有意义的。

与之相反，C++定义了迭代器的减法运算，两个迭代器相减的结果是它们之间的距离，也就是说，将运算符右侧的迭代器向前移动多少个元素后可以得到左侧的迭

代器，参与运算的两个迭代器必须指向同一个容器中的元素或尾后元素。

另外，C++还定义了迭代器与整数的加减法运算，用以控制迭代器在容器中左右移动。

在本题中，因为迭代器的加法不存在，所以  $mid = (\text{beg} + \text{end}) / 2$ ; 不合法。  
 $\text{mid} = \text{beg} + (\text{end} - \text{beg}) / 2$ ; 的含义是，先计算  $\text{end}-\text{beg}$  的值得到容器中的元素个数，然后控制迭代器从开始处向右移动二分之一容器的长度，从而定位到容器正中间的元素。

**练习 3.27：**假设 `txt_size` 是一个无参数的函数，它的返回值是 `int`。请回答下列哪个定义是非法的？为什么？

- ```
unsigned buf_size = 1024;
(a) int ia[buf_size];
(b) int ia[4 * 7 - 14];
(c) int ia[txt_size()];
(d) char st[11] = "fundamental";
```

#### 【出题思路】

本题考查数组的定义和初始化。数组是一种复合类型，其声明形如 `a[d]`，`a` 是数组的名字，`d` 是数组的维度（容量）。对数组维度的要求有两个，一是维度表示数组中元素的个数，因此必须大于 0；二是维度属于数组类型的一部分，因此在编译时应该是已知的，必须是一个常量表达式。

#### 【解答】

(a) 是非法的，`buf_size` 是一个普通的无符号数，不是常量，不能作为数组的维度。

(b) 是合法的，`4 * 7 - 14 = 14` 是一个常量表达式。

(c) 是非法的，`txt_size()` 是一个普通的函数调用，没有被定义为 `constexpr`，不能作为数组的维度。

(d) 是非法的，当使用字符串初始化字符数组时，默认在尾部添加一个空字符 '\0'，算上这个符号该字符串共有 12 个字符，但是字符数组 `st` 的维度只有 11，无法容纳题目中的字符串。

需要指出的是，在某些编译器环境中，上面的个别语句被判定为合法，这是所谓的编译器扩展。不过一般来说，建议读者避免使用非标准特性，因为含有非标准特性的程序很可能在其他编译器上失效。

**练习 3.28：**下列数组中元素的值是什么？

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

#### 【出题思路】

本题旨在考查数组默认初始化的几种不同情况，如全局变量和局部变量的区别、内置类型和复合类型的区别。

### 【解答】

与练习 2.10 类似，对于 `string` 类型的数组来说，因为 `string` 类本身接受无参数的初始化方式，所以不论数组定义在函数内还是函数外都被默认初始化为空串。

对于内置类型 `int` 来说，数组 `ia` 定义在所有函数体之外，根据 C++ 的规定，`ia` 的所有元素默认初始化为 0；而数组 `ia2` 定义在 `main` 函数的内部，将不被初始化，如果程序试图拷贝或输出未初始化的变量，将遇到未定义的奇异值。

下面的程序可以验证上述分析：

```
#include <iostream>

using namespace std;
// 定义在全局作用域中的数组
string sa[10];
int ia[10];

int main()
{
    // 定义在局部作用域中的数组
    string sa2[10];
    int ia2[10];
    for(auto c : sa)
        cout << c << " ";
    cout << endl;
    for(auto c : ia)
        cout << c << " ";
    cout << endl;
    for(auto c : sa2)
        cout << c << " ";
    cout << endl;
    for(auto c : ia2)
        cout << c << " ";
    return 0;
}
```

**练习 3.29：**相比于 `vector` 来说，数组有哪些缺点，请列举一些。

### 【出题思路】

数组与 `vector` 有一些类似之处，但是也有若干区别。

### 【解答】

数组与 `vector` 的相似之处是都能存放类型相同的对象，且这些对象本身没有名字，需要通过其所在位置访问。

数组与 `vector` 的最大不同是，数组的大小固定不变，不能随意向数组中增加额外的元素，虽然在某些情境下运行时性能较好，但是与 `vector` 相比损失了灵活性。

具体来说，数组的维度在定义时已经确定，如果我们想更改数组的长度，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中去。我们也无法像 `vector` 那样使用 `size` 函数直接获取数组的维度。如果是字符数组，可以调用 `strlen` 函数得到字符串的长度；如果是其他数组，只能使用 `sizeof(array)/sizeof(array[0])` 的方式计算数组的维度。

### 练习 3.30：指出下面代码中的索引错误。

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

#### 【出题思路】

本题旨在考查通过下标访问数组元素时可能发生的访问越界错误。数组的下标是否在合理范围之内应由程序员负责检查。对于一个程序来说即使编译通过，也不能排除包含越界错误的可能。

#### 【解答】

本题的原意是创建一个包含 10 个整数的数组，并把数组的每个元素初始化为元素的下标值。

上面的程序在 `for` 循环终止条件处有错，数组的下标应该大于等于 0 而小于数组的大小，在本题中下标的范围应该是 0~9。因此程序应该修改为：

```
constexpr size_t array_size = 10;
int ia[array_size];
for (size_t ix = 0; ix < array_size; ++ix)
    ia[ix] = ix;
```

### 练习 3.31：编写一段程序，定义一个含有 10 个 `int` 的数组，令每个元素的值就是其所在位置的值。

#### 【出题思路】

通过循环为数组的元素赋值，通过范围 `for` 循环输出数组的全部元素。

#### 【解答】

满足题意的程序如下所示：

```
#include <iostream>

using namespace std;

int main()
{
    const int sz = 10; // 常量 sz 作为数组的维度
    int a[sz];
    // 通过 for 循环为数组元素赋值
    for(int i = 0; i < sz; i++)
        a[i] = i;
```

```

// 通过范围 for 循环输出数组的全部元素
for(auto val : a)
    cout << val << " ";
cout << endl;

return 0;
}

```

**练习 3.32：**将上一题刚刚创建的数组拷贝给另外一个数组。利用 `vector` 重写程序，实现类似的功能。

#### 【出题思路】

如果想把数组的内容拷贝给另一个数组，不能直接对数组使用赋值运算符，而应该逐一拷贝数组的元素。`vector` 的拷贝原理与数组类似。

#### 【解答】

实现数组拷贝的程序如下所示：

```

#include <iostream>

using namespace std;

int main()
{
    const int sz = 10; // 常量 sz 作为数组的维度
    int a[sz], b[sz];
    // 通过 for 循环为数组元素赋值
    for(int i = 0; i < sz; i++)
        a[i] = i;
    for(int j = 0; j < sz; j++)
        b[j] = a[j];
    // 通过范围 for 循环输出数组的全部元素
    for(auto val : b)
        cout << val << " ";
    cout << endl;

    return 0;
}

```

实现 `vector` 拷贝的程序如下所示：

```

#include <iostream>
#include <vector>

using namespace std;

int main()
{
    const int sz = 10; // 常量 sz 作为 vector 的容量
    vector<int> vInt, vInt2;
    // 通过 for 循环为 vector 对象的元素赋值
    for(int i = 0; i < sz; i++)
        vInt.push_back(i);

```

```

    for(int j = 0; j < sz; j++)
        vInt2.push_back(vInt[j]);
    // 通过范围 for 循环输出 vector 对象的全部元素
    for(auto val : vInt2)
        cout << val << " ";
    cout << endl;

    return 0;
}

```

**练习 3.33:** 对于 104 页的程序来说，如果不初始化 `scores` 将发生什么？

**【出题思路】**

本题旨在考查内置类型数组的初始化。

**【解答】**

该程序对 `scores` 执行了列表初始化，为所有元素赋初值为 0，这样在后续统计时将会从 0 开始计算各个分数段的人数，是正确的做法。

如果不初始化 `scores`，则该数组会含有未定义的数值，这是因为 `scores` 是定义在函数内部的整型数组，不会执行默认初始化。

**练习 3.34:** 假定 `p1` 和 `p2` 指向同一个数组中的元素，则下面程序的功能是什么？什么情况下该程序是非法的？

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

**【出题思路】**

指针的算术运算与 `vector` 类似，也可以执行递增、递减、比较、与整数相加、两个指针相减等操作。

**【解答】**

如果 `p1` 和 `p2` 指向同一个数组中的元素，则该语句令 `p1` 指向 `p2` 原来所指向的元素。从语法上来说，即使 `p1` 和 `p2` 指向的元素不属于同一个数组，但只要 `p1` 和 `p2` 的类型相同，该语句也是合法的。

如果 `p1` 和 `p2` 的类型不同，则编译时报错。

**练习 3.35:** 编写一段程序，利用指针将数组中的元素置为 0。

**【出题思路】**

C++11 新标准为数组引入了名为 `begin` 和 `end` 的两个函数，这两个函数与容器中的同名成员功能类似，利用 `begin` 和 `end` 可以方便地定位到数组的边界。令指针在数组的元素间移动，解引用指针即可得到当前所指的元素值。

**【解答】**

满足题意的程序如下所示：

```
#include <iostream>
```

```

using namespace std;

int main()
{
    const int sz = 10;           // 常量 sz 作为数组的维度
    int a[sz], i = 0;
    // 通过 for 循环为数组元素赋值
    for(i = 0; i < 10; i++)
        a[i] = i;
    cout << "初始状态下数组的内容是: " << endl;
    for(auto val : a)
        cout << val << " ";
    cout << endl;
    int *p = begin(a);          // 令 p 指向数组首元素
    while(p != end(a))
    {
        *p = 0;                // 修改 p 所指元素的值
        p++;                   // p 向后移动一位
    }
    cout << "修改后的数组内容是: " << endl;
    // 通过范围 for 循环输出数组的全部元素
    for(auto val : a)
        cout << val << " ";
    cout << endl;

    return 0;
}

```

**练习 3.36:** 编写一段程序，比较两个数组是否相等。再写一段程序，比较两个 vector 对象是否相等。

#### 【出题思路】

无论对比两个数组是否相等还是两个 vector 对象是否相等，都必须逐一比较其元素。

#### 【解答】

对比两个数组是否相等的程序如下所示，因为长度不等的数组一定不相等，并且数组的维度一开始就要确定，所以为了简化起见，程序中设定两个待比较的数组维度一致，仅比较对应的元素是否相等。

该例类似于一个彩票游戏，先由程序随机选出 5 个 0~9 的数字，此过程类似于摇奖；再由用户手动输入 5 个猜测的数字，类似于购买彩票；分别把两组数字存入数组 a 和 b，然后逐一比对两个数组的元素；一旦有数字不一致，则告知用户猜测错误，只有当两个数组的所有元素都相等时，判定数组相等，即用户猜测正确。

```

#include <iostream>
#include <ctime>
#include <cstdlib>

```

```

using namespace std;

int main()
{
    const int sz = 5; // 常量 sz 作为数组的维度
    int a[sz], b[sz], i;
    srand( (unsigned) time (NULL) ); // 生成随机数种子
    // 通过 for 循环为数组元素赋值
    for(i = 0; i < sz; i++)
        // 每次循环生成一个 10 以内的随机数并添加到 a 中
        a[i] = rand() % 10;
    cout << "系统数据已经生成, 请输入您猜测的 5 个数字(0~9), 可以重复: " << endl;
    int uVal;
    // 通过 for 循环为数组元素赋值
    for(i = 0; i < sz; i++)
        if(cin >> uVal)
            b[i] = uVal;
    cout << "系统生成的数据是: " << endl;
    for(auto val : a)
        cout << val << " ";
    cout << endl;
    cout << "您猜测的数据是: " << endl;
    for(auto val : b)
        cout << val << " ";
    cout << endl;
    int *p = begin(a), *q = begin(b); // 令 p 和 q 分别指向数组 a 和 b 的首元素
    while(p != end(a) && q != end(b))
    {
        if(*p != *q)
        {
            cout << "您的猜测错误, 两个数组不相等" << endl;
            return -1;
        }
        p++; // p 向后移动一位
        q++; // q 向后移动一位
    }
    cout << "恭喜您全都猜对了! " << endl;
    return 0;
}

```

对比两个 vector 对象是否相等的程序如下所示, 其中使用迭代器遍历 vector 对象的元素。

```

#include <iostream>
#include <ctime>
#include <cstdlib>
#include <vector>

using namespace std;

int main()
{

```

```

const int sz = 5;                                // 常量 sz 作为 vector 的容量
int i;
vector<int> a, b;
srand( (unsigned) time (NULL) );    // 生成随机数种子
// 通过 for 循环为数组元素赋值
for(i = 0; i < sz; i++)
    // 每次循环生成一个 10 以内的随机数并添加到 a 中
    a.push_back(rand() % 10);
cout << "系统数据已经生成, 请输入您猜测的 5 个数字(0~9), 可以重复: " << endl;
int uVal;
// 通过 for 循环为数组元素赋值
for(i = 0; i < sz; i++)
    if(cin >> uVal)
        b.push_back(uVal);
cout << "系统生成的数据是: " << endl;
for(auto val : a)
    cout << val << " ";
cout << endl;
cout << "您猜测的数据是: " << endl;
for(auto val : b)
    cout << val << " ";
cout << endl;
// 令 it1, it2 分别指向 vector 对象 a 和 b 的首元素
auto it1 = a.cbegin(), it2 = b.cbegin();
while(it1 != a.cend() && it2 != b.cend())
{
    if(*it1 != *it2)
    {
        cout << "您的猜测错误, 两个 vector 不相等" << endl;
        return -1;
    }
    it1++;           // p 向后移动一位
    it2++;           // q 向后移动一位
}
cout << "恭喜您全都猜对了! " << endl;

return 0;
}

```

**练习 3.37:** 下面的程序是何含义, 程序输出结果是什么?

```

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

```

**【出题思路】**

考查 C 风格字符串和字符数组的关系, 尤其是串尾是否含有空字符的问题。C

风格字符串与标准库 `string` 对象既有联系又有区别。

### 【解答】

程序第一行声明了一个包含 5 个字符的字符数组，因为我们无须修改数组的内容，所以将其定义为常量。第二行定义了一个指向字符常量的指针，该指针可以指向不同的字符常量，但是不允许通过该指针修改所指常量的值

`while` 循环的条件是 `*cp`，只要指针 `cp` 所指的字符不是空字符'\0'，循环就重复执行，循环的任务有两项：首先输出指针当前所指的字符，然后将指针向后移动一位。

该程序的原意是输出 `ca` 中存储的 5 个字符，每个字符占一行，但实际的执行效果无法符合预期。因为以列表初始化方式赋值的 C 风格字符串与以字符串字面值赋值的有所区别，后者会在字符串最后额外增加一个空字符以示字符串的结束，而前者不会这样做。

因此在该程序中，`ca` 的 5 个字符全都输出后，并没有遇到预期的空字符，也就是说，`while` 循环的条件仍将满足，无法跳出。程序继续在内存中 `ca` 的存储位置之后挨个寻找空字符，直到找到为止。在这个过程中，额外经历的内容也将被输出出来，从而产生错误。

在作者的编译环境中，程序的输出结果是：

```
h
e
l
l
o
```

```
<
```

要想实现程序的原意，应该修改为：

```
const char ca[] = {'h', 'e', 'l', 'l', 'o', '\0'};
const char *cp = ca;
while (*cp) {
    cout << *cp << endl;
    ++cp;
}
```

或者修改为如下形式也能达到预期效果：

```
const char ca[] = "hello";
const char *cp = ca;
while (*cp) {
    cout << *cp << endl;
    ++cp;
}
```

**练习 3.38：**在本节中我们提到，将两个指针相加不但是非法的，而且也没什么意义。请问为什么两个指针相加没什么意义？

### 【出题思路】

与标准库 `vector` 类似, C++也为指针定义了一系列算术运算, 包括递增、递减、指针求差、指针与整数求和等, 但是并没有定义两个指针的求和运算。要想理解这一规定, 必须首先明白指针的含义。

### 【解答】

指针也是一个对象, 与指针相关的属性有 3 个, 分别是指针本身的值 (`value`)、指针所指的对象 (`content`) 以及指针本身在内存中的存储位置 (`address`)。它们的含义分别是:

指针本身的值是一个内存地址值, 表示指针所指对象在内存中的存储地址; 指针所指的对象可以通过解引用指针访问; 因为指针也是一个对象, 所以指针也存储在内存的某个位置, 它有自己的地址, 这也是为什么有“指针的指针”的原因。

通过上述分析我们知道, 指针的值是它所指对象的内存地址, 如果我们把两个指针加在一起, 就是试图把内存中两个对象的存储地址加在一起, 这显然是没有任何意义的。与之相反, 指针的减法是有意义的。如果两个指针指向同一个数组中的不同元素, 则它们相减的结果表征了它们所指的元素在数组中的距离。

**练习 3.39:** 编写一段程序, 比较两个 `string` 对象。再编写一段程序, 比较两个 C 风格字符串的内容。

### 【出题思路】

由于标准库 `string` 类定义了关系运算符, 所以比较两个 `string` 对象可以直接使用`<`、`>`、`==`等; 比较两个 C 风格字符串则必须使用 `cstring` 头文件中定义的 `strcmp` 函数。

### 【解答】

比较两个 `string` 对象的程序如下所示:

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    string str1, str2;
    cout << "请输入两个字符串: " << endl;
    cin >> str1 >> str2;

    if(str1 > str2)
        cout << "第一个字符串大于第二个字符串" << endl;
    else if(str1 < str2)
        cout << "第一个字符串小于第二个字符串" << endl;
    else
        cout << "两个字符串相等" << endl;
    return 0;
}
```

比较两个 C 风格字符串的程序如下所示，其中的分支部分选用了 switch-case 语句，其效果与上一个程序的 if-else 语句非常类似。

```
#include <iostream>
#include <cstring>

using namespace std;

int main()
{
    char str1[80], str2[80];
    cout << "请输入两个字符串: " << endl;
    cin >> str1 >> str2;
    // 利用cstring 头文件中定义的 strcmp 函数比较大小
    auto result = strcmp(str1, str2);
    switch(result)
    {
        case 1:
            cout << "第一个字符串大于第二个字符串" << endl;
            break;
        case -1:
            cout << "第一个字符串小于第二个字符串" << endl;
            break;
        case 0:
            cout << "两个字符串相等" << endl;
            break;
        default:
            cout << "未定义的结果" << endl;
            break;
    }

    return 0;
}
```

**练习 3.40：**编写一段程序，定义两个字符数组并用字符串字面值初始化它们；接着再定义一个字符数组存放前两个数组连接后的结果。使用 strcpy 和 strcat 把前两个数组的内容拷贝到第三个数组中。

#### 【出题思路】

C 风格字符串的操作函数定义在 cstring 头文件中。其中，strcpy 函数负责把字符串的内容拷贝给另一个字符串，strcat 函数则负责把字符串的内容拼接到另一个字符串之后。此外，strlen 函数用于计算字符串的长度。

需要注意的是，利用字符串字面值常量初始化 C 风格字符串时，默认在数组最后添加一个空字符，因此，strlen 的计算结果比字面值显示的字符数量多 1。为了细致起见，计算两个字符串拼接后的长字符串长度时，应该在两个字符串各自长度求和后减去 1，即减去 1 个多余空字符所占的额外空间。

#### 【解答】

满足题意的程序如下所示：

```

#include <iostream>
#include <cstring>

using namespace std;

int main()
{
    char str1[] = "Welcome to ";
    char str2[] = "C++ family!";
    // 利用 strlen 函数计算两个字符串的长度，并求得结果字符串的长度
    char result[strlen(str1) + strlen(str2) - 1];

    strcpy(result, str1); // 把第一个字符串拷贝到结果字符串中
    strcat(result, str2); // 把第二个字符串拼接到结果字符串中

    cout << "第一个字符串是: " << str1 << endl;
    cout << "第二个字符串是: " << str2 << endl;
    cout << "拼接后的字符串是: " << result << endl;
    return 0;
}

```

**练习 3.41:** 编写一段程序，用整型数组初始化一个 vector 对象。

#### 【出题思路】

C++不允许用一个数组初始化另一个数组，也不允许使用 vector 对象直接初始化数组，但是允许使用数组来初始化 vector 对象。要实现这一目的，只需要指明要拷贝区域的首元素地址和尾后地址。

#### 【解答】

满足题意的程序如下所示。使用随机数初始化数组，然后利用 begin 和 end 获得数组的范围。在用数组初始化 vector 对象时，只需要提供数组的元素区域。

```

#include <iostream>
#include <vector>
#include <ctime>
#include <cstdlib>

using namespace std;

int main()
{
    const int sz = 10; // 常量 sz 作为数组的维度
    int a[sz];
    srand( (unsigned) time (NULL)); // 生成随机数种子
    cout << "数组的内容是: " << endl;
    // 利用范围 for 循环遍历数组的每个元素
    for(auto &val : a)
    {
        val = rand()%100; // 生成一个 100 以内的随机数
        cout << val << " ";
    }
}

```

```

cout << endl;

// 利用 begin 和 end 初始化 vector 对象
vector<int> vInt(begin(a), end(a));
cout << "vector 的内容是: " << endl;
// 利用范围 for 循环遍历 vector 的每个元素
for(auto val : vInt)
{
    cout << val << " ";
}
cout << endl;
return 0;
}

```

**练习 3.42:** 编写一段程序，将含有整数元素的 vector 对象拷贝给一个整型数组。

### 【出题思路】

C++ 允许使用数组直接初始化 vector 对象，但是不允许使用 vector 对象初始化数组。如果想用 vector 对象初始化数组，则必须把 vector 对象的每个元素逐一赋值给数组。

### 【解答】

满足题意的程序如下所示：

```

#include <iostream>
#include <vector>
#include <ctime>
#include <cstdlib>

using namespace std;

int main()
{
    const int sz = 10; // 常量 sz 作为 vector 对象的容量
    vector<int> vInt;
    srand( (unsigned) time (NULL)); // 生成随机数种子
    cout << "vector 对象的内容是: " << endl;
    // 利用 for 循环遍历 vector 对象的每个元素
    for(int i = 0; i != sz; i++)
    {
        vInt.push_back(rand()%100); // 生成一个 100 以内的随机数
        cout << vInt[i] << " ";
    }
    cout << endl;

    auto it = vInt.cbegin();
    int a[vInt.size()];
    cout << "数组的内容是: " << endl;
    // 利用范围 for 循环遍历数组的每个元素
    for(auto &val : a)
    {
        val = *it;
    }
}

```

```

        cout << val << " ";
        it++;
    }
    cout << endl;
    return 0;
}

```

**练习 3.43:** 编写 3 个不同版本的程序，令其均能输出 ia 的元素。版本 1 使用范围 for 语句管理迭代过程；版本 2 和版本 3 都使用普通的 for 语句，其中版本 2 要求用下标运算符，版本 3 要求用指针。此外，在所有 3 个版本的程序中都要直接写出数据类型，而不能使用类型别名、auto 关键字或 decltype 关键字。

#### 【出题思路】

在不使用类型别名、auto 关键字和 decltype 关键字的前提下，必须在 for 循环或范围 for 循环的条件处准确写出循环变量的数据类型，这是本题的关键。

#### 【解答】

满足题意且不使用类型别名、auto 关键字和 decltype 关键字的程序如下所示：

```

#include <iostream>
using namespace std;
int main()
{
    int ia[3][4] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11};
    cout << "利用范围 for 语句输出多维数组的内容：" << endl;
    for(int (&row)[4] : ia)
    {
        for(int &col : row)
            cout << col << " ";
        cout << endl;
    }

    cout << "利用普通 for 语句和下标运算符输出多维数组的内容：" << endl;
    for(int i = 0; i != 3; i++)
    {
        for(int j = 0; j != 4; j++)
            cout << ia[i][j] << " ";
        cout << endl;
    }

    cout << "利用普通 for 语句和指针输出多维数组的内容：" << endl;
    for(int (*p)[4] = ia; p != ia + 3; p++)
    {
        for(int *q = *p; q != *p + 4; q++)
            cout << *q << " ";
        cout << endl;
    }
    return 0;
}

```

**练习 3.44:** 改写上个练习中的程序，使用类型别名来代替循环控制变量的类型。

### 【出题思路】

利用 C++11 新提供的类型别名声明代替循环控制变量的类型，使得外层循环变量的含义更加直观。

### 【解答】

满足题意的程序如下所示：

```
#include <iostream>

using namespace std;
using int_array = int[4];

int main()
{
    int ia[3][4] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11};
    cout << "利用范围 for 语句输出多维数组的内容：" << endl;
    for(int_array &row : ia)
    {
        for(int &col : row)
            cout << col << " ";
        cout << endl;
    }

    cout << "利用普通 for 语句和下标运算符输出多维数组的内容：" << endl;
    for(int i = 0; i != 3; i++)
    {
        for(int j = 0; j != 4; j++)
            cout << ia[i][j] << " ";
        cout << endl;
    }

    cout << "利用普通 for 语句和指针输出多维数组的内容：" << endl;
    for(int_array *p = ia; p != ia + 3; p++)
    {
        for(int *q = *p; q != *p + 4; q++)
            cout << *q << " ";
        cout << endl;
    }
    return 0;
}
```

**练习 3.45:** 再一次改写程序，这次使用 auto 关键字。

### 【出题思路】

进一步简化编写的程序，利用 C++11 新提供的 auto 关键字自动推断循环控制变量的类型，无须程序员再显式指定，使得程序更加简洁直观，极大降低了编写程序的难度。

### 【解答】

满足题意的程序如下所示：

```
#include <iostream>

using namespace std;

int main()
{
    int ia[3][4] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11};
    cout << "利用范围 for 语句输出多维数组的内容: " << endl;
    for(auto &row : ia)
    {
        for(auto &col : row)
            cout << col << " ";
        cout << endl;
    }

    cout << "利用普通 for 语句和下标运算符输出多维数组的内容: " << endl;
    for(auto i = 0; i != 3; i++)
    {
        for(auto j = 0; j != 4; j++)
            cout << ia[i][j] << " ";
        cout << endl;
    }

    cout << "利用普通 for 语句和指针输出多维数组的内容: " << endl;
    for(auto p = ia; p != ia + 3; p++)
    {
        for(auto q = *p; q != *p + 4; q++)
            cout << *q << " ";
        cout << endl;
    }
    return 0;
}
```

# 第4章 表达式

## 导读

C++语言提供了一套丰富的运算符，它们分别是：

- 算术运算符
- 逻辑运算符和关系运算符
- 赋值运算符
- 递增和递减运算符
- 成员访问运算符
- 条件运算符
- 位运算符
- sizeof 运算符
- 逗号运算符

学习 C++ 运算符和表达式的关键是掌握各种运算符的优先级关系以及它们各自满足哪种结合律。本章的练习题主要从优先级和结合律出发考查各种运算符的使用方法，在此基础上引导读者提高综合使用多种运算符的能力。本章最后介绍了隐式类型转换和显式类型转换。

**练习 4.1：** 表达式  $5+10*20/2$  的求值结果是多少？

### 【出题思路】

本题旨在考查几种常见算术运算符的优先级和结合律。

### 【解答】

在算术运算符中，乘法和除法的优先级相同，且均高于加减法的优先级。因此上式的计算结果应该是 105，在编程环境中很容易验证这一点。

**练习 4.2:** 根据 4.12 节中的表，在下述表达式的合理位置添加括号，使得添加括号后运算对象的组合顺序与添加括号前一致。

(a) `*vec.begin()`      (b) `*vec.begin() + 1`

### 【出题思路】

C++ 定义了各种各样的运算符，这些运算符根据运算意义的不同被划分成不同的优先级，本题意在让用户先区分出表达式中各个运算符的优先级关系，然后用括号的方式表示哪个运算符先执行、哪个运算符后执行。

### 【解答】

在本题涉及的运算符中，优先级最高的是成员选择运算符和函数调用运算符，其次是解引用运算符，最后是加法运算符。因此添加括号后的等价的式子是：

(a) `(*vec.begin())`      (b) `(*(vec.begin())) + 1`

下述程序可以用来验证我们的推断：

```
#include <iostream>
#include <vector>
#include <ctime>
#include <cstdlib>

using namespace std;

int main()
{
    vector<int> vec;
    srand( (unsigned) time( NULL ) );
    cout << "系统自动为向量生成一组元素....." << endl;
    for( int i = 0; i != 10; i++ )
        vec.push_back( rand() % 100 );
    cout << "生成的向量数据是：" << endl;
    for( auto c : vec )
        cout << c << " ";
    cout << endl;
    cout << "验证添加的括号是否正确：" << endl;
    cout << "*vec.begin() 的值是：" << *vec.begin() << endl;
    cout << "*(>vec.begin()) 的值是：" << *(vec.begin()) << endl;
    cout << "*vec.begin() + 1 的值是：" << *vec.begin() + 1 << endl;
    cout << "(*(>vec.begin())) + 1 的值是：" << (*(>vec.begin())) + 1 << endl;

    return 0;
}
```

**练习 4.3:** C++ 语言没有明确规定大多数二元运算符的求值顺序，给编译器优化留下了余地。这种策略实际上是在代码生成效率和程序潜在缺陷之间进行了权衡，你认为这可以接受吗？请说出你的理由。

### 【出题思路】

本题的关键是考查求值顺序对表达式求值过程的影响。

### 【解答】

正如题目所说，C++只规定了非常少的二元运算符（逻辑与运算符、逻辑或运算符、逗号运算符）的求值顺序，其他绝大多数二元运算符的求值顺序并没有明确规定。这样做提高了代码生成的效率，但是可能引发潜在的缺陷。

关键是缺陷的风险有多大？我们知道，对于没有指定执行顺序的运算符来说，如果表达式指向并修改了同一个对象，将会引发错误并产生未定义的行为；而如果运算对象彼此无关，它们既不会改变同一对象的状态也不执行IO任务，则函数的调用顺序不受限制。

就作者的观点而言，这样的做法在一定程度上是可以接受的，前提是在编写程序时注意以下两点：一是拿不准的时候最好用括号来强制让表达式的组合关系符合程序逻辑的要求；二是一旦改变了某个运算对象的值，在表达式的某地方就不要再使用这个运算对象了。

**练习 4.4：**在下面的表达式中添加括号，说明其求值的过程及最终结果。编写程序编译该（不加括号的）表达式并输出其结果验证之前的推断。

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

#### 【出题思路】

本题旨在考查算术运算符的优先级和结合律，乘法和除法的优先级相同，且均高于加减法的优先级；算术运算符都满足左结合律，意味着当优先级相同时按照从左向右的顺序进行结合。

#### 【解答】

添加括号之后的形式应该是 $((12 / 3) * 4) + (5 * 15) + ((24 \% 4) / 2)$ ，求值的过程是：首先计算 $12 / 3$ 得到4，接着 $4 * 4$ 得到16，同时计算 $5 * 15$ 得到75，计算 $24 \% 4$ 得到0，接着计算 $0 / 2$ 得到0，最后执行加法运算 $16 + 75 + 0$ 得到91。最终的计算结果是91。

**练习 4.5：**写出下列表达式的求值结果。

- |                        |                         |
|------------------------|-------------------------|
| (a) $-30 * 3 + 21 / 5$ | (b) $-30 + 3 * 21 / 5$  |
| (c) $30 / 3 * 21 \% 5$ | (d) $-30 / 3 * 21 \% 4$ |

#### 【出题思路】

本题主要考查乘法、除法、取余的计算规则。

#### 【解答】

本题用到两条典型的算术运算规则。

一是除法：整数相除结果还是整数，如果商含有小数部分，直接弃除。尤其当除法的两个运算对象的符号不同时，商为负，C++11 新标准规定商一律向0取整。

二是取余：如果取余的两个运算对象的符号不同，则负号所在的位置不同运算结果也不相同， $m \% (-n)$ 等于 $m \% n$ ， $(-m) \% n$ 等于 $-(m \% n)$ 。

因此，本题的求值结果是：

- (a)  $-30 * 3 + 21 / 5 = -86$
- (b)  $-30 + 3 * 21 / 5 = -18$
- (c)  $30 / 3 * 21 \% 5 = 0$
- (d)  $-30 / 3 * 21 \% 4 = -2$

**练习 4.6:** 写出一条表达式用于确定一个整数是奇数还是偶数。

#### 【出题思路】

根据奇数和偶数的定义可知，一个数能被 2 整除的数是偶数，不能被 2 整除的数是奇数。

#### 【解答】

下面的表达式可以用于确定一个整数是奇数还是偶数，假设该整数名为 num，则表达式  $num \% 2 == 0$  为真时 num 是偶数，该表达式为假时 num 是奇数。

**练习 4.7:** 溢出是何含义？写出三条将导致溢出的表达式。

#### 【出题思路】

当计算的结果超出类型所能表示的范围时，产生溢出。

#### 【解答】

溢出是一种常见的算术运算错误。因为在计算机中存储某种类型的内存空间有限，所以该类型的表示能力（范围）也是有限的，当计算的结果值超出这个范围时，就会产生未定义的数值，这种错误称为溢出。

假定编译器规定 int 占 32 位，则下面的 3 条表达式都将产生溢出错误：

```
int i = 2147483647 + 1;
int j = -100000 * 300000;
int k = 2015 * 2015 * 2015 * 2015;
```

在作者的编译环境中，i、j、k 的计算结果分别是-2147483648、64771072、1342568577，显然与我们的预期不相符，是溢出之后产生的错误结果。

**练习 4.8:** 说明在逻辑与、逻辑或及相等性运算符中运算对象求值的顺序。

#### 【出题思路】

逻辑与、逻辑或执行短路求值，相等性运算符则正常计算两个运算对象的值。

#### 【解答】

对于逻辑与运算符来说，当且仅当两个运算对象都为真时结果为真；对于逻辑或运算符来说，只要两个运算对象中的一个为真结果就为真。

逻辑与运算符和逻辑或运算符都是先求左侧运算对象的值再求右侧运算对象的值，当且仅当左侧运算对象无法确定表达式的结果时才会计算右侧运算对象的值。这种策略就是短路求值。其策略是：对于逻辑与运算符来说，当且仅当左侧运算对象为真时才计算右侧运算对象；对于逻辑或运算符来说，当且仅当左侧运算对象为假时才计算右侧运算对象。

值得注意的是，逻辑与运算符和逻辑或运算符是 C++ 中仅有的几个规定了求值顺序的运算符。相等性运算符的两个运算对象都要求值，C++ 没有规定其求值顺序。

### 练习 4.9：解释在下面的 if 语句中条件部分的判断过程。

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

#### 【出题思路】

本题旨在考查指针和解引用运算符作为 if 条件的用法。

#### 【解答】

`cp` 是指向字符串的指针，因此上式的条件部分含义是首先检查指针 `cp` 是否有效。如果 `cp` 为空指针或无效指针，则条件不满足。如果 `cp` 有效，即 `cp` 指向了内存中的某个有效地址，继续解引用指针 `cp` 并检查 `cp` 所指的对象是否为空字符'\'0'，如果 `cp` 所指的对象不是空字符则条件满足；否则不满足。

在本例中，显然初始状态下 `cp` 指向了字符串的首字符，是有效的；同时当前 `cp` 所指的对象是字符'H'，不是空字符，所以 if 的条件部分为真。

### 练习 4.10：为 while 循环写一个条件，使其从标准输入中读取整数，遇到 42 时停止。

#### 【出题思路】

综合运用逻辑与运算符及相等性运算符，注意在本题中利用了逻辑与运算符的短路求值特性，其左侧运算对象是为了确保右侧运算对象求值过程的正确性和安全性。

#### 【解答】

最简洁的形式是：

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

该语句首先检查从输入流读取数据是否正常，然后判断当前读入的数字是否是 42，遇到 42 则条件不满足，退出循环。还有一种形式也可以实现同样的目的：

```
int num;
while( cin >> num )
{
    if(num == 42)
        break;
    // 其他操作
}
```

### 练习 4.11：书写一条表达式用于测试 4 个值 `a`、`b`、`c`、`d` 的关系，确保 `a` 大于 `b`、`b` 大于 `c`、`c` 大于 `d`。

#### 【出题思路】

本题旨在考查关系运算符的用法。

**【解答】**

要想用一条表达式测试  $a > b && b > c && c > d$  的关系，并确保  $a$  大于  $b$ 、 $b$  大于  $c$ 、 $c$  大于  $d$ ，应该写成：

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

切勿写成：

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

因为关系运算符满足左结合律且运算的结果是布尔值，所以把几个关系运算符连写在一起必然会产生意想不到的结果。 $a > b > c > d$  的实际求值过程是先判断  $a > b$  是否成立，成立则为 1，不成立则为 0；接着用这个布尔值（1 或 0）与  $c$  比较，所得的结果仍然是一个布尔值；最后再用刚刚得到的布尔值与  $d$  进行比较。显然这一过程与用户的书写原意背道而驰。

**练习 4.12：**假设  $i$ 、 $j$  和  $k$  是三个整数，说明表达式  $i != j < k$  的含义。**【出题思路】**

需要明确两种关系运算符  $!=$  和  $<$  的优先级关系，这是解答本题的关键。

**【解答】**

C++ 规定  $<$ 、 $<=$ 、 $>$ 、 $>=$  的优先级高于  $==$  和  $!=$ ，因此上式的求值过程等同于  $i != (j < k)$ ，意即先比较  $j$  和  $k$  的大小，得到的结果是一个布尔值（1 或 0）；然后判断  $i$  的值与之是否相等。

**练习 4.13：**在下述语句中，当赋值完成后  $i$  和  $d$  的值分别是多少？

```
int i; double d;
(a) d = i = 3.5;      (b) i = d = 3.5;
```

**【出题思路】**

本题涉及的知识点有两个：第一，如果赋值运算符左右两个运算对象的类型不同，则右侧运算对象转换成左侧运算对象的类型；第二，赋值运算符满足右结合律。

**【解答】**

由题意可知，(a)式的含义是先把 3.5 赋值给整数  $i$ ，此时发生了自动类型转换，小数部分被舍弃， $i$  的值为 3；接着  $i$  的值再赋给双精度浮点数  $d$ ，所以  $d$  的值也是 3。

(b)式的含义是先把 3.5 赋值给双精度浮点数  $d$ ，因此  $d$  的值是 3.5；接着  $d$  的值再赋给整数  $i$ ，此时发生了自动类型转换，小数部分被舍弃， $i$  的值为 3。

**练习 4.14：**执行下述 if 语句后将发生什么情况？

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

**【出题思路】**

本题涉及的知识点有两个：第一，赋值运算符的左侧运算对象必须是左值，右

侧运算对象可以是左值，也可以是右值；第二，赋值运算符与相等性运算符在作为 if 语句的条件时含义不同。

### 【解答】

第一条语句发生编译错误，因为赋值运算符的左侧运算对象必须是左值，字面值常量 42 显然不是左值，不能作为左侧运算对象。

第二条语句从语法上来说是正确的，但是与程序的原意不符。程序的原意是判断 i 的值是否是 42，应该写成 `i==42`；而 `i=42` 的意思是把 42 赋值给 i，然后判断 i 的值是否为真。因为所有非 0 整数转换成布尔值时都对应 `true`，所以该条件是恒为真的。

### 练习 4.15：下面的赋值是非法的，为什么？应该如何修改？

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

### 【出题思路】

赋值运算符满足右结合律，自右向左分析赋值操作的含义。

### 【解答】

该赋值语句是非法的，虽然连续赋值的形式本身并没有错，但是参与赋值的几个变量类型不同。其中，`dval` 是双精度浮点数，`ival` 是整数，`pi` 是整型指针。

自右向左分析赋值操作的含义，`pi=0` 表示 `pi` 是一个空指针，接下来 `ival=pi` 试图把整型指针的值赋给整数，这是不符合语法规范的操作，无法编译通过。稍作调整，就可以把上述程序改为合法。

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

### 练习 4.16：尽管下面的语句合法，但它们实际执行的行为可能和预期并不一样，为什么？应该如何修改？

(a) `if (p = getPtr() != 0)`      (b) `if (i = 1024)`

### 【出题思路】

本题考查运算符优先级及赋值表达式作为 if 条件的含义。

### 【解答】

(a) 的原意是把 `getPtr()` 得到的指针赋值给 `p`，然后判断 `p` 是否是一个空指针，但上述表达式的实际执行结果与之相距甚远。因为赋值运算符的优先级低于不相等运算符，所以真正的表达式求值过程是先判断 `getPtr()` 的返回值是否为空指针，如果是则 `p=0`，否则 `p=1`，最后以 `p` 的值作为 if 语句的条件。要想符合原意，应该修改为：

```
if ((p = getPtr()) != 0)
```

(b) 的原意是判断 `i` 的值是否是 1024，但上述表达式实际上是把 1024 赋值给 `i`，

然后以 `i` 作为 `if` 语句的条件。因为所有非 0 整数转换成布尔值时都对应 `true`，所以该条件是恒为真的。

### 练习 4.17：说明前置递增运算符和后置递增运算符的区别。

#### 【出题思路】

C++实现了两种递增（递减）运算符：即前置版本和后置版本，二者的工作机理有所区别，一般来说前置版本是更好的选择。

#### 【解答】

递增和递减运算符有两种形式：前置版本和后置版本。前置版本首先将运算对象加 1（或减 1），然后把改变后的对象作为求值结果。后置版本也将运算对象加 1（或减 1），但是求值结果是运算对象改变之前那个值的副本。这两种运算符必须作用于左值运算对象。前置版本将对象本身作为左值返回，后置版本则将对象原始值的副本作为右值返回。

我们的建议是，除非必须，否则不用递增（递减）运算符的后置版本。前置版本的递增运算符避免了不必要的工作，它把值加 1 后直接返回改变了的运算对象。与之相比，后置版本需要将原始值存储下来以便于返回这个未修改的内容。如果我们不需要修改之前的值，那么后置版本的操作就是一种浪费。

对于整数和指针类型来说，编译器可能对这种额外的工作进行了一定的优化；但是对于相对复杂的迭代器类型来说，这种额外的工作就消耗巨大了。建议养成使用前置版本的习惯，这样不仅不需要担心性能问题，而且更重要的是写出的代码会更符合编程人员的初衷。

### 练习 4.18：如果第 132 页那个输出 `vector` 对象元素的 `while` 循环使用前置递增运算符，将得到什么结果？

#### 【出题思路】

前置递增运算符先将运算对象加 1，然后把改变后的对象作为求值结果；后置递增运算符也将运算对象加 1，但是求值结果是运算对象改变之前那个值的副本。

简言之，如果在一条表达式中出现了递增运算符，则其计算规律是：`++`在前，先加 1，后参与运算；`++`在后，先参与运算，后加 1。

#### 【解答】

基于上述分析，本题不应该把 `while` 循环的后置递增运算符改为前置递增运算符。如果这样做了，会产生两个错误结果：一是无法输出 `vector` 对象的第一个元素；二是当所有元素都不为负时，移动到最后一个元素的地方，程序试图继续向前移动迭代器并解引用一个根本不存在的元素。

**练习 4.19:**假设 `ptr` 的类型是指向 `int` 的指针、`vec` 的类型是 `vector<int>`、`ival` 的类型是 `int`，说明下面的表达式是何含义？如果有表达式不正确，为什么？应该如何修改？

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

#### 【出题思路】

本题旨在考查递增运算符与关系运算符、逻辑运算符、解引用运算符的优先级关系，读者也需要理解后置递增运算符的计算规则。

#### 【解答】

(a)的含义是先判定指针 `ptr` 是否为空，如果不为空，继续判断指针 `ptr` 所指的整数是否为非 0 数。如果非 0，则该表达式的最终求值结果为真；否则为假。最后把指针 `ptr` 向后移动一位。该表达式从语法上分析是合法的，但是最后的指针移位操作不一定有意义。如果 `ptr` 所指的是整型数组中的某个元素，则 `ptr` 可以按照预期移动到下一个元素。如果 `ptr` 所指的只是一个独立的整数变量，则移动指针操作将产生未定义的结果。

(b)的含义是先检查 `ival` 的值是否非 0，如果非 0 继续检查 `(ival+1)` 的值是否非 0。只有当两个值都是非 0 值时，表达式的求值结果为真；否则为假。在 4.1.3 节中我们学习到，如果二元运算符的两个运算对象涉及同一个对象并改变对象的值，则这是一种不好的程序写法，应该改写。所以按照程序的原意，本式应该改写成 `ival && (ival + 1)`。

(c)的含义是比较 `vec[ival]` 和 `vec[ival + 1]` 的大小，如果前者较小则求值结果为真，否则为假。与(b)式一样，本式也出现了二元运算符的两个运算对象涉及同一个对象并改变对象值的情况，应该改写为 `vec[ival] <= vec[ival + 1]`。

**练习 4.20:**假设 `iter` 的类型是 `vector<string>::iterator`，说明下面的表达式是否合法。如果合法，表达式的含义是什么？如果不合法，错在何处？

- |                                    |                             |                                      |
|------------------------------------|-----------------------------|--------------------------------------|
| (a) <code>*iter++;</code>          | (b) <code>(*iter)++;</code> | (c) <code>*iter.empty();</code>      |
| (d) <code>iter-&gt;empty();</code> | (e) <code>++*iter;</code>   | (f) <code>iter++-&gt;empty();</code> |

#### 【出题思路】

本题旨在考查成员访问运算符与递增运算符和解引用运算符的优先级关系。

#### 【解答】

(a)是合法的，后置递增运算符的优先级高于解引用运算符，其含义是解引用当前迭代器所处位置的对象内容，然后把迭代器的位置向后移动一位。

(b)是非法的，解引用 `iter` 得到 `vector` 对象当前的元素，结果是一个 `string`，显然 `string` 没有后置递增操作。

(c)是非法的，解引用运算符的优先级低于点运算符，所以该式先计算 `iter.empty()`，而迭代器并没有定义 `empty` 函数，所以无法通过编译。

(d)是合法的，`iter->empty();` 等价于 `(*iter).empty();`。解引用迭代器

得到迭代器当前所指的元素，结果是一个 `string`，显然字符串可以判断是否为空，`empty` 函数在此处有效。

(e) 是非法的，该式先解引用 `iter`，得到迭代器当前所指的元素，结果是一个 `string`，显然 `string` 没有后置递增操作。

(f) 是合法的，`iter++->empty()` 等价于 `(*iter++) .empty()`。含义是解引用迭代器当前位置的对象内容，得到一个字符串，判断该字符串是否为空，然后把迭代器向后移动一位。

**练习 4.21：**编写一段程序，使用条件运算符从 `vector<int>` 中找到哪些元素的值是奇数，然后将这些奇数值翻倍。

### 【出题思路】

条件运算符使得我们可以把简单的 `if-else` 结构嵌入到表达式中，条件表达式可以作为赋值运算符的右侧运算对象。

### 【解答】

满足题意的程序如下所示：

```
#include <iostream>
#include <vector>
#include <ctime>
#include <cstdlib>

using namespace std;

int main()
{
    vector<int> vInt;
    const int sz = 10; // 使用 sz 作为数组的维度
    srand( (unsigned) time (NULL)); // 生成随机数种子
    // 使用普通 for 循环为数组赋初值
    cout << "数组的初始值是：" << endl;
    for(int i = 0; i != sz; ++i)
    {
        vInt.push_back(rand() % 100); // 生成 100 以内的随机数
        cout << vInt[i] << " "; // 使用下标运算符输出数组内容
    }
    cout << endl;
    // 使用范围 for 循环把数组中的奇数翻倍
    for(auto &val : vInt)
        val = (val % 2 != 0) ? val * 2 : val; // 条件表达式
    // 使用范围 for 循环和迭代器输出数组的当前值
    cout << "调整后的数组值是：" << endl;
    for(auto it = vInt.cbegin(); it != vInt.cend(); ++it)
        cout << *it << " ";
    cout << endl;
    return 0;
}
```

**练习 4.22：**本节的示例程序将成绩划分成 high pass、pass 和 fail 三种，扩展该程序使其进一步将 60 分到 75 分之间的成绩设定为 low pass。要求程序包含两个版本：一个版本只使用条件运算符；另外一个版本使用 1 个或多个 if 语句。哪个版本的程序更容易理解呢？为什么？

### 【出题思路】

条件表达式的作用与 if-else 语句类似。条件表达式可以嵌套，但是嵌套的层数越少越好，如果嵌套层数过多，则代码的可读性会变得非常差。

### 【解答】

使用条件运算符实现的程序如下所示：

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    string finalgrade;
    int grade;
    cout << "请输入您要检查的成绩：" << endl;
    // 确保输入的成绩合法
    while(cin >> grade && grade >= 0 && grade <= 100)
    {
        // 使用三层嵌套的条件表达式
        finalgrade = (grade > 90) ? "high pass"
            : (grade > 75) ? "pass"
            : (grade > 60) ? "low pass" : "fail";
        cout << "该成绩所处的档次是：" << finalgrade << endl;
        cout << "请输入您要检查的成绩：" << endl;
    }

    return 0;
}
```

使用 if 语句实现的程序如下所示：

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    string finalgrade;
    int grade;
    cout << "请输入您要检查的成绩：" << endl;
    // 确保输入的成绩合法
    while(cin >> grade && grade >= 0 && grade <= 100)
    {
        // 使用 if 语句实现
        if(grade > 90)
            finalgrade = "high pass";
```

```

        else if(grade > 75)
            finalgrade = "pass";
        else if(grade > 60)
            finalgrade = "low pass";
        else
            finalgrade = "fail";
        cout << "该成绩所处的档次是：" << finalgrade << endl;
        cout << "请输入您要检查的成绩：" << endl;
    }

    return 0;
}

```

**练习 4.23:** 因为运算符的优先级问题，下面这条表达式无法通过编译。根据 4.12 节中的表（第 147 页）指出它的问题在哪里？应该如何修改？

```

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

```

#### 【出题思路】

条件运算符的优先级非常低，因此在使用条件运算符构成复合表达式时必须在适当的地方添加括号。

#### 【解答】

题目中的几个运算符的优先级次序从高到低是加法运算符、相等运算符、条件运算符和赋值运算符，因此式子的求值过程是先把 s 和 s[s.size() - 1] 相加得到一个新字符串，然后该字符串与字符's'比较是否相等，这是一个非法操作，并且与程序的原意不符。

要想实现程序的原意，即先判断字符串 s 的最后一个字符是否是's'，如果是，什么也不做；如果不是，在 s 的末尾添加一个字符's'，我们应该添加括号强制限定运算符的执行顺序。

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

**练习 4.24:** 本节的示例程序将成绩划分成 high pass、pass 和 fail 三种，它的依据是条件运算符满足右结合律。假如条件运算符满足的是左结合律，求值过程将是怎样的？

#### 【出题思路】

假设条件运算符满足的是左结合律，即嵌套的条件运算符从左向右求值。

#### 【解答】

原文的程序是：

```
finalgrade = (grade > 90) ? "high pass"
                      : (grade < 60) ? "fail" : "pass";
```

根据左结合律的含义，该式等价于：

```
finalgrade = ((grade > 90) ? "high pass" : (grade < 60))
? "fail" : "pass";
```

先考查 `grade > 90` 是否成立，如果成立，第一个条件表达式的值为 "high pass"；如果不成立，第一个条件表达式的值为 `grade < 60`。这条语句是无法编译通过的，因为条件运算符要求两个结果表达式的类型相同或者可以互相转化。即使假设语法上通过，也就是说，第一个条件表达式的求值结果分为3种，分别是 "high pass"、1和0。接下来根据第一个条件表达式的值求解第二个条件表达式，求值结果是 "fail" 或 "pass"。上述求值过程显然与我们的期望是不符的。

**练习 4.25:** 如果一台机器上 `int` 占 32 位、`char` 占 8 位，用的是 Latin-1 字符集，其中字符'q'的二进制形式是 01110001，那么表达式~'q'<<6 的值是什么？

#### 【出题思路】

本题旨在考查位运算符的优先级和计算规则。

#### 【解答】

在位运算符中，运算符~的优先级高于<<，因此先对 q 按位求反，因为位运算符的运算对象应该是整数类型，所以字符'q'首先转换为整数类型。如题所示，`char` 占 8 位而 `int` 占 32 位，所以字符'q'转换后得到 00000000 00000000 00000000 01110001，按位求反得到 11111111 11111111 11111111 10001110；接着执行移位操作，得到 11111111 11111111 11100011 10000000。

C++ 规定整数按照其补码形式存储，对上式求补，得到 10000000 00000000 00011100 10000000，即最终结果的二进制形式，转换成十进制形式是 -7296。

**练习 4.26:** 在本节关于测验成绩的例子中，如果使用 `unsigned int` 作为 `quiz1` 的类型会发生什么情况？

#### 【出题思路】

结合本题的题意，选择适当的数据类型。

#### 【解答】

根据题目的要求，班级中有 30 个学生，我们用一个二进制位代表某个学生在一次测验中是否通过，则原书中使用 `unsigned long` 作为 `quiz1` 的数据类型是恰当的，因为 C++ 规定 `unsigned long` 在内存中至少占 32 位，这样就足够存放 30 个学生的信息。

如果使用 `unsigned int` 作为 `quiz1` 的类型，则由于 C++ 规定 `unsigned int` 所占空间的最小值是 16，所以在很多机器环境中，该数据类型不足以存放全部学生的信息，从而造成了信息丢失，无法完成题目要求的任务。

**练习 4.27:** 下列表达式的结果是什么？

- ```
unsigned long ull = 3, ul2 = 7;
(a) ull & ul2           (b) ull | ul2
(c) ull && ul2         (d) ull || ul2
```

**【出题思路】**

读者需要掌握按位与和按位或的计算方法，同时注意区分按位与和逻辑与、按位或和逻辑或。

**【解答】**

`u11` 转换为二进制形式是：00000000 00000000 00000000 00000011，`u12` 转换为二进制形式是：00000000 00000000 00000000 00000111。各个式子的求值结果分别是：

(a)按位与，结果是：00000000 00000000 00000000 00000011，即 3。

(b)按位或，结果是：00000000 00000000 00000000 00000111，即 7。

(c)逻辑与，所有非 0 整数对应的布尔值都是 `true`，所以该式等价于 `true && true`，结果为 `true`。

(d)逻辑或，所有非 0 整数对应的布尔值都是 `true`，所以该式等价于 `true || true`，结果为 `true`。

**练习 4.28：编写一段程序，输出每一种内置类型所占空间的大小。****【出题思路】**

运用 `sizeof` 运算符输出内置类型的大小，注意 C++ 只规定了每种类型所占的最小空间，类型实际占多少空间依赖于具体实现。

**【解答】**

满足题意的程序如下所示：

```
#include <iostream>

using namespace std;

int main()
{
    cout << "类型名称\t" << "所占空间" << endl;
    cout << "bool\t\t" << sizeof(bool) << endl;
    cout << "char\t\t" << sizeof(char) << endl;
    cout << "wchar_t\t" << sizeof(wchar_t) << endl;
    cout << "char16_t\t" << sizeof(char16_t) << endl;
    cout << "char32_t\t" << sizeof(char32_t) << endl;
    cout << "short\t\t" << sizeof(short) << endl;
    cout << "int\t\t" << sizeof(int) << endl;
    cout << "long\t\t" << sizeof(long) << endl;
    cout << "long long\t" << sizeof(long long) << endl;
    cout << "float\t\t" << sizeof(float) << endl;
    cout << "double\t\t" << sizeof(double) << endl;
    cout << "long double\t" << sizeof(long double) << endl;
    return 0;
}
```

在作者的编译环境中，输出结果是：

类型名称	所占空间
<code>bool</code>	1
<code>char</code>	1

wchar_t	2
char16_t	2
char32_t	4
short	2
int	4
long	4
long long	8
float	4
double	8
long double	12

我们可以看到，`sizeof` 运算符的求值结果是类型在内存中所占的字节数。正常情况下，求值结果应该至少等于 C++ 规定的最小值，依赖于机器的不同，部分类型实际占用的空间会大于这个最小值。例如，在作者的编译环境中，`int` 占 4 字节，超过了 C++ 规定的 2 字节。

**练习 4.29：**推断下面代码的输出结果并说明理由。实际运行这段程序，结果和你想象的一样吗？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;
```

#### 【出题思路】

当 `sizeof` 的运算对象是数组名、数组内容、指针时，读者应该了解其区别。

#### 【解答】

`sizeof(x)` 的运算对象 `x` 是数组的名字，求值结果是整个数组所占空间的大小，等价于对数组中所有的元素各执行一次 `sizeof` 运算并对所得结果求和。读者尤其需要注意，`sizeof` 运算符不会把数组转换成指针来处理。在本例中，`x` 是一个 `int` 数组且包含 10 个元素，所以 `sizeof(x)` 的求值结果是 10 个 `int` 值所占的内存空间总和。

`sizeof(*x)` 的运算对象 `*x` 是一条解引用表达式，此处的 `x` 既是数组的名称，也表示指向数组首元素的指针，解引用该指针得到指针所指的内容，在本例中是一个 `int`。所以 `sizeof(*x)` 在这里等价于 `sizeof(int)`，即 `int` 所占的内存空间。

`sizeof(x)/sizeof(*x)` 可以理解为数组 `x` 所占的全部空间除以其中一个元素所占的空间，得到的结果应该是数组 `x` 的元素总数。实际上，因为 C++ 的内置数组并没有定义成员函数 `size()`，所以通常无法直接得到数组的容量。本题所示的方法是计算得到数组容量的一种常规方法。

`sizeof(p)` 的运算对象 `p` 是一个指针，求值结果是指针所占的空间大小。

`sizeof(*p)` 的运算对象 `*p` 是指针 `p` 所指的对象，即 `int` 变量 `x`，所以求值结果是 `int` 值所占的空间大小。

在作者的编译环境中，`int` 占 4 字节，指针也占 4 字节，所以本题程序的输出结果是：

```
10
1
```

**练习 4.30:** 根据 4.12 节中的表( 第 147 页 ), 在下述表达式的适当位置加上括号, 使得加上括号之后表达式的含义与原来的含义相同。

- |                                  |                                      |
|----------------------------------|--------------------------------------|
| (a) <code>sizeof x + y</code>    | (b) <code>sizeof p-&gt;mem[i]</code> |
| (c) <code>sizeof a &lt; b</code> | (d) <code>sizeof f()</code>          |

#### 【出题思路】

本题考查 `sizeof` 运算符与其他运算符的优先级关系。

#### 【解答】

(a) 的含义是先求变量 `x` 所占空间的大小, 然后与变量 `y` 的值相加; 因为 `sizeof` 运算符的优先级高于加法运算符的优先级, 所以如果想求表达式 `x+y` 所占的内存空间, 应该改为 `sizeof(x + y)`。

(b) 的含义是先定位到指针 `p` 所指向的对象, 然后求该对象中名为 `mem` 的数组成员第 `i` 个元素的尺寸。因为成员选择运算符的优先级高于 `sizeof` 的优先级, 所以本例无须添加括号。

(c) 的含义是先求变量 `a` 在内存中所占空间的大小, 再把求得的值与变量 `b` 的值比较。因为 `sizeof` 运算符的优先级高于关系运算符的优先级, 所以如果想求表达式 `a<b` 所占的内存空间, 应该改为 `sizeof(a < b)`。

(d) 的含义是求函数 `f()` 返回值所占内存空间的大小, 因为函数调用运算符的优先级高于 `sizeof` 的优先级, 所以本例无须添加括号。

**练习 4.31:** 本节的程序使用了前置版本的递增运算符和递减运算符, 解释为什么要用前置版本而不用后置版本。要想使用后置版本的递增递减运算符需要做哪些改动? 使用后置版本重写本节的程序。

#### 【出题思路】

读者需要掌握前置版本和后置版本的联系和区别。

#### 【解答】

本题从程序运行结果来说, 使用前置版本或后置版本是一样的, 这是因为递增递减运算符与真正使用这两个变量的语句位于不同的表达式中, 所以不会有什么影响。

使用后置版本重写的程序是:

```
vector<int>::size_type cnt = ivec.size();
// 将从 size 到 1 的值赋给 ivec 的元素。
for(vector<int>::size_type ix = 0; ix != ivec.size(); ix++, cnt--)
    ivec[ix] = cnt;
```

根据 4.5 节的介绍我们知道, 除非必须, 否则不用递增(递减)运算符的后置版本。前置版本的递增运算符避免了不必要的工作, 它把值加 1 后直接返回改变了的运算对象。与之相比, 后置版本需要将原始值存储下来以便于返回这个未修改的内容。如果我们不需要修改之前的值, 那么后置版本的操作就是一种浪费。

就本题而言, 使用前置版本是更好的选择。

**练习 4.32:** 解释下面这个循环的含义。

```
constexpr int size = 5;
int ia[size] = {1,2,3,4,5};
for (int *ptr = ia, ix = 0;
     ix != size && ptr != ia+size;
     ++ix, ++ptr) { /* ... */ }
```

**【出题思路】**

考查逗号运算符的计算规则。

**【解答】**

首先定义一个常量表达式 `size`, 它的值是 5; 接着以 `size` 作为维度创建一个整型数组 `ia`, 5 个元素分别是 1~5。

`for` 语句头包括三部分: 第一部分定义整型指针指向数组 `ia` 的首元素, 并且定义了一个整数 `ix`, 赋给它初值 0; 第二部分判断循环终止的条件, 当 `ix` 没有达到 `size` 同时指针 `ptr` 没有指向数组最后一个元素的下一位置时, 执行循环体; 第三部分令变量 `ix` 和指针 `ptr` 分别执行递增操作。

**练习 4.33:** 根据 4.12 节中的表 (第 147 页) 说明下面这条表达式的含义。

```
someValue ? ++x, ++y : --x, --y
```

**【出题思路】**

本题的关键是理解条件运算符和逗号运算符的优先级关系。

**【解答】**

C++ 规定条件运算符的优先级高于逗号运算符, 所以 `someValue ? ++x, ++y : --x, --y` 实际上等价于 `(someValue ? ++x, ++y : --x), --y`。它的求值过程是, 首先判断 `someValue` 是否为真, 如果为真, 依次执行 `++x` 和 `++y`, 最后执行 `--y`; 如果为假, 执行 `--x` 和 `--y`。

下面是检验上述推断的示例程序:

```
#include <iostream>

using namespace std;

int main()
{
    int x = 10, y = 20;
    // 检验条件为真的情况
    bool someValue = true;
    someValue ? ++x, ++y : --x, --y;
    cout << x << endl;
    cout << y << endl;
    cout << someValue << endl;

    x = 10, y = 20;
    // 检验条件为假的情况
    someValue = false;
    someValue ? ++x, ++y : --x, --y;
```

```

cout << x << endl;
cout << y << endl;
cout << someValue << endl;

return 0;
}

```

在作者的编译环境中，程序的运行结果是：

```

11
20
1
9
19
0

```

当 `someValue` 取值为 `true` 时，依次执行`++x`、`++y`、`--y`，也就是说，`x` 的值加 1 变为 11，`y` 的值先加 1 后减 1 保持不变，还是 20。

当 `someValue` 取值为 `false` 时，依次执行`--x`、`--y`，`x` 和 `y` 的值各减少 1 变为 9 和 19。

**练习 4.34：**根据本节给出的变量定义，说明在下面的表达式中将发生什么样的类型转换。

- (a) if (fval) (b) dval = fval + ival; (c) dval + ival \* cval;
- 需要注意每种运算符遵循的是左结合律还是右结合律。

#### 【出题思路】

读者需要掌握算术转换的规则。

#### 【解答】

(a)if 语句的条件应该是布尔值，因此 `float` 型变量 `fval` 自动转换成布尔值，转换规则是所有非 0 值转换为 `true`，0 转换为 `false`。

(b)ival 转换成 `float`，与 `fval` 求和后所得的结果进一步转换为 `double` 类型。

(c)cval 执行整型提升转换为 `int`，与 `ival` 相乘后所得的结果转换为 `double` 类型，最后再与 `dval` 相加。

**练习 4.35：**假设有如下的定义，

```

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

```

请回答，在下面的表达式中发生了隐式类型转换吗？如果有，指出来。

- (a) `cval = 'a' + 3;` (b) `fval = ui - ival * 1.0;`
- (c) `dval = ui * fval;` (d) `cval = ival + fval + dval;`

#### 【出题思路】

读者需要掌握算术转换的规则。

#### 【解答】

(a)字符'a'提升为 `int`，与 3 相加所得的结果再转换为 `char` 并赋给 `cval`。

(b)ival 转换为 `double`，与 1.0 相乘的结果也是 `double` 类型，ui 转换为

`double` 类型后与乘法得到的结果相减，最终的结果转换为 `float` 并赋给 `fval`。

(c) `i` 转换为 `float`，与 `fval` 相乘的结果转换为 `double` 类型并赋给 `dval`。

(d) `ival` 转换为 `float`，与 `fval` 相加所得的结果转换为 `double` 类型，再与 `dval` 相加后结果转换为 `char` 类型。

**练习 4.36：**假设 `i` 是 `int` 类型、`d` 是 `double` 类型，书写表达式 `i*=d` 使其执行整数类型的乘法而非浮点类型的乘法。

#### 【出题思路】

任何具有明确定义的类型转换，只要不包括底层 `const`，都可以使用 `static_cast`。

#### 【解答】

使用 `static_cast` 把 `double` 类型的变量 `d` 强制转换成 `int` 类型，就可以令表达式 `i*=d` 执行整数类型的乘法。语句的形式应该是：

```
i *= static_cast<int>(d);
```

**练习 4.37：**用命名的强制类型转换改写下列旧式的转换语句。

```
int i; double d; const string *ps; char *pc; void *pv;
(a) pv = (void*)ps;          (b) i = int(*pc);
(c) pv = &d;                 (d) pc = (char*) pv;
```

#### 【出题思路】

利用 `static_cast` 执行强制类型转换，对于底层 `const` 则使用 `const_cast`。

#### 【解答】

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

**练习 4.38：**说明下面这条表达式的含义。

```
double slope = static_cast<double>(j/i);
```

#### 【出题思路】

理解命名的强制类型转换。

#### 【解答】

把 `j/i` 的值强制类型转换成 `double`，然后赋值给 `slope`。请注意，如果 `i` 和 `j` 的类型都是 `int`，则 `j/i` 的求值结果仍然是 `int`，即使除不尽也只保留商的整数部分，最后再转换成 `double` 类型。

# 第 5 章

# 语句

## 导读

本章介绍了几种典型的 C++ 语句，它们分别是：

- 空语句和语句块
- 条件语句 (`if`、`switch`)
- 迭代语句 (`while`、`do-while`、`for`、范围 `for`)
- 跳转语句 (`break`、`continue`、`goto`)

此外还介绍了与语句密切相关的知识：语句的作用域和异常处理机制。本章的练习包含两种题型：一种是考查读者对本章介绍知识的掌握程度，尤其是对核心概念的理解是否准确；另一种是基于所学的语句语法规则，编写解决实际问题的程序。

对于同一类语句来说，不同的语法形式常常可以互相转换。读者应该理解并练习 `if` 语句和 `switch` 语句如何互相转换；`while` 语句、`do-while` 语句和 `for` 语句如何互相转换。

---

**练习 5.1：**什么是空语句？什么时候会用到空语句？

**【出题思路】**

理解空语句的形式和用法。

**【解答】**

空语句是最简单的语句，空语句由一个单独的分号构成。如果在程序的某个地方，语法上需要一条语句但是逻辑上不需要，此时应该使用空语句，空语句什么也不做。

一种常见的情况是，当循环的全部工作在条件部分就可以完成时，我们通常会用到空语句。使用空语句时最好加上注释，从而令代码的阅读者知道这条语句是有意省略内容的。

**练习 5.2：**什么是块？什么时候会用到块？**【出题思路】**

理解块的形式和用法。

**【解答】**

块是指用花括号括起来的语句和声明的序列，也称为复合语句。一个块就是一个作用域，在块中引入的名字只能在块内部以及嵌套在块中的子块里访问。如果在程序的某个地方，语法上需要一条语句，但是逻辑上需要多条语句，此时应该使用块。块不需要以分号结束。

例如，循环体必须是一条语句，但是我们通常需要在循环体内做很多事情，此时就应该把多条语句用花括号括起来，从而把语句序列转变成块。

**练习 5.3：**使用逗号运算符（参见 4.10 节，第 140 页）重写 1.4.1 节（第 10 页）的 while 循环，使它不再需要块，观察改写之后的代码的可读性提高了还是降低了。**【出题思路】**

使用连续的逗号运算符可以把多条语句合并为一条，这一点与块的作用类似。但是一般来说，直接使用块在程序的可读性上更有优势。

**【解答】**

原文的 while 循环使用了块，其形式是：

```
while ( val <= 10 )
{
    sum += val;
    ++val;
}
```

利用逗号运算符改写之后的形式如下所示：

```
while ( val <= 10 )
    sum += val, ++val;
```

很明显，改写之后的代码不够清晰，可读性降低了。

**练习 5.4：**说明下列例子的含义，如果存在问题，试着修改它。

- (a)    `while (string::iterator iter != s.end()) { /* ... */ }`
- (b)    `while (bool status = find(word)) { /* ... */ }`  
`if (!status) { /* ... */ }`

**【出题思路】**

我们可以在 if、switch、while 和 for 语句的控制结构内定义变量。定义在控制结构当中的变量只在相应语句的内部可见，一旦语句结束，变量也就超出其作用范围了。如果其他代码也需要访问控制变量，则变量必须定义在语句的外部。

**【解答】**

- (a) 是非法的，它的原意是希望在 while 语句的控制结构当中定义一个

`string::iterator`类型的变量`iter`, 然后判断`iter`是否到达了`s`的末尾, 只要还没有到达末尾就执行循环体的内容。但是该式把变量的定义和关系判断混合在了一起, 如果要使用`iter`与其他值比较, 必须首先为`iter`赋初值。修改后的程序应该是:

```
string::iterator iter=s.begin();
while (iter != s.end())
{
    ++iter;
    /*...*/
}
```

(b)是非法的, 变量`status`定义在`while`循环控制结构的内部, 其作用域仅限于`while`循环。`if`语句已经位于`while`循环的作用域之外, `status`在`if`语句内是一个未命名的无效变量。要想在`if`语句中继续使用`status`, 需要把它定义在`while`循环之前。修改后的程序应该是:

```
bool status;
while (status = find(word)) { /* ... */ }
if (!status) { /* ... */ }
```

**练习 5.5:** 写一段自己的程序, 使用`if-else`语句实现把数字成绩转换成字母成绩的要求。

### 【出题思路】

练习`if-else`语句的基本语法结构。

### 【解答】

满足题意的程序如下所示:

```
#include <iostream>

using namespace std;

int main()
{
    int grade;
    cout << "请输入您的成绩: ";
    cin >> grade;
    if(grade < 0 || grade > 100)
    {
        cout << "该成绩不合法" << endl;
        return -1;
    }
    if(grade == 100) // 处理满分的情况
    {
        cout << "等级成绩是: " << "A++" << endl;
        return -1;
    }
    if(grade < 60) // 处理不及格的情况
    {
        cout << "等级成绩是: " << "F" << endl;
    }
}
```

```

        return -1;
    }
    int iU = grade / 10;      // 成绩的十位数
    int iT = grade % 10;      // 成绩的个位数
    string score, level, lettergrade;
    // 根据成绩的十位数字确定 score
    if(iU == 9)
        score = "A";
    else if(iU == 8)
        score = "B";
    else if(iU == 7)
        score = "C";
    else
        score = "D";
    // 根据成绩的个位数字确定 level
    if(iT < 3)
        level = "-";
    else if(iT > 7)
        level = "+";
    else
        level = "";
    // 累加求得等级成绩
    lettergrade = score + level;
    cout << "等级成绩是：" << lettergrade << endl;

    return 0;
}

```

**练习 5.6：**改写上一题的程序，使用条件运算符（参见 4.7 节，第 134 页）代替 if-else 语句。

### 【出题思路】

条件运算符可以实现与 if-else 语句类似的功能，当有嵌套的条件表达式时，注意其满足右结合律。

### 【解答】

改写了 score 和 level 的求值形式，得到满足题意的程序如下所示：

```

#include <iostream>

using namespace std;

int main()
{
    int grade;
    cout << "请输入您的成绩：";
    cin >> grade;
    if(grade < 0 || grade > 100)
    {
        cout << "该成绩不合法" << endl;
        return -1;
    }

```

```

if(grade == 100)      // 处理满分的情况
{
    cout << "等级成绩是: " << "A++" << endl;
    return -1;
}
if(grade < 60)        // 处理不及格的情况
{
    cout << "等级成绩是: " << "F" << endl;
    return -1;
}
int iU = grade / 10;   // 成绩的十位数
int iT = grade % 10;   // 成绩的个位数
string score, level, lettergrade;
// 根据成绩的十位数字确定 score
score = (iU == 9) ? "A"
                  : (iU == 8) ? "B"
                  : (iU == 7) ? "C" : "D";
// 根据成绩的个位数字确定 level
level = (iT < 3) ? "-"
              : (iT > 7) ? "+" : "";
// 累加求得等级成绩
lettergrade = score + level;
cout << "等级成绩是: " << lettergrade << endl;

return 0;
}

```

**练习 5.7:** 改正下列代码段中的错误。

- (a) if (ival1 != ival2)
 ival1 = ival2
 else ival1 = ival2 = 0;
- (b) if (ival < minval)
 minval = ival;
 occurs = 1;
- (c) if (int ival = get\_value())
 cout << "ival = " << ival << endl;
 if (!ival)
 cout << "ival = 0\n";
- (d) if (ival = 0)
 ival = get\_value();

**【出题思路】**

理解 if 语句的语法规则。

**【解答】**

(a) if 语句的循环体应该是一条语句，需要以分号结束，程序修改为：

```

if (ival1 != ival2)
    ival1 = ival2;
```

```
else ival1 = ival2 = 0;
```

(b) if语句的循环体只能是一条语句，本题从逻辑上来说需要做两件事，一是修改minval的值，二是重置occurs的值，所以必须把这两条语句放在一个块里。程序修改为：

```
if (ival < minval)
{
    minval = ival;
    occurs = 1;
}
```

(c) ival是定义在if语句中的变量，其作用域仅限于第一个if语句，要想在第二个if语句中也使用它，就必须把它定义在两个if语句的外部。程序修改为：

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

(d) 程序的原意是判断ival的值是否是0，原题使用赋值运算符的结果是把0赋给了ival，然后检验ival的值，这样使得条件永远不会满足。程序修改为：

```
if (ival == 0)
    ival = get_value();
```

**练习 5.8：**什么是“悬垂else”？C++语言是如何处理else子句的？

#### 【出题思路】

理解悬垂else的定义，以及C++是如何处理悬垂else的。

#### 【解答】

悬垂else是指当程序中的if分支多于else分支时，如何为else寻找与之匹配的if分支的问题。

C++规定，else与离它最近的尚未匹配的if匹配，从而消除了二义性。

**练习 5.9：**编写一段程序，使用一系列if语句统计从cin读入的文本中有多少元音字母。

#### 【出题思路】

if语句和switch语句是分支语句的两种形式，一般来说可以互相转化。

#### 【解答】

满足题意的程序如下所示：

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

int main()
{
    unsigned int vowelCnt = 0;
    char ch;
    cout << "请输入一段文本：" << endl;
```

```

while(cin >> ch)
{
    if(ch == 'a')
        ++vowelCnt;
    if(ch == 'e')
        ++vowelCnt;
    if(ch == 'i')
        ++vowelCnt;
    if(ch == 'o')
        ++vowelCnt;
    if(ch == 'u')
        ++vowelCnt;
}
cout << "您输入的文本中共有 " << vowelCnt << "个元音字母" << endl;
return 0;
}

```

**练习 5.10：**我们之前实现的统计元音字母的程序存在一个问题：如果元音字母以大写形式出现，不会被统计在内。编写一段程序，既统计元音字母的小写形式，也统计大写形式，也就是说，新程序遇到'a'和'A'都应该递增 aCnt 的值，以此类推。

### 【出题思路】

要实现本题的要求，只需更新 switch 语句的 case 分支即可。

### 【解答】

本题的关键是在适当位置添加 break; 语句。其中，表示同一个字母的大小写的 case 标签之间不写 break;，因为它们要做的操作是一样的；而不同字母的 case 分支最后要写上 break;。满足题意的程序如下所示：

```

#include <iostream>
using namespace std;

int main()
{
    unsigned int aCnt = 0, eCnt = 0, iCnt = 0, oCnt = 0, uCnt = 0;
    char ch;
    cout << "请输入一段文本：" << endl;
    while(cin >> ch)
    {
        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;
        }
    }
    cout << "您输入的文本中共有 " << aCnt + eCnt + iCnt + oCnt + uCnt << "个元音字母" << endl;
    return 0;
}

```

```

        break;
    case 'o':
    case 'O':
        ++oCnt;
        break;
    case 'u':
    case 'U':
        ++uCnt;
        break;
    }
}

cout << "元音字母 a 的数量是: " << aCnt << endl;
cout << "元音字母 e 的数量是: " << eCnt << endl;
cout << "元音字母 i 的数量是: " << iCnt << endl;
cout << "元音字母 o 的数量是: " << oCnt << endl;
cout << "元音字母 u 的数量是: " << uCnt << endl;
return 0;
}

```

**练习 5.11：**修改统计元音字母的程序，使其也能统计空格、制表符和换行符的数量。

### 【出题思路】

继续扩充 case 标签即可。其中，读入字符的语句应该使用 `cin.get(ch)`，而不能使用`>>`，因为后者会忽略本题所要统计的特殊符号。

### 【解答】

满足题意的程序如下所示：

```

#include <iostream>
using namespace std;

int main()
{
    unsigned int aCnt = 0, eCnt = 0, iCnt = 0, oCnt = 0, uCnt = 0;
    unsigned int spaceCnt = 0, tabCnt = 0, newlineCnt = 0;
    char ch;
    cout << "请输入一段文本: " << endl;
    while(cin.get(ch))
    {
        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;
        }
    }
    cout << "元音字母 a 的数量是: " << aCnt << endl;
    cout << "元音字母 e 的数量是: " << eCnt << endl;
    cout << "元音字母 i 的数量是: " << iCnt << endl;
    cout << "元音字母 o 的数量是: " << oCnt << endl;
    cout << "元音字母 u 的数量是: " << uCnt << endl;
    cout << "空格字符的数量是: " << spaceCnt << endl;
    cout << "制表符字符的数量是: " << tabCnt << endl;
    cout << "换行符字符的数量是: " << newlineCnt << endl;
}

```

```

        ++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;
    }
}

cout << "元音字母 a 的数量是: " << aCnt << endl;
cout << "元音字母 e 的数量是: " << eCnt << endl;
cout << "元音字母 i 的数量是: " << iCnt << endl;
cout << "元音字母 o 的数量是: " << oCnt << endl;
cout << "元音字母 u 的数量是: " << uCnt << endl;
cout << "空格的数量是: " << spaceCnt << endl;
cout << "制表符的数量是: " << tabCnt << endl;
cout << "换行符的数量是: " << newlineCnt << endl;
return 0;
}

```

**练习 5.12：**修改统计元音字母的程序，使其能统计以下含有两个字符的字符串序列的数量：ff、fl 和 fi。

#### 【出题思路】

为了统计字符串序列的情况，必须记录前一个字符的内容。

#### 【解答】

我们的设定是一个字符只会被统计一次。如果用户输入的序列是xxxxxfflxxx，则统计结果是 ff: 1 次、fl: 0 次、fi: 0 次。如果用户输入的序列是xxxxxffffflxxx，则统计结果是 ff: 2 次、fl: 1 次、fi: 1 次。满足题意的程序如下所示：

```

#include <iostream>
using namespace std;

int main()
{
    unsigned int ffCnt = 0, flCnt = 0, fiCnt = 0;
    char ch, prech = '\0';

```

```

cout << "请输入一段文本: " << endl;
while(cin >> ch)
{
    bool bl = true;
    if(prech == 'f')
    {
        switch(ch)
        {
            case 'f':
                ++ffCnt;
                bl = false;
                break;
            case 'l':
                ++flCnt;
                break;
            case 'i':
                ++fiCnt;
                break;
        }
    }
    if(!bl)
        prech = '\0';
    else
        prech = ch;
}
cout << "ff 的数量是: " << ffCnt << endl;
cout << "fl 的数量是: " << flCnt << endl;
cout << "fi 的数量是: " << fiCnt << endl;
return 0;
}

```

**练习 5.13:** 下面显示的每个程序都含有一个常见的编程错误，指出错误在哪里，然后修改它们。

- (a) 

```

unsigned aCnt = 0, eCnt = 0, iouCnt = 0;
char ch = next_text();
switch (ch) {
    case 'a': aCnt++;
    case 'e': eCnt++;
    default: iouCnt++;
}
```
- (b) 

```

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

```

(c)  unsigned evenCnt = 0, oddCnt = 0;
    int digit = get_num() % 10;
    switch (digit) {
        case 1, 3, 5, 7, 9:
            oddcnt++;
            break;
        case 2, 4, 6, 8, 10:
            evencnt++;
            break;
    }
(d)  unsigned ival=512, jval=1024, kval=4096;
    unsigned bufsize;
    unsigned swt = get_bufCnt();
    switch(swt) {
        case ival:
            bufsize = ival * sizeof(int);
            break;
        case jval:
            bufsize = jval * sizeof(int);
            break;
        case kval:
            bufsize = kval * sizeof(int);
            break;
    }
}

```

### 【出题思路】

`switch` 语句有几个语法要点：必须在必要的地方使用 `break;` 语句，应该把变量定义在块作用域内，`case` 标签只能有一个值且不能是变量。

### 【解答】

(a) 的错误是在每个 `case` 分支中都缺少了 `break;` 语句，造成的后果是一旦执行了前面的 `case` 分支，必定还会继续执行接下来的其他 `case` 分支。举例说明，如果 `ch` 的内容是字符'a'，则 `aCnt`、`eCnt` 和 `iouCnt` 的值都会增加；如果 `ch` 的内容是字符'e'，则 `eCnt` 和 `iouCnt` 的值都会增加，这显然与程序的预期是不相符的。

修改后的程序如下所示：

```

unsigned aCnt = 0, eCnt = 0, iouCnt = 0;
char ch = next_text();
switch (ch) {
    case 'a':
        aCnt++;
        break;
    case 'e':
        eCnt++;
        break;
    default:
        iouCnt++;
        break;
}

```

(b) 的错误是在 `case` 分支中定义并初始化了变量 `ix`，同时在 `default` 分支中使用了该变量，此时如果控制流跳过 `case` 分支而直接到达 `default` 分支，则会试图使用未经初始化的变量，因而该程序无法通过编译。解决办法是，把 `ix` 的定义放

置在 switch 语句之前。

修改后的程序如下所示：

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

(c)的错误是在同一个 case 标签中放置了多个值，而 C++规定一个 case 标签只能对应一个值。修改后的程序如下所示：

```
unsigned evenCnt = 0, oddCnt = 0;
int digit = get_num() % 10;
switch (digit) {
    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)的错误是使用变量作为 case 标签的内容，C++规定，case 标签的内容只能是整型常量表达式。修改后的程序如下所示：

```
const unsigned ival=512, jval=1024, kval=4096;
unsigned bufsize;
unsigned swt = get_bufCnt();
switch(swt) {
    case ival:
        bufsize = ival * sizeof(int);
        break;
    case jval:
        bufsize = jval * sizeof(int);
        break;
    case kval:
        bufsize = kval * sizeof(int);
        break;
}
```

**练习 5.14:** 编写一段程序，从标准输入中读取若干 `string` 对象并查找连续重复出现的单词。所谓连续重复出现的意思是：一个单词后面紧跟着这个单词本身。要求记录连续重复出现的最大次数以及对应的单词。如果这样的单词存在，输出重复出现的最大次数；如果不存在，输出一条信息说明任何单词都没有连续出现过。例如，如果输入是

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

那么输出应该表明单词 `now` 连续出现了 3 次。

### 【出题思路】

使用 `while` 循环遍历用户输入的每一个字符串，根据当前字符串与上一个字符串是否相等决定如何更新各个变量的状态。如果相等，判断当前字符串连续出现的次数是否已经超过了系统记录的值，如果超过则及时更新。

### 【解答】

满足题意的程序如下所示：

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

int main()
{
    // 定义 3 个字符串变量，分别表示：
    // 当前操作的字符串、前一个字符串、当前出现次数最多的字符串
    string currString, preString = "", maxString;
    // 定义 2 个整型变量，分别表示：
    // 当前连续出现的字符串数量、当前出现次数最多的字符串数量
    int currCnt = 1, maxCnt = 0;
    while(cin >> currString)    // 检查每个字符串
    {
        // 如果当前字符串与前一个字符串一致，更新状态
        if(currString == preString)
        {
            ++currCnt;
            if(currCnt > maxCnt)
            {
                maxCnt = currCnt;
                maxString = currString;
            }
        }
        // 如果当前字符串与前一个字符串不一致，重置 currCnt
        else
        {
            currCnt = 1;
        }
        // 更新 preString 以便于下一次循环使用
        preString = currString;
    }
    if(maxCnt > 1)
        cout << "出现最多的字符串是：" << maxString
}
```

```

        << ", 次数是: " << maxCnt << endl;
    else
        cout << "每个字符串都只出现了一次" << endl;
    return 0;
}

```

**练习 5.15:** 说明下列循环的含义并改正其中的错误。

- (a)    for (int ix = 0; ix != sz; ++ix) { /\* ... \*/ }
       if (ix != sz)
          // ...
- (b)    int ix;
       for (ix != sz; ++ix) { /\* ... \*/ }
- (c)    for (int ix = 0; ix != sz; ++ix, ++ sz) { /\* ... \*/ }

#### 【出题思路】

理解 `for` 循环的语法规则，特别是 `for` 循环控制结构中三条语句的作用。理解循环控制变量与循环终止条件的关系，从而能够判断循环是否会无限执行下去。

#### 【解答】

(a) 的错误是在 `for` 语句中定义了变量 `ix`，然后试图在 `for` 语句之外继续使用 `ix`。因为 `ix` 定义在 `for` 语句的内部，所以其作用域仅限于 `for` 语句。在 `if` 语句中 `ix` 已经失效，因此程序无法编译通过。

修改后的程序如下：

```

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

```

(b) 的错误有两个，一是变量 `ix` 未经初始化就直接使用，二是 `for` 语句的控制结构缺少一句话，在语法上是错误的。

修改后的程序如下：

```

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

```

(c) 的错误是一旦进入循环，程序就会无休止地执行下去。也就是说，当初始情况下 `ix != sz` 时，由题意可知 `ix` 和 `sz` 一直同步增长，循环的终止条件永远不会满足，所以该循环是一个死循环。

修改后的程序如下：

```

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

```

**练习 5.16:** `while` 循环特别适用于那种条件保持不变、反复执行操作的情况，例如，当未达到文件末尾时不断读取下一个值。`for` 循环则更像是在按步骤迭代，它的索引值在某个范围内依次变化。根据每种循环的习惯用法各自编写一段程序，然后分别用另一种循环改写。如果只能使用一种循环，你倾向于使用哪种呢？为什么？

#### 【出题思路】

一般情况下，`while` 循环和 `for` 循环可以互相转换，不论哪种循环形式关键点

都有三个：循环控制变量的初始值、循环控制变量的变化规则、循环终止条件。编写循环语句时除了要满足应用需求，还必须时刻关注循环控制变量和循环终止条件，谨防写出死循环。

### 【解答】

从标准输入流读取数据的程序一般使用 `while` 循环，其形式是：

```
char ch;
while(cin >> ch)
{
    /* ... */
}
```

因为这项功能事实上不太要求我们严格跟踪循环变量的变化过程，所以改写成 `for` 循环后稍显冗余：

```
for( ; cin >> ch; )
{
    /* ... */
}
```

请注意：在上面的 `for` 循环控制结构中，第一条语句和第三条语句都是空语句，额外添加一个循环控制变量（比如 `int i`）是没有意义的。

整数累加求和的程序一般使用 `for` 循环，其形式是：

```
int iCount = 0;
for(int i = 0; i < 10; ++i)
{
    iCount += i;
}
```

使用 `while` 循环改写后的形式是：

```
int iCount = 0, i = 0;
while(i < 10)
{
    iCount += i;
    ++i;
}
```

在大多数情况下，两种循环形式可以互相转换。如果只能使用一种循环，作者倾向于使用 `for` 循环。`for` 循环的优点是结构严谨，便于控制程序的逻辑。

**练习 5.17：**假设有两个包含整数的 `vector` 对象，编写一段程序，检验其中一个 `vector` 对象是否是另一个的前缀。为了实现这一目标，对于两个不等长的 `vector` 对象，只需挑出长度较短的那个，把它的所有元素和另一个 `vector` 对象比较即可。例如，如果两个 `vector` 对象的元素分别是 0、1、1、2 和 0、1、1、2、3、5、8，则程序的返回结果应该为真。

### 【出题思路】

使用 `while` 循环实现两个 `vector` 对象的逐元素比较。

### 【解答】

满足题意的程序如下所示：

```

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

int main()
{
    vector<int> v1 = {0, 1, 1, 2};
    vector<int> v2 = {0, 1, 1, 2, 3, 5, 8};
    vector<int> v3 = {3, 5, 8};
    vector<int> v4 = {3, 5, 0, 9, 2, 7};

    auto it1 = v1.cbegin();           // 定义 v1 的迭代器
    auto it2 = v2.cbegin();           // 定义 v2 的迭代器
    auto it3 = v3.cbegin();           // 定义 v3 的迭代器
    auto it4 = v4.cbegin();           // 定义 v4 的迭代器
    // 设定循环条件：v1 和 v2 都尚未检查完
    while(it1 != v1.cend() && it2 != v2.cend())
    {
        // 如果当前位置的两个元素不相等，则肯定没有前缀关系，退出循环
        if(*it1 != *it2)
        {
            cout << "v1 和 v2 之间不存在前缀关系" << endl;
            break;
        }
        ++it1;                      // 迭代器移动到下一个元素
        ++it2;                      // 迭代器移动到下一个元素
    }
    if(it1 == v1.cend())           // 如果 v1 较短
    {
        cout << "v1 是 v2 的前缀" << endl;
    }
    if(it2 == v2.cend())           // 如果 v2 较短
    {
        cout << "v2 是 v1 的前缀" << endl;
    }
    return 0;
}

```

程序的输出结果是：v1 是 v2 的前缀。如果更新程序使其处理 v3 和 v4，则程序将显示：v3 和 v4 之间不存在前缀关系。

### 练习 5.18：说明下列循环的含义并改正其中的错误。

- (a) do
 

```

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

```

// . . .
      
```

```

    } while (int ival = get_response());
(c) do {
    int ival = get_response();
} while (ival);

```

### 【出题思路】

理解 do-while 循环的语法规则。do-while 与 while 的最主要区别是：do-while 语句至少会执行一次循环体；而 while 有可能一次都不执行循环体。

### 【解答】

(a)的含义是每次循环读入两个整数并输出它们的和。因为 do-while 语句的循环体必须是一条语句或者一个语句块，所以在本题中应该把循环体的内容用花括号括起来。修改后的程序是：

```

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

```

(b)的含义是当 get\_response 的返回值不为 0 时执行循环体。因为 do-while 语句不允许在循环条件内定义变量，所以该程序是错误的。修改后的程序是：

```

int ival;
do {
    ival = get_response();
} while (ival);

```

(c) 的含义是当 get\_response 的返回值不为 0 时执行循环体。因为出现在 do-while 语句条件部分的变量必须定义在循环体之外，所以该程序是错误的。修改后的程序是：

```

int ival;
do {
    ival = get_response();
} while (ival);

```

**练习 5.19：**编写一段程序，使用 do-while 循环重复地执行下述任务：首先提示用户输入两个 string 对象，然后挑出较短的那个并输出它。

### 【出题思路】

把题目要求的任务写在 do-while 循环体中即可，该程序至少会执行一次比较操作。把字符串定义成 string 对象，调用 size 函数即可得到 string 对象的长度。

### 【解答】

满足题意的程序如下所示：

```

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

int main()
{
    do{

```

```

        cout << "请输入两个字符串: " << endl;
        string str1, str2;
        cin >> str1 >> str2;
        if(str1.size() < str2.size())
            cout << "长度较小的字符串是: " << str1 << endl;
        else if(str1.size() > str2.size())
            cout << "长度较小的字符串是: " << str2 << endl;
        else
            cout << "两个字符串等长" << endl;
    }while (cin);

    return 0;
}

```

**练习 5.20：**编写一段程序，从标准输入中读取 `string` 对象的序列直到连续出现两个相同的单词或者所有单词都读完为止。使用 `while` 循环一次读取一个单词，当一个单词连续出现两次时使用 `break` 语句终止循环。输出连续重复出现的单词，或者输出一个消息说明没有任何单词是连续重复出现的。

### 【出题思路】

`break` 语句是一种跳转语句，负责终止离它最近的 `while`、`do-while`、`for` 和 `switch` 语句，并把程序的控制权交给这些语句之后的第一条语句。

### 【解答】

满足题意的程序如下所示：

```

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

int main()
{
    string currString, preString;
    bool bl = true;
    cout << "请输入一组字符串: " << endl;
    while(cin >> currString)
    {
        if(currString == preString)
        {
            bl = false;
            cout << "连续出现的字符串是: " << currString << endl;
            break;
        }
        preString = currString;
    }
    if(bl)
        cout << "没有连续出现的字符串" << endl;
    return 0;
}

```

本题用到了一个起标识作用的布尔值 `bl`，当 `bl` 为真时表示没有连续出现的字符串，一旦我们发现了存在连续出现的字符串，就立即把 `bl` 的值置为 `false`。

**练习 5.21:** 修改 5.5.1 节（第 171 页）练习题的程序，使其找到的重复单词必须以大写字母开头。

### 【出题思路】

`continue` 语句是一种跳转语句，负责终止最近的循环中的当前迭代并立即开始下一次迭代。与 `break` 语句的区别是，`continue` 虽然终止了当前迭代，但是并不终止循环；而 `break` 语句则直接跳出循环。

### 【解答】

根据题目的要求，我们首先检查读入的单词是否以大写字母开头，如果不是则终止当前迭代并开始下一次循环。满足题意的程序是：

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

int main()
{
    string currString, preString;
    bool bl = true;
    cout << "请输入一组字符串：" << endl;
    while(cin >> currString)
    {
        if(!isupper(currString[0]))
            continue;
        if(currString == preString)
        {
            bl = false;
            cout << "连续出现的字符串是：" << currString << endl;
            break;
        }
        preString = currString;
    }
    if(bl)
        cout << "没有连续出现的字符串" << endl;
    return 0;
}
```

**练习 5.22:** 本节的最后一个例子跳回到 `begin`，其实使用循环能更好地完成该任务。重写这段代码，注意不再使用 `goto` 语句。

### 【出题思路】

`goto` 语句的作用是从 `goto` 语句无条件跳转到同一函数的另一条语句。但是一般情况下，不建议读者使用 `goto` 语句，因为它使得程序既难理解又难修改。

### 【解答】

用 `do-while` 语句改写后的程序如下所示：

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

**练习 5.23：**编写一段程序，从标准输入读取两个整数，输出第一个数除以第二个数的结果。

#### 【出题思路】

对于整数除法来说，我们必须检查除数是否为0。在这个版本的程序中，暂时不使用try-catch机制，而是直接输出错误信息。

#### 【解答】

满足题意的程序如下所示：

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

int main()
{
    cout << "请依次输入被除数和除数：" << endl;
    int ival1, ival2;
    cin >> ival1 >> ival2;
    if(ival2 == 0)
    {
        cout << "除数不能为0" << endl;
        return -1;
    }
    cout << "两数相除的结果是：" << ival1 / ival2 << endl;
    return 0;
}
```

**练习 5.24：**修改你的程序，使得当第二个数是0时抛出异常。先不要设定catch子句，运行程序并真的为除数输入0，看看会发生什么？

#### 【出题思路】

与上一版本相比，通过抛出异常来提示用户的输入错误。

#### 【解答】

满足题意的程序如下所示：

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

int main()
{
    cout << "请依次输入被除数和除数：" << endl;
    int ival1, ival2;
    cin >> ival1 >> ival2;
    if(ival2 == 0)
    {
        throw runtime_error("除数不能为0");
    }
    cout << "两数相除的结果是：" << ival1 / ival2 << endl;
    return 0;
}
```

在本题中，我们设定当检测到除数为 0 时抛出一个 `runtime_error` 异常，因为没有 `catch` 语句，所以系统只报告异常而并不处理它。在我们的编译环境中，系统给出的报错信息是：

```
terminate called after throwing an instance of 'std::runtime_error'
what(): 除数不能为 0
This application has requested the Runtime to terminate it in an unusual
way. Please contact the application's support team for more information.
```

**练习 5.25：**修改上一题的程序，使用 `try` 语句块捕获异常。`catch` 子句应该为用户输出一条提示信息，询问其是否输入新数并重新执行 `try` 语句块的内容。

### 【出题思路】

使用 `try-catch` 结构实现完整的异常处理机制。

### 【解答】

满足题意的程序如下所示：

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

int main()
{
    cout << "请依次输入被除数和除数：" << endl;
    int ival1, ival2;
    while(cin >> ival1 >> ival2)
    {
        try
        {
            if(ival2 == 0)
            {
                throw runtime_error("除数不能为 0");
            }
            cout << "两数相除的结果是：" << ival1 / ival2 << endl;
        }catch(runtime_error err)
        {
            cout << err.what() << endl;
            cout << "需要继续吗 (y or n) ? ";
            char ch;
            cin >> ch;
            if(ch != 'y' && ch != 'Y')
                break;
        }
    }

    return 0;
}
```

# 第 6 章

## 函数

### 导读

本章介绍函数的用法，函数有三个重要组成部分：函数名、参数列表、返回值。本章的核心知识点与之对应：

- 传值参数、传引用参数、数组参数和可变参数
- 无返回值、有返回值、返回数组指针
- 同名函数重载及重载函数的匹配

读者需要理解函数声明和定义的方法，弄明白在函数调用的过程中到底发生了什么。形参和实参是关于函数参数的一对重要概念，实参与形参的匹配涉及函数的很多知识点。值传递的参数只在函数体内有效，而引用传递的参数可以作用于函数体之外。

函数重载是 C++ 的一个重要语法特征，允许程序员使用同名函数执行目标类似但细节有所区别的任务。函数匹配和发生在其中的参数类型转换是函数重载的关键。

### 练习 6.1：实参和形参的区别是什么？

#### 【出题思路】

函数名、返回值、参数三者组成了函数，其中参数分为实参和形参。

#### 【解答】

形参出现在函数定义的地方，形参列表可以包含 0 个、1 个或多个形参，多个形参之间以逗号分隔。形参规定了一个函数所接受数据的类型和数量。

实参出现在函数调用的地方，实参的数量与形参一样多。实参的主要作用是初始化形参，并且这种初始化过程是一一对应的，即第一个实参初始化第一个形参、第二个实参初始化第二个形参，以此类推。实参的类型必须与对应的形参类型匹配。

**练习 6.2:** 请指出下列函数哪个有错误，为什么？应该如何修改这些错误呢？

- (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;

#### 【出题思路】

本题旨在考查函数的一般语法规则。首先，函数的三要素（函数名、返回值、参数）必不可少；其次，函数的主体内容必须放在一对花括号内；最后，函数体内返回的结果必须与函数的返回值类型相同。

#### 【解答】

(a)是错误的，因为函数体返回的结果类型是 `string`，而函数的返回值类型是 `int`，二者不一致且不能自动转换。修改后的程序是：

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

(b)是错误的，因为函数缺少返回值类型。如果该函数确实不需要返回任何值，则程序应该修改为：

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

(c)是错误的，同一个函数如果含有多个形参，则这些形参的名字不能重复；另外，函数体左侧的花括号丢失了。修改后的程序应该是：

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

(d)是错误的，因为函数体必须放在一对花括号内。修改后的程序是：

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

**练习 6.3:** 编写你自己的 `fact` 函数，上机检查是否正确。

#### 【出题思路】

通过练习编写自己的阶乘函数，熟悉函数的语法规则和定义函数的方法。

#### 【解答】

原书中的 `fact` 函数使用 `while` 循环执行递减连乘运算，我们改写该程序，使用普通 `for` 循环得到满足题意的函数，如下所示：

```
int fact(int val)
{
    if(val < 0)
        return -1;
    int ret = 1;
    // 从 1 连乘到 val
    for(int i = 1; i != val + 1; ++i)
        ret *= i;
```

```

        return ret;
    }
}

```

**练习 6.4:** 编写一个与用户交互的函数，要求用户输入一个数字，计算生成该数字的阶乘。在 main 函数中调用该函数。

### 【出题思路】

理解函数的调用过程，尤其是注意区别形参和实参。

### 【解答】

满足题意的程序如下所示：

```

#include <iostream>
using namespace std;

int fact(int val)
{
    if(val < 0)
        return -1;
    int ret = 1;
    // 从1连乘到val
    for(int i = 1; i != val + 1; ++i)
        ret *= i;
    return ret;
}

int main()
{
    int num;
    cout << "请输入一个整数：";
    cin >> num;
    cout << num << "的阶乘是：" << fact(num) << endl;
    return 0;
}

```

**练习 6.5:** 编写一个函数，输出其实参的绝对值。

### 【出题思路】

练习定义函数并在 main 中调用该函数。

### 【解答】

根据参数类型的不同，我们可以分别求整数的绝对值和浮点数的绝对值。因为本题主要考查的知识点是编写和使用函数，因此参数的类型不是关键。从通用性的角度出发，我们不妨设定参数类型是双精度浮点数 double。满足题意的程序如下所示：

```

#include <iostream>
#include <cmath>
using namespace std;
// 第一个函数通过 if-else 语句计算绝对值
double myABS(double val)

```

```

{
    if(val < 0)
        return val * -1;
    else
        return val;
}
// 第二个函数调用 cmath 头文件的 abs 函数计算绝对值
double myABS2(double val)
{
    return abs(val);
}

int main()
{
    double num;
    cout << "请输入一个数: ";
    cin >> num;
    cout << num << "的绝对值是: " << myABS(num) << endl;
    cout << num << "的绝对值是: " << myABS2(num) << endl;
    return 0;
}

```

在这个程序中，我们定义了两个函数 myABS 和 myABS2，它们使用两种不同的方法求绝对值。第一个函数使用 if-else 分支语句判断实参是正数还是负数，从而计算实参的绝对值。第二个函数直接调用 cmath 头文件的 abs 函数实现同样的功能。

**练习 6.6：**说明形参、局部变量以及局部静态变量的区别。编写一个函数，同时用到这三种形式。

#### 【出题思路】

理解局部变量的含义，理解自动对象的创建和销毁机制，弄清楚我们为什么需要局部静态变量，应该如何使用局部静态变量。

#### 【解答】

形参和定义在函数体内部的变量统称为局部变量，它们对函数而言是局部的，仅在函数的作用域内可见。函数体内的局部变量又分为普通局部变量和静态局部变量，对于形参和普通局部变量来说，当函数的控制路径经过变量定义语句时创建该对象，当到达定义所在的块末尾时销毁它。我们把只存在于块执行期间的对象称为自动对象。这几个概念的区别是：

- 形参是一种自动对象，函数开始时为形参申请内存空间，我们用调用函数时提供的实参初始化形参对应的自动对象。
- 普通变量对应的自动对象也容易理解，我们在定义该变量的语句处创建自动对象，如果定义语句提供了初始值，则用该值初始化；否则，执行默认初始化。当该变量所在的块结束后，变量失效。
- 局部静态变量比较特殊，它的生命周期贯穿函数调用及之后的时间。局部静态变量对应的对象称为局部静态对象，它的生命周期从定义语句处开始，直到程序结束才终止。

下面的程序同时使用了形参、普通局部变量和静态局部变量：

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

// 该函数同时使用了形参、普通局部变量和静态局部变量
double myADD(double val1, double val2) // val1 和 val2 是形参
{
    double result = val1 + val2;          // result 是普通局部变量
    static unsigned iCnt = 0;             // iCnt 是静态局部变量
    ++iCnt;
    cout << "该函数已经累计执行了" << iCnt << "次" << endl;
    return result;
}

int main()
{
    double num1, num2;
    cout << "请输入两个数：";
    while(cin >> num1 >> num2)
    {
        cout << num1 << "与" << num2 << "的求和结果是：" 
            << myADD(num1, num2) << endl;
    }
    return 0;
}
```

**练习 6.7：**编写一个函数，当它第一次被调用时返回 0，以后每次被调用返回值加 1。

#### 【出题思路】

本题旨在考查局部静态变量的使用方法。

#### 【解答】

我们定义一个非常简单的函数，该函数除了统计执行次数外什么也不做。该函数的定义和调用形式如下所示：

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

// 该函数仅用于统计执行的次数
unsigned myCnt()                      // 完成该函数的任务不需要任何参数
{
    static unsigned iCnt = -1;           // iCnt 是静态局部变量
    ++iCnt;
    return iCnt;
}

int main()
{
    cout << "请输入任意字符后按回车键继续" << endl;
    char ch;
```

```

while(cin >> ch)
{
    cout << "函数 myCnt() 的执行次数是: " << myCnt() << endl;
}
return 0;
}

```

**练习 6.8:** 编写一个名为 Chapter6.h 的头文件，令其包含 6.1 节练习（第 184 页）中的函数声明。

#### 【出题思路】

函数声明与函数定义的形式非常类似，唯一的区别是函数声明无须函数体，用一个分号替代即可。函数应该在头文件中声明而在源文件中定义。

#### 【解答】

根据题目要求，Chapter6.h 头文件的内容如下：

```

#ifndef CHAPTER6_H_INCLUDED
#define CHAPTER6_H_INCLUDED

int fact(int );
double myABS(double );
double myABS2(double );

#endif // CHAPTER6_H_INCLUDED

```

**练习 6.9:** 编写你自己的 fact.cc 和 factMain.cc，这两个文件都应该导入上一小节的练习中编写的 Chapter6.h 头文件。通过这些文件，理解你的编译器是如何支持分离式编译的。

#### 【出题思路】

理解分离式编译的机制。

#### 【解答】

fact.cc 文件的内容是：

```

#include "Chapter6.h"
using namespace std;

int fact(int val)
{
    if(val < 0)
        return -1;
    int ret = 1;
    for(int i = 1; i != val + 1; ++i)
        ret *= i;
    return ret;
}

```

factMain.cc 的内容是：

```

#include <iostream>
#include "Chapter6.h"

```

```

using namespace std;

int main()
{
    int num;
    cout << "请输入一个整数: ";
    cin >> num;
    cout << num << "的阶乘是: " << fact(num) << endl;
    return 0;
}

```

**练习 6.10:** 编写一个函数，使用指针形参交换两个整数的值。在代码中调用该函数并输出交换后的结果，以此验证函数的正确性。

#### 【出题思路】

使用指针作为参数时，在函数内部交换指针的值只改变局部变量，不会影响实参原本的值，无法满足题目要求。我们应该在函数内部通过解引用操作改变指针所指的内容。

#### 【解答】

满足题意的程序如下所示：

```

#include <iostream>
using namespace std;
// 在函数体内部通过解引用操作改变指针所指的内容
void mySWAP(int *p, int *q)
{
    int tmp = *p;    // tmp 是一个整数
    *p = *q;
    *q = tmp;
}

int main()
{
    int a = 5, b = 10;
    int *r = &a, *s = &b;
    cout << "交换前: a = " << a << ", b = " << b << endl;
    mySWAP(r, s);
    cout << "交换后: a = " << a << ", b = " << b << endl;
    return 0;
}

```

请特别注意，下面这个程序无法满足题目的要求：

```

#include <iostream>
using namespace std;
// 在函数体内部交换了两个形参指针本身的价值，未能影响实参
void mySWAP(int *p, int *q)
{
    int *tmp = p;    // tmp 是一个指针
    p = q;
    q = tmp;
}

```

```

int main()
{
    int a = 5, b = 10;
    int *r = &a, *s = &b;
    cout << "交换前: a = " << a << ", b = " << b << endl;
    mySWAP(r, s);
    cout << "交换后: a = " << a << ", b = " << b << endl;
    return 0;
}

```

**练习 6.11:** 编写并验证你自己的 `reset` 函数，使其作用于引用类型的参数。

#### 【出题思路】

参考原书中的示例编写 `reset` 函数，它的参数是引用类型，从而使得我们可以直接操作引用所引的对象，再把该函数置于 `main` 函数体中检验效果。

#### 【解答】

满足题意的程序如下所示：

```

#include <iostream>
using namespace std;

void reset(int &i)
{
    i = 0;
}

int main()
{
    int num = 10;
    cout << "重置前: num = " << num << endl;
    reset(num);
    cout << "重置后: num = " << num << endl;
    return 0;
}

```

**练习 6.12:** 改写 6.2.1 节中练习 6.10（第 188 页）的程序，使用引用而非指针交换两个整数的值。你觉得哪种方法更易于使用呢？为什么？

#### 【出题思路】

使用引用可以直接操作所引用的对象，从而实现对象内容的交换。

#### 【解答】

满足题意的程序如下所示：

```

#include <iostream>
using namespace std;

void mySWAP(int &i, int &j)
{
    int tmp = i;

```

```

    i = j;
    j = tmp;
}

int main()
{
    int a = 5, b = 10;
    cout << "交换前: a = " << a << ", b = " << b << endl;
    mySWAP(a, b);
    cout << "交换后: a = " << a << ", b = " << b << endl;
    return 0;
}

```

与使用指针相比，使用引用交换变量的内容从形式上看更简单一些，并且无须额外声明指针变量，也避免了拷贝指针的值。

**练习 6.13：**假设 T 是某种类型的名字，说明以下两个函数声明的区别：一个是 void f(T)，另一个是 void f(T&)。

#### 【出题思路】

区别传值参数与传引用参数。

#### 【解答】

void f(T) 的形参采用的是传值方式，也就是说，实参的值被拷贝给形参，形参和实参是两个相互独立的变量，在函数 f 内部对形参所做的任何改动都不会影响实参的值。

void f(T&) 的形参采用的是传引用方式，此时形参是对应的实参的别名，形参绑定到初始化它的对象。如果我们改变了形参的值，也就是改变了对应实参的值。

下面这个程序可以说明两个函数声明的区别：

```

#include <iostream>
using namespace std;

void a(int);      // 传值参数
void b(int&);   // 传引用参数

int main()
{
    int s = 0, t = 10;
    a(s);
    cout << s << endl;
    b(t);
    cout << t << endl;
    return 0;
}

void a(int i)
{
    ++i;
    cout << i << endl;
}

```

```
void b(int& j)
{
    ++j;
    cout << j << endl;
}
```

程序的运行结果是：

```
1
0
11
11
```

**练习 6.14：**举一个形参应该是引用类型的例子，再举一个形参不能是引用类型的例子。

#### 【出题思路】

与值传递相比，引用传递的优势主要体现在三个方面：一是可以直接操作引用形参所引的对象；二是使用引用形参可以避免拷贝大的类类型对象或容器类型对象；三是使用引用形参可以帮助我们从函数中返回多个值。

#### 【解答】

基于对引用传递优势的分析，我们可以举出几个适合使用引用类型形参的例子：第一，当函数的目的是交换两个参数的内容时应该使用引用类型的形参；第二，当参数是 `string` 对象时，为了避免拷贝很长的字符串，应该使用引用类型。

在其他情况下可以使用值传递的方式，而无须使用引用传递，例如求整数的绝对值或者阶乘的程序。

**练习 6.15：**说明 `find_char` 函数中的三个形参为什么是现在的类型，特别说明为什么 `s` 是常量引用而 `occurs` 是普通引用？为什么 `s` 和 `occurs` 是引用类型而 `c` 不是？如果令 `s` 是普通引用会发生什么情况？如果令 `occurs` 是常量引用会发生什么情况？

#### 【出题思路】

理解并对比普通变量、引用类型和常量引用作为函数参数的区别。

#### 【解答】

`find_char` 函数的三个参数的类型设定与该函数的处理逻辑密切相关，原因分别如下：

- 对于待查找的字符串 `s` 来说，为了避免拷贝长字符串，使用引用类型；同时我们只执行查找操作，无须改变字符串的内容，所以将其声明为常量引用。
- 对于待查找的字符 `c` 来说，它的类型是 `char`，只占 1 字节，拷贝的代价很低，而且我们无须操作实参在内存中实际存储的内容，只把它的值拷贝给形参即可，所以不需要使用引用类型。
- 对于字符出现的次数 `occurs` 来说，因为需要把函数内对实参值的更改反映在函数外部，所以必须将其定义成引用类型；但是不能把它定义成常量引用，

否则就不能改变所引的内容了。

**练习 6.16：**下面的这个函数虽然合法，但是不算特别有用。指出它的局限性并设法改善。

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

#### 【出题思路】

本题旨在考查普通引用和常量引用作为参数类型的区别。一般情况下，与普通引用相比，我们更应该选择使用常量引用作为参数类型。

#### 【解答】

本题的程序把参数类型设为非常量引用，这样做有几个缺陷：一是容易给使用者一种误导，即程序允许修改变量 s 的内容；二是限制了该函数所能接受的实参类型，我们无法把 const 对象、字面值常量或者需要进行类型转换的对象传递给普通的引用形参。

根据上述分析，该函数应该修改为：

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

**练习 6.17：**编写一个函数，判断 string 对象中是否含有大写字母。编写另一个函数，把 string 对象全都改成小写形式。在这两个函数中你使用的形参类型相同吗？为什么？

#### 【出题思路】

当函数对参数所做的操作不同时，应该选择适当的参数类型。如果需要修改参数的内容，则将其设置为普通引用类型；否则，如果不需要对参数内容做任何更改，最好设为常量引用类型。

#### 【解答】

第一个函数的任务是判断 string 对象中是否含有大写字母，无须修改参数的内容，因此将其设为常量引用类型。第二个函数需要修改参数的内容，所以应该将其设定为非常量引用类型。满足题意的程序如下所示：

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

bool HasUpper(const string& str) // 判断字符串是否含有大写字母
{
    for(auto c : str)
        if(isupper(c))
            return true;
    return false;
}

void ChangeToLower(string& str) // 把字符串中的所有大写字母转成小写
{
```

```

for(auto &c : str)
    c = tolower(c);
}

int main()
{
    cout << "请输入一个字符串: " << endl;
    string str;
    cin >> str;
    if(HasUpper(str))
    {
        ChangeToLower(str);
        cout << "转换后的字符串是: " << str << endl;
    }
    else
        cout << "该字符串不含大写字母, 无须转换" << endl;
    return 0;
}

```

**练习 6.18:** 为下面的函数编写函数声明, 从给定的名字中推测函数具备的功能。

- (a) 名为 compare 的函数, 返回布尔值, 两个参数都是 matrix 类的引用。
- (b) 名为 change\_val 的函数, 返回 vector<int>的迭代器, 有两个参数: 一个是 int, 另一个是 vector<int>的迭代器。

#### 【出题思路】

根据题意确定函数名、函数返回值以及函数的参数列表。

#### 【解答】

(a)的函数声明是:

```
bool compare( const matrix&, const matrix& )
```

(b)的函数声明是:

```
vector<int>::iterator change_val( int, vector<int>::iterator )
```

**练习 6.19:** 假定有如下声明, 判断哪个调用合法、哪个调用不合法。对于不合法的函数调用, 说明原因。

```

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("abcda", 'a');
(c) calc(66);              (d) sum(vec.begin(), vec.end(), 3.8);

```

#### 【出题思路】

调用函数时, 实参类型必须与形参类型匹配。

#### 【解答】

(a)是非法的, 函数的声明只包含一个参数, 而函数的调用提供了两个参数, 因此无法编译通过。

(b) 是合法的，字面值常量可以作为常量引用形参的值，字符'a'作为 char 类型形参的值也是可以的。

(c) 是合法的，66 虽然是 int 类型，但是在调用函数时自动转换为 double 类型。

(d) 是合法的，vec.begin() 和 vec.end() 的类型都是形参所需的 vector<int>::iterator，第三个实参3.8 可以自动转换为形参所需的 int 类型。

**练习 6.20：**引用形参什么时候应该是常量引用？如果形参应该是常量引用，而我们将其设为了普通引用，会发生什么情况？

#### 【出题思路】

本题考查常量引用形参和非常量引用形参的原理及适用范围。

#### 【解答】

当函数对参数所做的操作不同时，应该选择适当的参数类型。如果需要修改参数的内容，则将其设置为普通引用类型；否则，如果不需要对参数内容做任何更改，最好设为常量引用类型。

就像前面几个练习题展示的那样，如果把一个本来应该是常量引用的形参设成了普通引用类型，有可能遇到几个问题：一是容易给使用者一种误导，即程序允许修改实参的内容；二是限制了该函数所能接受的实参类型，无法把 const 对象、字面值常量或者需要类型转换的对象传递给普通的引用形参。

**练习 6.21：**编写一个函数，令其接受两个参数：一个是 int 型的数，另一个是 int 指针。函数比较 int 的值和指针所指的值，返回较大的那个。在该函数中指针的类型应该是什么？

#### 【出题思路】

第一个参数的实际值毫无疑问是 int 类型，第二个参数是 int 指针，实际上有可能表示的是一个 int 数组，该指针指向了数组的首元素。

#### 【解答】

通过分析题意我们知道，函数实际上比较的是第一个实参的值和第二个实参所指数组首元素的值。因为两个参数的内容都不会被修改，所以指针的类型应该是 const int\*。满足题意的程序如下所示：

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

int myCompare(const int val, const int *p)
{
    return (val > *p) ? val : *p;
```

```

}

int main()
{
    srand((unsigned) time (NULL));
    int a[10];
    for(auto &i : a)
        i = rand() % 100;
    cout << "请输入一个数: ";
    int j;
    cin >> j;
    cout << "您输入的数与数组首元素中较大的是: " << myCompare(j, a) << endl;
    cout << "数组的全部元素是: " << endl;
    for(auto i : a)
        cout << i << " ";
    cout << endl;
    return 0;
}

```

**练习 6.22：**编写一个函数，令其交换两个 int 指针。

#### 【出题思路】

对于题目的要求有两种理解，一种是交换指针本身的价值，即指针所指向的内存地址；另一种是交换指针所指向的内容。

#### 【解答】

为了全面性考虑，我们在程序中实现了三个不同版本的函数。

第一个函数以值传递的方式使用指针，所有改变都局限于函数内部，当函数执行完毕后既不会改变指针本身的价值，也不会改变指针所指向的内容。

第二个函数同样以值传递的方式使用指针，但是在函数内部通过解引用的方式直接访问内存并修改了指针所指向的内容。

第三个函数的参数形式是 `int *&`，其含义是，该参数是一个引用，引用的对象是内存中的一个 `int` 指针，使用这种方式可以把指针当成对象，交换指针本身的价值。需要注意的是，最后一个函数既然交换了指针，当然解引用该指针所得的结果也会相应发生改变。

```

#include <iostream>
using namespace std;
// 该函数既不交换指针，也不交换指针所指向的内容
void SwapPointer1(int *p, int *q)
{
    int *temp = p;
    p = q;
    q = temp;
}
// 该函数交换指针所指向的内容
void SwapPointer2(int *p, int *q)
{
    int temp = *p;
    *p = *q;
    *q = temp;
}

```

```

        *q = temp;
    }
    // 该函数交换指针本身的价值，即交换指针所指的内存地址
    void SwapPointer3(int *&p, int *&q)
    {
        int *temp = p;
        p = q;
        q = temp;
    }

    int main()
    {
        int a = 5, b = 10;
        int *p = &a, *q = &b;
        cout << "交换前：" << endl;
        cout << "p 的值是：" << p << ", q 的值是：" << q << endl;
        cout << "p 所指的值是：" << *p << ", q 所指的值是：" << *q << endl;
        SwapPointer1(p, q);
        cout << "交换后：" << endl;
        cout << "p 的值是：" << p << ", q 的值是：" << q << endl;
        cout << "p 所指的值是：" << *p << ", q 所指的值是：" << *q << endl;

        a = 5, b = 10;
        p = &a, q = &b;
        cout << "交换前：" << endl;
        cout << "p 的值是：" << p << ", q 的值是：" << q << endl;
        cout << "p 所指的值是：" << *p << ", q 所指的值是：" << *q << endl;
        SwapPointer2(p, q);
        cout << "交换后：" << endl;
        cout << "p 的值是：" << p << ", q 的值是：" << q << endl;
        cout << "p 所指的值是：" << *p << ", q 所指的值是：" << *q << endl;

        a = 5, b = 10;
        p = &a, q = &b;
        cout << "交换前：" << endl;
        cout << "p 的值是：" << p << ", q 的值是：" << q << endl;
        cout << "p 所指的值是：" << *p << ", q 所指的值是：" << *q << endl;
        SwapPointer3(p, q);
        cout << "交换后：" << endl;
        cout << "p 的值是：" << p << ", q 的值是：" << q << endl;
        cout << "p 所指的值是：" << *p << ", q 所指的值是：" << *q << endl;
        return 0;
    }
}

```

**练习 6.23：**参考本节介绍的几个 print 函数，根据理解编写你自己的版本。依次调用每个函数使其输入下面定义的 i 和 j：

```
int i = 0, j[2] = {0, 1};
```

【出题思路】

根据参数的不同，为 print 函数设计几个版本。版本的区别主要体现在对指针参数的管理方式不同。

### 【解答】

实现了三个版本的 print 函数，第一个版本不控制指针的边界，第二个版本由调用者指定数组的维度，第三个版本使用 C++11 新规定的 begin 和 end 函数限定数组边界。满足题意的程序如下所示：

```
#include <iostream>
using namespace std;
// 参数是常量整型指针
void print1(const int *p)
{
    cout << *p << endl;
}
// 参数有两个，分别是常量整型指针和数组的容量
void print2(const int *p, const int sz)
{
    int i = 0;
    while(i != sz)
    {
        cout << *p++ << endl;
        ++i;
    }
}
// 参数有两个，分别是数组的首尾边界
void print3(const int *b, const int *e)
{
    for(auto q = b; q != e; ++q)
        cout << *q << endl;
}

int main()
{
    int i = 0, j[2] = {0, 1};
    print1(&i);
    print1(j);
    print2(&i, 1);
    // 计算得到数组 j 的容量
    print2(j, sizeof(j) / sizeof(*j));
    auto b = begin(j);
    auto e = end(j);
    print3(b, e);
    return 0;
}
```

**练习 6.24：**描述下面这个函数的行为。如果代码中存在问题，请指出并改正。

```
void print(const int ia[10])
{
    for (size_t i = 0; i != 10; ++i)
        cout << ia[i] << endl;
}
```

**【出题思路】**

当我们想把数组作为函数的形参时，有三种可供选择的方式：一是声明为指针，二是声明为不限维度的数组，三是声明为维度确定的数组。实际上，因为数组传入函数时实参自动转换成指向数组首元素的指针，所以这三种方式是等价的。

**【解答】**

由之前的分析可知，print 函数的参数实际上等同于一个常量整型指针 const int\*，形参 ia 的维度 10 只是我们期望的数组维度，实际上不一定。即使实参数组的真实维度不是 10，也可以正常调用 print 函数。

上述 print 函数的定义存在一个潜在风险，即虽然我们期望传入的数组维度是 10，但实际上任意维度的数组都可以传入。如果传入的数组维度较大，print 函数输出数组的前 10 个元素，不至于引发错误；相反如果传入的数组维度不足 10，则 print 函数将强行输出一些未定义的值。

修改后的程序是：

```
void print(const int ia[], const int sz)
{
    for (size_t i = 0; i != sz; ++i)
        cout << ia[i] << endl;
}
```

**练习 6.25：**编写一个 main 函数，令其接受两个实参。把实参的内容连接成一个 string 对象并输出出来。

**【出题思路】**

传递给 main 函数的参数有两个，第一个参数 argc 指明数组中字符串的数量，第二个参数 argv 是存有字符串的数组。

**【解答】**

满足题意的程序如下所示：

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

int main(int argc, char **argv)
{
    string str;
    for(int i = 0; i != argc; ++i)
        str += argv[i];
    cout << str << endl;
    return 0;
}
```

**练习 6.26：**编写一个程序，使其接受本节所示的选项；输出传递给 main 函数的实参的内容。

**【出题思路】**

格式化输出 argv 所包含的每一个字符串。

### 【解答】

满足题意的程序如下所示：

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

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

**练习 6.27:** 编写一个函数，它的参数是 `initializer_list<int>` 类型的对象，函数的功能是计算列表中所有元素的和。

### 【出题思路】

掌握 `initializer_list` 对象的声明和初始化方法，利用 `initializer_list` 对象设计形参可变的函数。

### 【解答】

满足题意的程序如下所示，注意 `iCount` 的参数是 `initializer_list` 对象，在调用该函数时，我们使用了列表初始化的方式生成实参。

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

int iCount(initializer_list<int> il)
{
    int count = 0;
    // 遍历 il 的每一个元素
    for(auto val : il)
        count += val;
    return count;
}

int main()
{
    // 使用列表初始化的方式构建 initializer_list<int> 对象
    // 然后把它作为实参传递给函数 iCount
    cout << "1,6,9 的和是：" << iCount({1, 6, 9}) << endl;
    cout << "4,5,9,18 的和是：" << iCount({4, 5, 9, 18}) << endl;
    cout << "10,10,10,10,10,10,10,10 的和是：" << iCount({10, 10, 10, 10, 10, 10, 10, 10}) << endl;
    return 0;
}
```

**练习 6.28:** 在 `error_msg` 函数的第二个版本中包含 `ErrCode` 类型的参数，其中循环内的 `elem` 是什么类型？

**【出题思路】**

理解 `initializer_list` 的含义和用法。

**【解答】**

`initializer_list<string>` 的所有元素类型都是 `string`，因此 `const auto &elem : il` 推断得到的 `elem` 的类型是 `const string&`。使用引用是为了避免拷贝长字符串，把它定义为常量的原因是我们只需读取字符串的内容，不需要修改它。

**练习 6.29:** 在范围 `for` 循环中使用 `initializer_list` 对象时，应该将循环控制变量声明成引用类型吗？为什么？

**【出题思路】**

考虑引用类型与普通类型的区别。

**【解答】**

引用类型的优势主要是可以直接操作所引用的对象以及避免拷贝较为复杂的类类型对象和容器对象。因为 `initializer_list` 对象的元素永远是常量值，所以我们不可能通过设定引用类型来更改循环控制变量的内容。只有当 `initializer_list` 对象的元素类型是类类型或容器类型（比如 `string`）时，才有必要把范围 `for` 循环的循环控制变量设为引用类型。

**练习 6.30:** 编译第 200 页的 `str_subrange` 函数，看看你的编译器是如何处理函数中的错误的。

**【出题思路】**

函数对于返回结果的要求是：每一条 `return` 语句的结果类型必须与函数的返回值类型相同，并且在函数执行逻辑中每一个可能的结束点都应该有一条 `return` 语句。

**【解答】**

该函数在作者所用的编译环境中无法编译通过，编译器发现了一个严重错误，即 `for` 循环中的 `return` 语句是非法的。函数的返回值类型是布尔值，而该条 `return` 语句没有返回任何值。

事实上程序还存在另一个严重错误，按照程序的逻辑，`for` 循环有可能不会中途退出而是一直执行完毕，此时显然缺少一条 `return` 语句处理这种情况。遗憾的是，编译器无法发现这一错误。

**练习 6.31:** 什么情况下返回的引用有效？什么情况下返回常量的引用有效？

#### 【出题思路】

函数返回其结果的过程与它接受参数的过程类似。如果返回的是值，则创建一个未命名的临时对象，并把要返回的值拷贝给这个临时对象；如果返回的是引用，则该引用是它所引对象的别名，不会真正拷贝对象。

#### 【解答】

如果引用所引的是函数开始之前就已经存在的对象，则返回该引用是有效的；如果引用所引的是函数的局部变量，则随着函数结束局部变量也失效了，此时返回的引用无效。

当不希望返回的对象被修改时，返回对常量的引用。

**练习 6.32:** 下面的函数合法吗？如果合法，说明其功能；如果不合法，修改其中的错误并解释原因。

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

#### 【出题思路】

考查当函数的参数和返回值是复合类型时，如何向函数传入数据及如何接受返回结果。读者尤其需要理解函数是怎样返回引用类型的。

#### 【解答】

该函数是合法的。`get` 函数接受一个整型指针，该指针实际指向一个整型数组的首元素，另外还接受一个整数表示数组中某个元素的索引值。它的返回值类型是整型引用，引用的对象是 `arry` 数组的某个元素。当 `get` 函数执行完毕后，调用者得到实参数组 `arry` 中索引为 `index` 的元素的引用。

在 `main` 函数中，首先创建一个包含 10 个整数的数组，名字是 `ia`。请注意，由于 `ia` 定义在 `main` 函数的内部，所以 `ia` 不会执行默认初始化操作，如果此时我们直接输出 `ia` 每个元素的值，则这些值都是未定义的。接下来进入循环，每次循环使用 `get` 函数得到数组 `ia` 中第 `i` 个元素的引用，为该引用赋值 `i`，也就是说，为第 `i` 个元素赋值 `i`。循环结束时，`ia` 的元素依次被赋值为 0~9。

读者可以在编程环境中验证上述程序的功能。

**练习 6.33:** 编写一个递归函数，输出 `vector` 对象的内容。

#### 【出题思路】

函数的递归分为直接递归和间接递归。编写递归函数的关键是确定递归规律和递归终止条件。

**【解答】**

满足题意的程序如下所示：

```
#include <iostream>
#include <vector>
using namespace std;
// 递归函数输出 vector<int> 的内容
void print(vector<int> vInt, unsigned index)
{
    unsigned sz = vInt.size();
    if(!vInt.empty() && index < sz)
    {
        cout << vInt[index] << endl;
        print(vInt, index + 1);
    }
}

int main()
{
    vector<int> v = {1, 3, 5, 7, 9, 11, 13, 15};
    print(v, 0);
    return 0;
}
```

**练习 6.34：**如果 factorial 函数的停止条件如下所示，将发生什么情况？

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

**【出题思路】**

理解递归函数的执行逻辑。

**【解答】**

因为原文中递归函数的参数类型是 int，所以理论上用户传入 factorial 函数的参数可以是负数。按照原程序的逻辑，参数为负数时函数的返回值是 1。

如果修改递归函数的停止条件，则当参数的值为负时，会依次递归下去，执行连续乘法操作直至溢出。因此，不能把 if 语句的条件改成上述形式。

**练习 6.35：**在调用 factorial 函数时，为什么我们传入的值是 val-1 而非 val--？

**【出题思路】**

回顾后置递减运算符参与表达式运算时的求值规律。

**【解答】**

如果把 val-1 改成 val--，则出现一种我们不期望看到的情况，即变量的递减操作与读取变量值的操作共存于同一条表达式中，这时有可能产生未定义的值。

**练习 6.36:** 编写一个函数的声明，使其返回数组的引用并且该数组包含 10 个 string 对象。不要使用尾置返回类型、decltype 或者类型别名。

#### 【出题思路】

因为数组不能被拷贝，所以函数不能直接返回数组，但是可以返回数组的指针或引用。

#### 【解答】

要想使函数返回数组的引用并且该数组包含 10 个 string 对象，可以按照如下所示的形式声明函数：

```
string (&func( ))[10];
```

上述声明的含义是：func() 表示调用 func 函数无须任何实参，(&func( )) 表示函数的返回结果是一个引用，(&func( ))[10] 表示引用的对象是一个维度为 10 的数组，string (&func( ))[10] 表示数组的元素是 string 对象。

**练习 6.37:** 为上一题的函数再写三个声明，一个使用类型别名，另一个使用尾置返回类型，最后一个使用 decltype 关键字。你觉得哪种形式最好？为什么？

#### 【出题思路】

直接编写返回数组引用的函数比较烦琐且不易理解，使用类型别名、尾置返回类型和 decltype 关键字都可以简化这一过程。

#### 【解答】

使用类型别名：

```
typedef string arr[10];
arr& func( );
```

使用尾置返回类型：

```
auto func( ) -> string(&) [10];
```

使用 decltype 关键字：

```
string str[10];
decltype(str) &func( );
```

**练习 6.38:** 修改 arrPtr 函数，使其返回数组的引用。

#### 【出题思路】

数组也是一个对象，所以可以定义数组的引用。要想为数组的引用赋值，只需要把数组名赋给该引用即可。

#### 【解答】

满足题意的 arrPtr 函数是：

```
int odd[] = {1, 3, 5, 7, 9};
int even[] = {0, 2, 4, 6, 8};
// 返回一个引用，该引用所引的对象是一个含有 5 个整数的数组。
decltype(odd) &arrPtr(int i)
{
    return (i % 2) ? odd : even; // 返回数组的引用。
}
```

**练习 6.39:** 说明在下面的每组声明中第二条声明语句是何含义。如果有非法的声明，请指出来。

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

#### 【出题思路】

本题旨在考查重载函数的含义以及如何判断两个函数是否是重载关系。

#### 【解答】

(a) 的第二个声明是非法的。它的意图是声明另外一个函数，该函数只接受整型常量作为实参，但是因为顶层 `const` 不影响传入函数的对象，所以一个拥有顶层 `const` 的形参无法与另一个没有顶层 `const` 的形参区分开来。

(b) 的第二个声明是非法的。它的意图是通过函数的返回值区分两个同名的函数，但是这不可行，因为 C++ 规定重载函数必须在形参数量或形参类型上有所区别。如果两个同名函数的形参数量和类型都一样，那么即使返回类型不同也不行。

(c) 的两个函数是重载关系，它们的形参类型有区别。

**练习 6.40:** 下面的哪个声明是错误的？为什么？

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

#### 【出题思路】

如果函数的某个形参在多次调用中都被赋予了同一个值，则我们可以把这个反复出现的值定义为默认实参。C++ 对于默认实参在参数列表中可能出现的位置有明确规定。

#### 【解答】

在上面的两个声明中，(a) 是正确的而(b) 是错误的。它们都用到了默认实参，但是 C++ 规定一旦某个形参被赋予了默认实参，则它后面的所有形参都必须有默认实参。

这一规定是为了防范可能出现的二义性，显然(b) 违反了这一规定。

**练习 6.41:** 下面的哪个调用是非法的？为什么？哪个调用虽然合法但显然与程序员的初衷不符？为什么？

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

#### 【出题思路】

要想使用默认实参，只需要在调用函数时省略该实参就可以了。实参按照其位置解析，默认实参负责填补函数调用缺少的尾部实参。

#### 【解答】

(a) 是非法的，该函数有两个默认实参，但是总计有三个形参，其中第一个形参并未设定默认实参，所以要想调用该函数，至少需要提供一个实参。

(b) 是合法的，本次调用提供了两个实参，第一个实参对应第一个形参 ht，第二个实参对应第二个形参 wd，其中 wd 的默认实参没有用到，第三个形参 bckgrnd 使用它的默认实参。

(c) 在语法上是合法的，但是与程序的原意不符。从语法上来说，第一个实参对应第一个形参 ht，第二个实参的类型虽然是 char，但是它可以自动转换为第二个形参 wd 所需的 int 类型，所以编译时可以通过，但这显然违背了程序的原意，正常情况下，字符'\*'应该被用来构成背景。

**练习 6.42：**给 make\_plural 函数（参见 6.3.2 节，第 201 页）的第二个形参赋予默认实参's'，利用新版本的函数输出单词 success 和 failure 的单数和复数形式。

#### 【出题思路】

对于英语单词来说，大多数名词的复数是在单词末尾加's'得到的，也有一部分名词在单数转变为复数时需要在末尾加'es'。我们可以把's'作为默认实参，大多数情况下不必考虑这个参数，只有在遇到末尾是'es'的单词时才专门处理。

#### 【解答】

满足题意的程序如下所示：

```
#include <iostream>
#include <string>
using namespace std;
// 最后一个形参赋予了默认实参
string make_plural(size_t ctr, const string &word, const string &ending
= "s")
{
    return (ctr > 1) ? word + ending : word;
}

int main()
{
    cout << "success 的单数形式是：" << make_plural(1, "success", "es")
        << endl;
    cout << "success 的复数形式是：" << make_plural(2, "success", "es")
        << endl;
    // 一般情况下调用该函数只需要两个实参
    cout << "failure 的单数形式是：" << make_plural(1, "failure") << endl;
    cout << "failure 的复数形式是：" << make_plural(2, "failure") << endl;
    return 0;
}
```

**练习 6.43:** 你会把下面的哪个声明和定义放在头文件中？哪个放在源文件中？为什么？

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

#### 【出题思路】

函数的声明应该放在头文件中，同时内联函数的定义也应该放在头文件中。

#### 【解答】

(a) 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，所以仅有函数的原型不够。并且，与一般函数不同，内联函数有可能在程序中定义不止一次，此时必须保证在所有源文件中定义完全相同，把内联函数的定义放在头文件中可以确保这一点。

(b) 是函数声明，应该放在头文件中。

**练习 6.44:** 将 6.2.2 节（第 189 页）的 `isShorter` 函数改写成内联函数。

#### 【出题思路】

只需要在普通函数的前面加上关键字 `inline`，就可以将该函数设置为内联了。内联函数在编译时展开，从而消除了调用函数时产生的开销。

#### 【解答】

改写后的内联函数是：

```
inline bool isShorter(const string &s1, const string &s2)
{
    return s1.size() < s2.size();
}
```

**练习 6.45:** 回顾在前面的练习中你编写的那些函数，它们应该是内联函数吗？如果是，将它们改写成内联函数；如果不是，说明原因。

#### 【出题思路】

决定一个函数是否应该是内联函数有很多评判的依据。一般来说，内联机制适用于规模较小、流程直接、频繁调用的函数。一旦函数被定义成内联的，则在编译阶段就展开该函数，以消除运行时产生的额外开销。如果函数的规模很大（比如上百行）不利于展开或者函数只被调用了一两次，那么这样的函数没必要也不应该是内联的。

#### 【解答】

在本章前面实现的函数中，大多规模较小且流程直接，适合于设置为内联函数；如果以后遇到一些代码行数较多的函数，就不适合了。举两个例子：

练习 6.11 中的 `reset` 函数改写后的形式是：

```
inline void reset(int &i)
{
    i = 0;
}
```

练习 6.21 中的 myCompare 函数改写后的形式是：

```
inline int myCompare(const int val, const int *p)
{
    return (val > *p) ? val : *p;
}
```

**练习 6.46：**能把 isShorter 函数定义成 constexpr 函数吗？如果能，将它改成 constexpr 函数；如果不能，说明原因。

#### 【出题思路】

constexpr 函数是指能用于常量表达式的函数，constexpr 函数的返回类型和所有形参的类型都得是字面值类型，而且函数体中必须有且只有一条 return 语句。

#### 【解答】

显然 isShorter 函数不符合 constexpr 函数的要求，它虽然只有一条 return 语句，但是返回的结果调用了标准库 string 类的 size() 函数和 < 比较符，无法构成常量表达式，因此不能改写成 constexpr 函数。

**练习 6.47：**改写 6.3.2 节（205 页）练习中使用递归输出 vector 内容的程序，使其有条件地输出与执行过程有关的信息。例如，每次调用时输出 vector 对象的大小。分别在打开和关闭调试器的情况下编译并执行这个程序。

#### 【出题思路】

本题旨在考查如何在程序中打开和关闭调试器。

#### 【解答】

满足题意的程序如下所示：

```
#include <iostream>
#include <vector>
using namespace std;
// 递归函数输出 vector<int> 的内容
void print(vector<int> vInt, unsigned index)
{
    unsigned sz = vInt.size();
    // 设置在此处输出调试信息
#ifndef NDEBUG
    cout << "vector 对象的大小是：" << sz << endl;
#endif // NDEBUG
    if(!vInt.empty() && index < sz)
    {
        cout << vInt[index] << endl;
        print(vInt, index + 1);
    }
}
int main()
```

```

{
    vector<int> v = {1,3,5,7,9,11,13,15};
    print(v, 0);
    return 0;
}

```

打开调试器时，每次递归调用 `print` 函数都会输出“`vector` 对象的大小是:8”；关闭调试器时，程序只输出 `vector` 对象的内容，不再输出其大小。

**练习 6.48：**说明下面这个循环的含义，它对 `assert` 的使用合理吗？

```

string s;
while (cin >> s && s != sought) {} // 空函数体
assert(cin);

```

#### 【出题思路】

`assert` 是一种预处理宏，当 `assert` 的条件为真时什么也不做，当它的条件为假时输出信息并终止程序。

#### 【解答】

该程序对 `assert` 的使用有不合理之处。在调试器打开的情况下，当用户输入字符串 `s` 并且 `s` 的内容与 `sought` 不相等时，执行循环体，否则继续执行 `assert(cin);` 语句。换句话说，程序执行到 `assert` 的原因可能有两个，一是用户终止了输入，二是用户输入的内容正好与 `sought` 的内容一样。如果用户尝试终止输入（事实上用户总有停止输入结束程序的时候），则 `assert` 的条件为假，输出错误信息，这与程序的原意是不相符的。

当调试器关闭时，`assert` 什么也不做。

**练习 6.49：**什么是候选函数？什么是可行函数？

#### 【出题思路】

本题考查候选函数和可行函数的含义，理解函数调用和匹配的过程，尤其是当程序中存在一组重载函数时。

#### 【解答】

当程序中存在多个同名的重载函数时，编译器需要判断调用的是其中哪个函数，这时就有了候选函数和可行函数两个概念。

函数匹配的第一步是选定本次调用对应的重载函数集，集合中的函数称为候选函数。候选函数具备两个典型特征：一是与被调用的函数同名，二是其声明在调用点可见。

函数匹配的第二步是考查本次调用提供的实参，然后从候选函数中选出能被这组实参调用的函数，这些新选出的函数称为可行函数。可行函数也有两个特征：一是其形参数量与本次调用提供的实参数量相等，二是每个实参的类型与对应的形参类型相同或者能转换成形参的类型。

**练习 6.50:** 已知有第 217 页对函数 `f` 的声明，对于下面的每一个调用列出可行函数。其中哪个函数是最佳匹配？如果调用不合法，是因为没有可匹配的函数还是因为调用具有二义性？

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

#### 【出题思路】

该题需要明确两点：什么是可行函数？满足什么条件的函数是最佳匹配？

#### 【解答】

可行函数是指形参数量与本次调用提供的实参数量相等且每个实参的类型都与对应的形参类型相同或者能转换成形参类型的函数。

最佳匹配是指该函数每个实参的匹配都不劣于其他可行函数需要的匹配且至少有一个实参的匹配优于其他可行函数提供的匹配。

根据上述分析，我们可以推断出：

`f(2.56, 42)` 的可行函数是 `void f(int, int)` 和 `void f(double, double = 3.14)`。但是最佳匹配不存在，因为这两个可行函数各有所长。对于这次调用来说，如果只考虑第一个实参 2.56，我们发现，`void f(double, double = 3.14)` 能够精确匹配，但是要想匹配第二个参数，`int` 类型的实参 42 必须转换成 `double` 类型。如果考虑第二个实参 42，我们发现，`void f(int, int)` 能够精确匹配，但是要想调用 `void f(int, int)` 就必须把第一个 `double` 类型的实参 2.56 转换成 `int` 类型。最终的结果是这两个可行函数各自在一个实参上实现了更好的匹配，但是把它们比较起来无从判断孰优孰劣，因此编译器将因为这个调用具有二义性而拒绝其请求。

`f(42)` 的可行函数是 `void f(int)` 和 `void f(double, double = 3.14)`，其中最佳匹配是 `void f(int)`，因为参数无须做任何类型转换。

`f(42, 0)` 的可行函数是 `void f(int, int)` 和 `void f(double, double = 3.14)`，其中最佳匹配是 `void f(int, int)`，因为参数无须做任何类型转换。

`f(2.56, 3.14)` 的可行函数是 `void f(int, int)` 和 `void f(double, double = 3.14)`，其中最佳匹配是 `void f(double, double = 3.14)`，因为参数无须做任何类型转换。

**练习 6.51:** 编写函数 `f` 的 4 个版本，令其各输出一条可以区分的消息。验证上一个练习的答案，如果你回答错了，反复研究本节的内容直到你弄清自己错在何处。

#### 【出题思路】

只需在函数 `f` 的每个版本中输出其参数的数量和类型即可。

#### 【解答】

满足题意的程序如下所示：

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

```
void f()
{
    cout << "该函数无须参数" << endl;
}
void f(int)
{
    cout << "该函数有一个整型参数" << endl;
}
void f(int, int)
{
    cout << "该函数有两个整型参数" << endl;
}
void f(double a, double b = 3.14)
{
    cout << "该函数有两个双精度浮点型参数" << endl;
}

int main()
{
    f(2.56, 42);
    f(42);
    f(42, 0);
    f(2.56, 3.14);
    return 0;
}
```

**练习 6.52:** 已知有如下声明，

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

请指出下列调用中每个类型转换的等级（参见 6.6.1 节，第 219 页）。



### 【出题思路】

为了确定最佳匹配，编译器将实参类型到形参类型的转换划分成几个等级，读者需要掌握精确匹配、`const` 匹配、类型提升匹配、算术类型转换和类类型转换的使用。

### 【解答】

- (a)发生的参数类型转换是类型提升，字符型实参自动提升成整型。
  - (b)发生的参数类型转换是算术类型转换，双精度浮点型自动转换成整型。

**练习 6.53:** 说明下列每组声明中的第二条语句会产生什么影响，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);

**【出题思路】**

考查顶层 `const` 和底层 `const` 对函数重载的影响。

**【本题解答】**

(a)是合法的，两个函数的区别是它们的引用类型的形参是否引用了常量，属于底层 `const`，可以把两个函数区分开来。

(b)是合法的，两个函数的区别是它们的指针类型的形参是否指向了常量，属于底层 `const`，可以把两个函数区分开来。

(c)是非法的，两个函数的区别是它们的指针类型的形参本身是否是常量，属于顶层 `const`，根据本节介绍的匹配规则可知，向实参添加顶层 `const` 或者从实参中删除顶层 `const` 属于精确匹配，无法区分两个函数。

**练习 6.54：**编写函数的声明，令其接受两个 `int` 形参并且返回类型也是 `int`；然后声明一个 `vector` 对象，令其元素是指向该函数的指针。

**【出题思路】**

考查函数指针的声明和使用。

**【解答】**

满足题意的函数如下所示：

```
int func(int, int);
```

满足题意的 `vector` 对象如下所示：

```
vector<decltype(func)*> vF;
```

**练习 6.55：**编写 4 个函数，分别对两个 `int` 值执行加、减、乘、除运算；在上一题创建的 `vector` 对象中保存指向这些值的指针。

**【出题思路】**

考查函数指针的声明和使用。

**【解答】**

满足题意的函数和 `vector` 对象如下所示：

```
#include <iostream>
#include <vector>
using namespace std;
// 加法
int func1(int a, int b)
{
    return a + b;
}
// 减法
int func2(int a, int b)
{
    return a - b;
}
// 乘法
```

```

int func3(int a, int b)
{
    return a * b;
}
// 除法
int func4(int a, int b)
{
    return a / b;
}

int main()
{
    decltype(func1) *p1 = func1, *p2 = func2, *p3 = func3, *p4 = func4;
    vector<decltype(func1)*> vF = {p1, p2, p3, p4};
    return 0;
}

```

**练习 6.56:** 调用上述 `vector` 对象中的每个元素并输出其结果。

**【出题思路】**

构建一个新的函数，以指向 4 种运算的函数的指针为参数。

**【解答】**

满足题意的程序如下所示：

```

#include <iostream>
#include <vector>
using namespace std;
// 加法
int func1(int a, int b)
{
    return a + b;
}
// 减法
int func2(int a, int b)
{
    return a - b;
}
// 乘法
int func3(int a, int b)
{
    return a * b;
}
// 除法
int func4(int a, int b)
{
    return a / b;
}

void Compute(int a, int b, int (*p)(int, int))
{
    cout << p(a,b) << endl;
}

```

```
int main()
{
    int i = 5, j = 10;
    decltype(func1) *p1 = func1, *p2 = func2, *p3 = func3, *p4 = func4;
    vector<decltype(func1)*> vF = {p1, p2, p3, p4};
    for(auto p : vF) // 遍历 vector 中的每个元素，依次调用四则运算函数
    {
        Compute(i, j, p);
    }
    return 0;
}
```

# 第 7 章

# 类

## 导读

---

本章介绍类的基础知识，主要包括：

- 如何设计一个类，如何定义类及类的成员。
- 类的成员具有哪些访问权限，如何声明并使用友元。
- 构造函数作用何在，默认构造函数、委托构造函数是什么。
- 类的静态成员。

总的来说，与前面几章相比，本章习题中需要编程实现的较少，多数习题旨在考查读者对于类的基本原理是否熟悉。因此在完成本章习题时，应该多比较、多分析，甚至需要回顾前面几章的知识点，综合考虑后给出令人满意的解答。

通过阅读本章内容并完成相应练习，为学习类的高级知识做好准备。

---

**练习 7.1：**使用 2.6.1 节练习定义的 Sales\_data 类为 1.6 节（第 21 页）的交易处理程序编写一个新版本。

### 【出题思路】

程序的思路是：只要 ISBN 相同，就不断累加销量并重新计算平均售价，直至输入新的书籍为止。

### 【解答】

满足题意的程序如下所示：

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

int main()
{
```

```

cout << "请输入交易记录 (ISBN、销售量、原价、实际售价): " << endl;
Sales_data total;                                // 保存下一条交易记录的变量
// 读入第一条交易记录，并确保有数据可以处理
if (cin >> total)
{
    Sales_data trans;                          // 保存和的变量
    // 读入并处理剩余交易记录
    while (cin >> trans)
    {
        // 如果我们仍在处理相同的书
        if (total.isbn() == trans.isbn())
            total += trans;                    // 更新总销售额
        else
        {
            // 打印前一本书的结果
            cout << total << endl;
            total = trans;                  // total 现在表示下一本书的销售额
        }
    }
    cout << total << endl;                      // 打印最后一本书的结果
}
else
{
    // 没有输入！警告读者
    cerr << "No data?!" << endl;
    return -1;                                  // 表示失败
}
return 0;
}

```

**练习 7.2：**曾在 2.6.2 节的练习（第 67 页）中编写了一个 Sales\_data 类，请向这个类添加 combine 和 isbn 成员。

### 【出题思路】

声明和定义成员函数的方式与普通函数差不多。成员函数的声明必须在类的内部，它的定义可以在类的内部，也可以在类的外部。在本题中，我们把成员函数的定义放在类的内部。

### 【解答】

添加 combine 和 isbn 成员后的 Sales\_data 类是：

```

class Sales_data
{
private:                                     // 定义私有数据成员
    string bookNo;                         // 书籍编号，隐式初始化为空串
    unsigned units_sold = 0;                // 销售量，显式初始化为 0
    double sellingprice = 0.0;              // 原始价格，显式初始化为 0.0
    double saleprice = 0.0;                 // 实售价格，显式初始化为 0.0
    double discount = 0.0;                  // 折扣，显式初始化为 0.0
public:                                       // 定义公有函数成员

```

```

// isbn 函数只有一条语句，返回 bookNo
string isbn() const { return bookNo; }
// combine 函数用于把两个 ISBN 相同的销售记录合并在一起
Sales_data& combine( const Sales_data &rhs )
{
    units_sold += rhs.units_sold;           // 累加书籍的销售量
    saleprice = (rhs.saleprice * rhs.units_sold + saleprice * units_
sold) / (rhs.units_sold + units_sold);   // 重新计算实际销售价格
    if(sellingprice != 0)
        discount = saleprice / sellingprice; // 重新计算实际折扣
    return *this;                          // 返回合并后的结果
}
}

```

**练习 7.3：**修改 7.1.1 节（第 229 页）的交易处理程序，令其使用这些成员。

### 【出题思路】

该程序的作用是累加相同编号的书籍销售记录并输出，直至遇到下一个编号为止。改写程序时，用上一题定义的 isbn 函数获取书籍的编号，用 combine 函数把两条销售记录相加。

### 【解答】

满足题意的程序如下所示：

```

#include <iostream>
#include "Sales_data.h"
using namespace std;

int main()
{
    cout << "请输入交易记录 (ISBN、销售量、原价、实际售价): " << endl;
    Sales_data total;                         // 保存下一条交易记录的变量
    // 读入第一条交易记录，并确保有数据可以处理
    if (cin >> total)
    {
        Sales_data trans;                   // 保存和的变量
        // 读入并处理剩余交易记录
        while (cin >> trans)
        {
            // 如果我们仍在处理相同的书
            if (total.isbn() == trans.isbn())
                total.combine(trans);        // 更新总销售额
            else
            {
                // 打印前一本书的结果
                cout << total << endl;
                total = trans;              // total 现在表示下一本书的销售额
            }
        }
        cout << total << endl;               // 打印最后一本书的结果
    }
}

```

```

    }
else
{
    // 没有输入！警告读者
    cerr << "No data?!" << endl;
    return -1;                                // 表示失败
}
return 0;
}

```

**练习 7.4：**编写一个名为 Person 的类，使其表示人员的姓名和住址。使用 string 对象存放这些元素，接下来的练习将不断充实这个类的其他特征。

#### 【出题思路】

练习定义类并添加必要的数据成员。

#### 【解答】

满足题意的 Person 类是：

```

class Person
{
private:
    string strName;      // 姓名
    string strAddress;   // 地址
};

```

**练习 7.5：**在你的 Person 类中提供一些操作使其能够返回姓名和住址。这些函数是否应该是 const 的呢？解释原因。

#### 【出题思路】

练习向类添加函数成员的方法，理解常量成员函数。

#### 【解答】

修改后的 Person 类是：

```

class Person
{
private:
    string strName;      // 姓名
    string strAddress;   // 地址
public:
    string getName() const { return strName; }          // 返回姓名
    string getAddress() const { return strAddress; }     // 返回地址
};

```

上述两个函数应该被定义成常量成员函数，因为不论返回姓名还是返回地址，在函数体内都只是读取数据成员的值，而不会做任何改变。

**练习 7.6:** 对于函数 add、read 和 print，定义你自己的版本。

### 【出题思路】

参考书中的示例，定义自己的版本，注意读入和输出的具体信息应与类的数据成员保持一致。

### 【解答】

满足题意的 add、read 和 print 函数分别如下所示：

```
Sales_data add(const Sales_data &lhs, const Sales_data &rhs)
{
    Sales_data sum = lhs;
    sum.combine(rhs);
    return sum;
}

std::istream &read(std::istream &is, Sales_data &item)
{
    is >> item.bookNo >> item.units_sold >> item.sellingprice >>
        item.saleprice;
    return is;
}

std::ostream &print(std::ostream &os, const Sales_data &item)
{
    os << item.isbn() << " " << item.units_sold << " " << item.Sellingprice
    << " " << item.saleprice << " " << item.discount;
    return os;
}
```

**练习 7.7:** 使用这些新函数重写 7.1.2 节（第 233 页）练习中的交易处理程序。

### 【出题思路】

用 read 函数替代>>，print 函数替代<<，add 函数替代 combine 函数。

### 【解答】

我们自定义的 print 函数不负责输出回车符，满足题意的程序如下所示：

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

int main()
{
    cout << "请输入交易记录 (ISBN、销售量、原价、实际售价): " << endl;
    Sales_data total; // 保存下一条交易记录的变量
    // 读入第一条交易记录，并确保有数据可以处理
    if ( read(cin, total) )
    {
        Sales_data trans; // 保存和的变量
        // 读入并处理剩余交易记录
        while ( read(cin, trans) )
        {
```

```

// 如果我们仍在处理相同的书
if (total.isbn() == trans.isbn())
    total = add(total, trans); // 更新总销售额
else
{
    // 打印前一本书的结果
    print(cout, total);
    cout << endl;
    total = trans;           // total 现在表示下一本的销售额
}
print(cout, total);
cout << endl;             // 打印最后一本书的结果
}
else
{
    // 没有输入! 警告读者
    cerr << "No data?!" << endl;
    return -1;                // 表示失败
}
return 0;
}

```

**练习 7.8:**为什么 `read` 函数将其 `Sales_data` 参数定义成普通的引用,而 `print` 将其参数定义成常量引用?

#### 【出题思路】

本题考查理解 `Sales_data` 类中输入输出函数的原理。

#### 【解答】

`read` 函数将其 `Sales_data` 参数定义成普通的引用是因为我们需要从标准输入流中读取数据并将其写入到给定的 `Sales_data` 对象,因此需要有修改对象的权限。而 `print` 将其参数定义成常量引用是因为它只负责数据的输出,不对其做任何更改。

**练习 7.9:**对于 7.1.2 节(第 233 页)练习中的代码,添加读取和打印 `Person` 对象的操作。

#### 【出题思路】

仿照 `Sales_data` 类,为 `Person` 类添加相应的 `read` 和 `print` 函数。

#### 【解答】

满足题意的 `read` 和 `print` 函数如下所示:

```

std::istream &read(std::istream &is, Person &per)
{
    is >> per.strName >> per.strAddress;
    return is;
}

```

```
std::ostream &print(std::ostream &os, const Person &per)
{
    os << per.getName() << per.getAddress();
    return os;
}
```

**练习 7.10:** 在下面这条 if 语句中，条件部分的作用是什么？

```
if (read(read(cin, data1), data2))
```

### 【出题思路】

read 函数的返回类型是 std::istream &，体会这里使用引用的作用。

### 【解答】

因为 read 函数的返回类型是引用，所以 read(cin, data1) 的返回值可以继续作为外层 read 函数的实参使用。该条件检验读入 data1 和 data2 的过程是否正确，如果正确，条件满足；否则条件不满足。

**练习 7.11:** 在你的 Sales\_data 类中添加构造函数，然后编写一段程序令其用到每个构造函数。

### 【出题思路】

在不同情况下，初始化 Sales\_data 对象所需的数据有所不同，分别为其设计构造函数，同时也利用 C++11 新标准提供的=default 定义默认构造函数。

### 【解答】

满足题意的 4 个构造函数分别如下所示：

```
class Sales_data
{
public:                                // 构造函数的 4 种形式
    Sales_data() = default;
    Sales_data(const std::string &book): bookNo(book) { }
    Sales_data(const std::string &book, const unsigned num,
               const double sellp, const double salep);
    Sales_data(std::istream &is);
public:
    std::string bookNo;                  // 书籍编号，隐式初始化为空串
    unsigned units_sold = 0;             // 销售量，显式初始化为 0
    double sellingprice = 0.0;          // 原始价格，显式初始化为 0.0
    double saleprice = 0.0;              // 实售价格，显式初始化为 0.0
    double discount = 0.0;               // 折扣，显式初始化为 0.0
};

Sales_data::Sales_data(const std::string &book, const unsigned num,
                      const double sellp, const double salep)
{
    bookNo = book;
    units_sold = num;
    sellingprice = sellp;
```

```

    saleprice = salep;
    if(sellingprice != 0)
        discount = saleprice / sellingprice;      // 计算实际折扣
}

Sales_data::Sales_data(std::istream &is)
{
    is >> *this;
}

```

在类的定义中，我们设计了 4 个构造函数。

第一个构造函数是默认构造函数，它使用了 C++11 新标准提供的= default。它的参数列表为空，即不需要我们提供任何数据也能构造一个对象。

第二个构造函数只接受一个 const string&，表示书籍的 ISBN 编号，编译器赋予其他数据成员类内初始值。

第三个构造函数接受完整的销售记录信息，const string& 表示书籍的 ISBN 编号，const unsigned 表示销售量，后面两个 const double 分别表示书籍的原价和实际售价。

最后一个构造函数接受 istream& 并从中读取书籍的销售信息。

我们在 main 函数中创建 4 个 Sales\_data 对象并依次输出其内容，上面定义的构造函数各被用到了一次。

```

#include <iostream>
#include "Sales_data.h"
using namespace std;

int main()
{
    Sales_data data1;
    Sales_data data2("978-7-121-15535-2");
    Sales_data data3("978-7-121-15535-2", 100, 128, 109);
    Sales_data data4(cin);

    cout << "书籍的销售情况是：" << endl;
    cout << data1 << "\n" << data2 << "\n" << data3 << "\n" << data4
        << "\n";
    return 0;
}

```

**练习 7.12：**把只接受一个 istream 作为参数的构造函数定义移到类的内部。

### 【出题思路】

构造函数既可以定义在类的外部，也可以定义在类的内部。

### 【解答】

按照题目要求，把只接受一个 istream 作为参数的构造函数定义到类的内部之后，类的形式如下所示：

```

class Sales_data
{

```

```

public:                                // 构造函数的 4 种形式
    Sales_data() = default;
    Sales_data(const std::string &book): bookNo(book) { }
    Sales_data(const std::string &book, const unsigned num,
               const double sellp, const double salep);
    Sales_data(std::istream &is) { is >> *this; }
public:
    std::string bookNo;                  // 书籍编号, 隐式初始化为空串
    unsigned units_sold = 0;             // 销售量, 显式初始化为 0
    double sellingprice = 0.0;           // 原始价格, 显式初始化为 0.0
    double saleprice = 0.0;              // 实售价格, 显式初始化为 0.0
    double discount = 0.0;               // 折扣, 显式初始化为 0.0
};

```

**练习 7.13:** 使用 `istream` 构造函数重写第 229 页的程序。

### 【出题思路】

原来的程序使用 `Sales_data` 类的默认构造函数, 本题改为使用接受 `istream` 的构造函数。

### 【解答】

改写后的程序如下所示:

```

#include <iostream>
#include "Sales_data.h"
using namespace std;

int main()
{
    Sales_data total(cin);                      // 保存当前求和结果的变量
    if (cin)
    {
        Sales_data trans(cin);                  // 读入第一条交易
  // 保存下一条交易数据的变量
        do
        {
            if (total.isbn() == trans.isbn()) // 检查 isbn
                total.combine(trans);        // 更新变量 total 当前的值
            else
            {
                print(cout, total) << endl; // 输出结果
                total = trans;             // 处理下一本书
            }
        }while( read(cin, trans) );
        print(cout, total) << endl;             // 输出最后一条交易
    }
    else
    {
        cerr << "No data?!" << endl;          // 没有输入任何信息
    }
    return 0;
}

```

**练习 7.14:** 编写一个构造函数，令其用我们提供的类内初始值显式地初始化成员。

#### 【出题思路】

构造函数初始值列表负责为新创建的对象的一个或几个数据成员赋初值。构造函数初始值是成员名字的一个列表，每个名字后面紧跟括号括起来的成员初始值，不同成员的初始化通过逗号分隔开。

#### 【解答】

使用初始值列表的构造函数是：

```
Sales_data (const std::string &book)
    : bookNo(book), units_sold(0), sellingprice(0), saleprice(0), discount(0) {}
```

**练习 7.15:** 为你的 Person 类添加正确的构造函数。

#### 【出题思路】

仿照 Sales\_data 类，为 Person 类添加默认构造函数、接受两个实参的构造函数和从标准输入流中读取数据的构造函数。

#### 【解答】

添加上述 3 个构造函数之后，新的 Person 类如下所示：

```
class Person
{
private:
    string strName;      // 姓名
    string strAddress;   // 地址
public:
    Person() = default;
    Person(const string &name, const string &add)
    {
        strName = name;
        strAddress = add;
    }
    Person(std::istream &is) { is >> *this; }
public:
    string getName() const { return strName; }           // 返回姓名
    string getAddress() const { return strAddress; }      // 返回地址
};
```

**练习 7.16:** 在类的定义中对于访问说明符出现的位置和次数有限定吗？如果有，是什么？什么样的成员应该定义在 public 说明符之后？什么样的成员应该定义在 private 说明符之后？

#### 【出题思路】

考查访问说明符的用法。

#### 【解答】

在类的定义中，可以包含 0 个或者多个访问说明符，并且对于某个访问说明符能出现多少次以及能出现在哪里都没有严格规定。每个访问说明符指定接下来的成

员的访问级别，有效范围直到出现下一个访问说明符或者到达类的结尾为止。

一般来说，作为接口的一部分，构造函数和一部分成员函数应该定义在 `public` 说明符之后，而数据成员和作为实现部分的函数则应该跟在 `private` 说明符之后。

### 练习 7.17：使用 `class` 和 `struct` 时有区别吗？如果有，是什么？

#### 【出题思路】

`class` 和 `struct` 都可以用来声明类，它们的大多数功能都类似，唯一的区别是默认访问权限不同。

#### 【解答】

类可以在它的第一个访问说明符之前定义成员，对这种成员的访问权限依赖于类定义的方式。如果使用 `struct` 关键字，则定义在第一个访问说明符之前的成员是 `public` 的；相反，如果使用 `class` 关键字，则这些成员是 `private` 的。

### 练习 7.18：封装是何含义？它有什么用处？

#### 【出题思路】

封装、继承、多态是类的三个特性，本题首先考查封装的含义。

#### 【解答】

封装是指保护类的成员不被随意访问的能力。通过把类的实现细节设置为 `private`，我们就能完成类的封装。封装实现了类的接口和实现的分离。

如书中所述，封装有两个重要的优点：一是确保用户代码不会无意间破坏封装对象的状态；二是被封装的类的具体实现细节可以随时改变，而无须调整用户级别的代码。

一旦把数据成员定义成 `private` 的，类的作者就可以比较自由地修改数据了。当实现部分发生改变时，只需要检查类的代码本身以确认这次改变有什么影响；换句话说，只要类的接口不变，用户代码就无须改变。如果数据是 `public` 的，则所有使用了原来数据成员的代码都可能失效，这时我们必须定位并重写所有依赖于老版本实现的代码，之后才能重新使用该程序。

把数据成员的访问权限设成 `private` 还有另外一个好处，这么做能防止由于用户的原因造成数据被破坏。如果我们发现有程序缺陷破坏了对象的状态，则可以在有限的范围内定位缺陷：因为只有实现部分的代码可能产生这样的错误。因此，将错误的搜索限制在有限范围内将能极大地简化更改问题及修正程序等工作。

### 练习 7.19：在你的 `Person` 类中，你将把哪些成员声明成 `public` 的？哪些声明成 `private` 的？解释你这样做的原因。

#### 【出题思路】

根据封装的含义我们知道，作为接口的一部分，构造函数和一部分成员函数应

该定义在 `public` 说明符之后，而数据成员和作为实现部分的函数则应该跟在 `private` 说明符之后。

### 【解答】

根据上述分析，我们把数据成员 `strName` 和 `strAddress` 设置为 `private`，这样可以避免用户程序不经意间修改和破坏它们；同时把构造函数和两个获取数据成员的接口函数设置为 `public`，以便于我们在类的外部访问。

### 练习 7.20：友元在什么时候有用？请分别列举出使用友元的利弊。

#### 【出题思路】

友元为类的非成员接口函数提供了访问其私有成员的能力，这种能力的提升利弊共存。

### 【解答】

当非成员函数确实需要访问类的私有成员时，我们可以把它声明成该类的友元。此时，友元可以“工作在类的内部”，像类的成员一样访问类的所有数据和函数。但是，一旦使用不慎（比如随意设定友元），就有可能破坏类的封装性。

### 练习 7.21：修改你的 `Sales_data` 类使其隐藏实现的细节。你之前编写的关于 `Sales_data` 操作的程序应该继续使用，借助类的新定义重新编译该程序，确保其工作正常。

#### 【出题思路】

本题主要涉及 `read`、`print` 和 `add` 三个函数，在修改之前由于它们不属于类的成员函数，所以一旦把数据成员设置为 `private`，则这三个函数无法访问它们，因而编译不能通过。

### 【解答】

解决方法是把函数 `read`、`print` 和 `add` 设置为 `Sales_data` 类的友元，其形式如下所示，此时即使数据成员的访问说明符是 `private` 也能编译通过。

```
class Sales_data {
    friend Sales_data add(const Sales_data &lhs, const Sales_data &rhs);
    friend std::istream &read(std::istream &is, Sales_data &item);
    friend std::ostream &print(std::ostream &os, const Sales_data &item);
private:
    std::string bookNo;           // 书籍编号，隐式初始化为空串
    unsigned units_sold = 0;      // 销售量，显式初始化为 0
    double sellingprice = 0.0;    // 原始价格，显式初始化为 0.0
    double saleprice = 0.0;       // 实售价格，显式初始化为 0.0
    double discount = 0.0;        // 折扣，显式初始化为 0.0
};
```

这三个函数的实现细节与之前相同，不再赘述。

**练习 7.22:** 修改你的 Person 类使其隐藏实现的细节。

**【出题思路】**

隐藏细节的含义是指把 Person 类的数据成员以及不应该被外部访问的函数成员设置成 `private`。

**【解答】**

到目前为止，我们设计的 Person 类包含两个数据成员、三个构造函数和两个用来获取数据的接口函数。显然，除了数据成员之外其他几个函数都有权被外部程序访问，所以我们通过把数据成员设置为 `private` 来确保类的封装性。

```
class Person
{
private:
    string strName;           // 姓名
    string strAddress;        // 地址
public:
    Person() = default;
    Person(const string &name, const string &add)
    {
        strName = name;
        strAddress = add;
    }
    Person(std::istream &is) { is >> *this; }
public:
    string getName() const { return strName; }           // 返回姓名
    string getAddress() const { return strAddress; }      // 返回地址
};
```

**练习 7.23:** 编写你自己的 Screen 类。

**【出题思路】**

思考 Screen 类应该包含哪些数据成员和函数成员，设置适当的访问权限。

**【解答】**

对于 Screen 类来说，必不可少的数据成员有：屏幕的宽度和高度、屏幕的内容以及光标的当前位置，这与书中的示例是一致的。因此，仅包含数据成员的 Screen 类是：

```
class Screen
{
private:
    unsigned height = 0, width = 0;
    unsigned cursor = 0;
    string contents;
}
```

**练习 7.24:** 给你的 Screen 类添加三个构造函数：一个默认构造函数；另一个构造函数接受宽和高的值，然后将 `contents` 初始化成给定数量的空白；第三个构

造函数接受宽和高的值以及一个字符，该字符作为初始化之后屏幕的内容。

### 【出题思路】

同一个类可以包含多个构造函数，构造函数的定义可以在类的内部也可以在类的外部。

### 【解答】

使用构造函数的列表初始值执行初始化操作，添加构造函数之后的 Screen 类是：

```
class Screen
{
private:
    unsigned height = 0, width = 0;
    unsigned cursor = 0;
    string contents;
public:
    Screen() = default;      // 默认构造函数
    Screen(unsigned ht, unsigned wd) : height(ht), width(wd),
        contents(ht * wd, ' ') { }
    Screen(unsigned ht, unsigned wd, char c)
        : height(ht), width(wd), contents(ht * wd, c) { }
}
```

**练习 7.25：** Screen 能安全地依赖于拷贝和赋值操作的默认版本吗？如果能，为什么？如果不能，为什么？

### 【出题思路】

含有指针数据成员的类一般不宜使用默认的拷贝和赋值操作，如果类的数据成员都是内置类型的，则不受干扰。

### 【解答】

Screen 的 4 个数据成员都是内置类型(string 类定义了拷贝和赋值运算符)，因此直接使用类对象执行拷贝和赋值操作是可以的。

**练习 7.26：** 将 Sales\_data::avg\_price 定义成内联函数。

### 【出题思路】

要想把类的成员函数定义成内联函数，有几种不同的途径。第一种是直接把函数定义放在类的内部，第二种是把函数定义放在类的外部，并且在定义之前显式地指定 inline。

### 【解答】

隐式内联，把 avg\_price 函数的定义放在类的内部：

```
class Sales_data
{
public:
    double avg_price() const
{
```

```

        if (units_sold)
            return revenue/units_sold;
        else
            return 0;
    }
}

显式内联，把 avg_price 函数的定义放在类的外部，并且指定 inline:
class Sales_data
{
    double avg_price() const;
}
inline double Sales_data:: avg_price() const
{
    if (units_sold)
        return revenue/units_sold;
    else
        return 0;
}

```

**练习 7.27:** 给你自己的 Screen 类添加 move、set 和 display 函数，通过执行下面的代码检验你的类是否正确。

```

Screen myScreen(5, 5, 'X');
myScreen.move(4,0).set('#').display(cout);
cout << "\n";
myScreen.display(cout);
cout << "\n";

```

### 【出题思路】

添加 3 个成员函数，注意函数的返回值类型应该是引用类型，在成员函数内部可以直接使用类的数据成员。

### 【解答】

添加 move、set 和 display 函数之后，新的 Screen 类是：

```

class Screen
{
private:
    unsigned height = 0, width = 0;
    unsigned cursor = 0;
    string contents;
public:
    Screen() = default;      // 默认构造函数
    Screen(unsigned ht, unsigned wd) : height(ht), width(wd),
        contents(ht * wd, ' ') { }
    Screen(unsigned ht, unsigned wd, char c)
        : height(ht), width(wd), contents(ht * wd, c) { }
public:
    Screen& move(unsigned r, unsigned c)
    {
        cursor = r * width + c;
        return *this;
    }
}

```

```

Screen& set(char ch)
{
    contents[ cursor ] = ch;
    return *this;
}
Screen& set(unsigned r, unsigned c, char ch)
{
    contents[ r * width + c ] = ch;
    return *this;
}
Screen& display()
{
    cout << contents;
    return *this;
}
};

```

执行本题提供的代码，程序的运行结果是：

```

XXXXXXXXXXXXXXXXXXXX#XXXX
XXXXXXXXXXXXXXXXXXXX#XXXX

```

**练习 7.28：**如果 move、set 和 display 函数的返回类型不是 Screen&而是 Screen，则在上一个练习中将会发生什么情况？

#### 【出题思路】

函数的返回值如果是引用，则表明函数返回的是对象本身；函数的返回值如果不是引用，则表明函数返回的是对象的副本。

#### 【解答】

返回引用的函数是左值的，意味着这些函数返回的是对象本身而非对象的副本。如果我们把一系列这样的操作连接在一起的话，所有这些操作将在同一个对象上执行。

在上一个练习中，move、set 和 display 函数的返回类型都是 Screen&，表示我们首先移动光标至(4, 0)位置，然后将该位置的字符修改为'#'，最后输出 myScreen 的内容。

相反，如果我们把 move、set 和 display 函数的返回类型改成 Screen，则上述函数各自只返回一个临时副本，不会改变 myScreen 的值。

**练习 7.29：**修改你的 Screen 类，令 move、set 和 display 函数返回 Screen，并检查程序的运行结果，在上一个练习中你的推测正确吗？

#### 【出题思路】

在编程环境中验证修改前后的程序差别，注意对比运行结果的变化并思考原因。

#### 【解答】

修改 move、set 和 display 函数的返回类型后运行程序，得到的结果是：

```

XXXXXXXXXXXXXXXXXXXX#XXXX
XXXXXXXXXXXXXXXXXXXX#XXXX

```

与练习 7.27 的结果有细微的差别。

**练习 7.30：**通过 `this` 指针使用成员的做法虽然合法，但是有点多余。讨论显式地使用指针访问成员的优缺点。

#### 【出题思路】

对比使用 `this` 指针访问成员的利弊。

#### 【解答】

通过 `this` 指针访问成员的优点是，可以非常明确地指出访问的是对象的成员，并且可以在成员函数中使用与数据成员同名的形参；缺点是显得多余，代码不够简洁。

**练习 7.31：**定义一对类 X 和 Y，其中 X 包含一个指向 Y 的指针，而 Y 包含一个类型为 X 的对象。

#### 【出题思路】

理解类的声明和定义。声明的作用是告知程序类的名字合法可用；定义的作用是规定类的细节。

#### 【解答】

满足题意的程序如下所示：

```
class X;           // 声明类型 X
class Y;           // 定义类型 Y
{
    X* x;
};
class X           // 定义类型 X
{
    Y y;
};
```

类 X 的声明称为前向声明，它向程序中引入了名字 X 并且指明 X 是一种类类型。对于类型 X 来说，此时我们已知它是一个类类型，但是不清楚它到底包含哪些成员，所以它是一个不完全类型。我们可以定义指向不完全类型的指针，但是无法创建不完全类型的对象。

如果试图写成下面的形式，将引发编译器错误。

```
class Y;
class X
{
    Y y;
};
class Y
{
    X* x;
};
```

此时我们试图在类 X 中创建不完全类型 Y 的对象，编译器给出报错信息：  
`error: field 'y' has incomplete type`

**练习 7.32:** 定义你自己的 Screen 和 Window\_mgr，其中 clear 是 Window\_mgr 的成员，是 Screen 的友元。

### 【出题思路】

类可以把其他类定义成友元，也可以把其他类的成员函数定义成友元。当把成员函数定义成友元时，要特别注意程序的组织结构。

### 【解答】

要想让 clear 函数作为 Screen 的友元，只需要在 Screen 类中做出友元声明即可。本题的真正关键之处是程序的组织结构，我们必须首先定义 Window\_mgr 类，其中声明 clear 函数，但是不能定义它；接下来定义 Screen 类，并且在其中指明 clear 函数是其友元；最后定义 clear 函数。满足题意的程序如下所示：

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

class Window_mgr
{
public:
    void clear();
};

class Screen
{
friend void Window_mgr::clear();
private:
    unsigned height = 0, width = 0;
    unsigned cursor = 0;
    string contents;
public:
    Screen() = default;           // 默认构造函数
    Screen(unsigned ht, unsigned wd, char c)
        : height(ht), width(wd), contents(ht * wd, c) { }
};

void Window_mgr::clear()
{
    Screen myScreen(10, 20, 'X');
    cout << "清理之前 myScreen 的内容是: " << endl;
    cout << myScreen.contents << endl;
    myScreen.contents = "";
    cout << "清理之后 myScreen 的内容是: " << endl;
    cout << myScreen.contents << endl;
}

int main()
{
    Window_mgr w;
    w.clear();
    return 0;
}
```

**练习 7.33:** 如果我们给 Screen 添加一个如下所示的 size 成员将发生什么情况？如果出现了问题，请尝试修改它。

```
pos Screen::size() const
{
    return height * width;
}
```

### 【出题思路】

考查类的作用域。

### 【解答】

如果添加如题目所示的 size 函数将会出现编译错误。因为该函数的返回类型 pos 本身定义在 Screen 类的内部，所以在类的外部无法直接使用 pos。要想使用 pos，需要在它的前面加上作用域 Screen::。修改后的程序是：

```
Screen::pos Screen::size() const
{
    return height * width;
}
```

**练习 7.34:** 如果我们把第 256 页 Screen 类的 pos 的 typedef 放在类的最后一行会发生什么情况？

### 【出题思路】

本题考查用于类成员声明的名字查找。

### 【本题解答】

这样做会导致编译出错，因为对 pos 的使用出现在它的声明之前，此时编译器并不知道 pos 到底是什么含义。

**练习 7.35:** 解释下面代码的含义，说明其中的 Type 和 initVal 分别使用了哪个定义。如果代码存在错误，尝试修改它。

```
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();
    return val;
}
```

### 【出题思路】

理解名字查找与类的作用域的关系，包括用于类成员声明的名字查找和成员定义中的名字查找。

### 【解答】

代码的含义及 Type 和 initVal 的使用请参考注释。

```
typedef string Type;      // 声明类型别名 Type 表示 string
Type initVal();          // 声明函数 initVal, 返回类型是 Type
class Exercise {          // 定义一个新类 Exercise
public:
    typedef double Type; // 在内层作用域重新声明类型别名 Type 表示 double
    Type setVal(Type);   // 声明函数 setVal, 参数和返回值的类型都是 Type
    Type initVal();      // 在内层作用域重新声明函数 initVal, 返回类型是 Type
private:
    int val;             // 声明私有数据成员 val
};
// 定义函数 setVal, 此时的 Type 显然是外层作用域的
Type Exercise::setVal(Type parm) {
    val = parm + initVal(); // 此处使用的是类内的 initVal 函数
    return val;
}
```

其中，在 Exercise 类的内部，函数 setVal 和 initVal 用到的 Type 都是 Exercise 内部声明的类型别名，对应的实际类型是 double。

在 Exercise 类的外部，定义 Exercise::setVal 函数时形参类型 Type 用的是 Exercise 内部定义的别名，对应 double；返回类型 Type 用的是全局作用域的别名，对应 string。使用的 initVal 函数是 Exercise 类内定义的版本。

编译上述程序时在 setVal 的定义处发生错误，此处定义的函数形参类型是 double、返回值类型是 string，而类内声明的同名函数形参类型是 double、返回值类型也是 double，二者无法匹配。修改的措施是在定义 setVal 函数时使用作用域运算符强制指定函数的返回值类型。

```
Exercise::Type Exercise::setVal(Type parm) {
    val = parm + initVal(); // 此处使用的是类内的 initVal 函数
    return val;
}
```

**练习 7.36：**下面的初始值是错误的，请找出问题所在并尝试修改它。

```
struct X {
    X (int i, int j): base(i), rem(base % j) { }
    int rem, base;
};
```

### 【出题思路】

本题旨在考查使用构造函数初始值列表时成员的初始化顺序，初始化顺序只与数据成员在类中出现的次序有关，而与初始值列表的顺序无关。

### 【解答】

在类 X 中，两个数据成员出现的顺序是 rem 在前，base 在后，所以当执行 X

对象的初始化操作时先初始化 rem。如上述代码所示，初始化 rem 要用到 base 的值，而此时 base 尚未被初始化，因此会出现错误。该过程与构造函数初始值列表中谁出现在前面谁出现在后面没有任何关系。

修改的方法很简单，只需要把变量 rem 和 base 的次序调换即可，形式是：

```
struct X {
    X (int i, int j): base(i), rem(base % j) { }
    int base, rem;
};
```

**练习 7.37：**使用本节提供的 Sales\_data 类，确定初始化下面的变量时分别使用了哪个构造函数，然后罗列出每个对象所有数据成员的值。

```
Sales_data first_item(cin);

int main() {
    Sales_data next;
    Sales_data last("9-999-99999-9");
}
```

#### 【出题思路】

根据实参的不同调用实现了最佳匹配的构造函数，对于没有提供实参的成员使用其类内初始值进行初始化。

#### 【解答】

`Sales_data first_item(cin);` 使用了接受 `std::istream&` 参数的构造函数，该对象的成员值依赖于用户的输入。

`Sales_data next;` 使用了 `Sales_data` 的默认构造函数，其中 `string` 类型的成员 `bookNo` 默认初始化为空字符串，其他几个成员使用类内初始值初始化为 0。

`Sales_data last("9-999-99999-9");` 使用了接受 `const string&` 参数的构造函数，其中 `bookNo` 使用实参初始化为 "9-999-99999-9"，其他几个成员使用类内初始值初始化为 0。

**练习 7.38：**有些情况下我们希望提供 `cin` 作为接受 `istream&` 参数的构造函数的默认实参，请声明这样的构造函数。

#### 【出题思路】

可以直接在函数声明的地方为 `istream&` 类型的参数设置默认实参 `cin`。

#### 【解答】

满足题意的构造函数如下所示：

```
Sales_data(std::istream &is=std::cin) { is >> *this; }
```

此时该函数具有了默认构造函数的作用，因此我们原来声明的默认构造函数 `Sales_data() = default;` 应该去掉，否则会引起调用的二义性。

**练习 7.39:** 如果接受 `string` 的构造函数和接受 `istream&` 的构造函数都使用默认实参，这种行为合法吗？如果不，为什么？

#### 【出题思路】

本题考查使用默认实参对构造函数的影响。

#### 【解答】

如果我们为构造函数的全部形参都提供了默认实参（包括为只接受一个形参的构造函数提供默认实参），则该构造函数同时具备了默认构造函数的作用。此时即使我们不提供任何实参地创建类的对象，也可以找到可用的构造函数。

然而，如果按照本题的叙述，我们为两个构造函数同样都赋予了默认实参，则这两个构造函数都具有了默认构造函数的作用。一旦我们不提供任何实参地创建类的对象，则编译器无法判断这两个（重载的）构造函数哪个更好，从而出现了二义性错误。

**练习 7.40:** 从下面的抽象概念中选择一个（或者你自己指定一个），思考这样的类需要哪些数据成员，提供一组合理的构造函数并阐明这样做的原因。

- |             |            |              |
|-------------|------------|--------------|
| (a) Book    | (b) Date   | (c) Employee |
| (d) Vehicle | (e) Object | (f) Tree     |

#### 【出题思路】

掌握创建类类型的方法，理解不同构造函数的区别。

#### 【解答】

首先选择(a) `Book`，一本书通常包含书名、ISBN 编号、定价、作者、出版社等信息，因此令其数据成员为：`Name`、`ISBN`、`Price`、`Author`、`Publisher`，其中 `Price` 是 `double` 类型，其他都是 `string` 类型。`Book` 的构造函数有三个：一个默认构造函数、一个包含完整书籍信息的构造函数和一个接受用户输入的构造函数。其定义如下：

```
class Book
{
private:
    string Name, ISBN, Author, Publisher;
    double Price = 0;
public:
    Book() = default;
    Book(const string &n, const string &I, double pr, const string &a,
        const string &p)
    {
        Name = n;
        ISBN = I;
        Price = pr;
        Author = a;
        Publisher = p;
    }
    Book(std::istream &is) { is >> *this; }
};
```

也可以选择(f) Tree，一棵树通常包含树的名称、存活年份、树高等信息，因此令其数据成员为：Name、Age、Height，其中Name是string类型，Age是unsigned类型，Height是double类型。假如我们不希望由用户输入Tree的信息，则可以去掉接受std::istream&形参的构造函数，只保留默认构造函数和接受全部信息的构造函数。其定义如下：

```
class Tree
{
private:
    string Name;
    unsigned Age = 0;
    double Height = 0;
public:
    Tree() = default;
    Tree(const string &n, unsigned a, double h) : Name(n), Age(a),
        Height(h);
}
```

**练习 7.41：**使用委托构造函数重新编写你的Sales\_data类，给每个构造函数体添加一条语句，令其一旦执行就打印一条信息。用各种可能的方式分别创建Sales\_data对象，认真研究每次输出的信息直到你确实理解了委托构造函数的执行顺序位置。

### 【出题思路】

委托构造函数使用它所属类的其他构造函数执行它自己的初始化过程，或者说它把自己的一些或全部职责委托给了其他构造函数。程序先执行受委托构造函数，然后才执行委托构造函数本身的语句。

### 【解答】

改写后的Sales\_data类及其验证程序如下所示：

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

class Sales_data {
friend std::istream &read(std::istream &is, Sales_data &item);
friend std::ostream &print(std::ostream &os, const Sales_data &item);
public: // 委托构造函数
    Sales_data(const string &book, unsigned num, double sellp, double salep)
: bookNo(book), units_sold(num), sellingprice(sellp), saleprice(salep)
    {
        if(sellingprice)
            discount = saleprice / sellingprice;
        cout << "该构造函数接受书号、销售量、原价、实际售价四个信息" << endl;
    }
    Sales_data() : Sales_data("", 0, 0, 0)
    {
        cout << "该构造函数无须接受任何信息" << endl;
    }
}
```

```

    Sales_data(const string &book): Sales_data(book, 0, 0, 0)
{
    cout << "该构造函数接受书号信息" << endl;
}
Sales_data(std::istream &is) : Sales_data()
{
    read(is, *this);
    cout << "该构造函数接受用户输入的信息" << endl;
}

private:
    std::string bookNo;           // 书籍编号, 隐式初始化为空串
    unsigned units_sold = 0;      // 销售量, 显式初始化为 0
    double sellingprice = 0.0;    // 原始价格, 显式初始化为 0.0
    double saleprice = 0.0;       // 实售价格, 显式初始化为 0.0
    double discount = 0.0;        // 折扣, 显式初始化为 0.0
};

std::istream &read(std::istream &is, Sales_data &item)
{
    is >> item.bookNo >> item.units_sold >> item.sellingprice >>
        item.saleprice;
    return is;
}

std::ostream &print(std::ostream &os, const Sales_data &item)
{
    os << item.bookNo << " " << item.units_sold << " " << item.sellingprice
        << " " << item.saleprice << " " << item.discount;
    return os;
}

int main()
{
    Sales_data fist("978-7-121-15535-2", 85, 128, 109);
    Sales_data second;
    Sales_data third("978-7-121-15535-2");
    Sales_data last(cin);
    return 0;
}

```

执行上述程序的输出结果是：

该构造函数接受书号、销售量、原价、实际售价四个信息

该构造函数接受书号、销售量、原价、实际售价四个信息

该构造函数无须接受任何信息

该构造函数接受书号、销售量、原价、实际售价四个信息

该构造函数接受书号信息

该构造函数接受书号、销售量、原价、实际售价四个信息

该构造函数无须接受任何信息

a 10 100 85 (此行由用户输入)

该构造函数接受用户输入的信息

**练习 7.42:** 对于你在练习 7.40（参见 7.5.1 节，第 261 页）中编写的类，确定哪些构造函数可以使用委托。如果可以的话，编写委托构造函数。如果不可以，从抽象概念列表中重新选择一个你认为可以使用委托构造函数的，为挑选出的这个概念编写类定义。

### 【出题思路】

委托构造函数是指使用它所属类的其他构造函数执行它自己的初始化过程，因此在类中应该设计一些构造函数使其具备自主的构造函数功能，而把另外一些设计成委托构造函数。

### 【解答】

以练习 7.40 构建的 Book 类为例，我们令其中的构造函数 Book(const string &n, const string &I, double pr, const string &a, const string &p) 为普通构造函数，而令另外两个作为委托构造函数。其具体形式如下所示：

```
class Book
{
private:
    string Name, ISBN, Author, Publisher;
    double Price = 0;
public:
    Book(const string &n, const string &I, double pr, const string &a, const string &p)
        :Name(n), ISBN(I), Price(pr), Author(a), Publisher(p) { }
    Book() : Book("", "", 0, "", "") { }
    Book(std::istream &is) : Book() { is >> *this; }
};
```

**练习 7.43:** 假定有一个名为 NoDefault 的类，它有一个接受 int 的构造函数，但是没有默认构造函数。定义类 C，C 有一个 NoDefault 类型的成员，定义 C 的默认构造函数。

### 【出题思路】

因为 NoDefault 仅有的一个构造函数并不是默认构造函数，所以在类 C 中，不能使用无参数的默认构造函数，那样的话，类 C 的 NoDefault 成员无法正确初始化。

### 【解答】

我们需要为类 C 的构造函数提供一个默认的 int 值作为参数，满足题意的类定义及验证程序如下所示：

```
#include <iostream>
#include <string>
using namespace std;
// 该类型没有显式定义默认构造函数，编译器也不会为它合成一个
class NoDefault
{
public:
    NoDefault(int i)
```

```

    {
        val = i;
    }
    int val;
};

class C
{
public:
    NoDefault nd;
    // 必须显式调用 NoDefault 的带参构造函数初始化 nd
    C(int i = 0) : nd(i) { }

};

int main()
{
    C c;      // 使用了类型 C 的默认构造函数
    cout << c.nd.val << endl;
    return 0;
}

```

**练习 7.44:** 下面这条声明合法吗？如果不，为什么？

```
vector<NoDefault> vec(10);
```

#### 【出题思路】

理解默认构造函数的用法，理解 `vector` 对象是如何定义和初始化的。

#### 【解答】

上述语句的含义是创建一个 `vector` 对象 `vec`，该对象包含 10 个元素，每个元素的类型都是 `NoDefault` 且执行默认初始化。然而，因为我们在类 `NoDefault` 的定义中没有设计默认构造函数，所以所需的默认初始化过程无法执行。编译器会报告这一错误。

**练习 7.45:** 如果在上一个练习中定义的 `vector` 的元素类型是 `C`，则声明合法吗？为什么？

#### 【出题思路】

理解默认构造函数的用法，理解 `vector` 对象是如何定义和初始化的。

#### 【解答】

与上一个练习相比，如果把 `vector` 的元素类型更改为 `C`，则该声明是合法的，这是因为我们给类型 `C` 定义了带参数的默认构造函数，它可以完成声明语句所需的默认初始化操作。

**练习 7.46:** 下面哪些论断是不正确的？为什么？

- (a) 一个类必须至少提供一个构造函数。

- (b) 默认构造函数是参数列表为空的构造函数。
- (c) 如果对于类来说不存在有意义的默认值，则类不应该提供默认构造函数。
- (d) 如果类没有定义默认构造函数，则编译器将为其生成一个并把每个数据成员初始化成相应类型的默认值。

#### 【出题思路】

本题旨在考查读者对默认构造函数原理的熟悉程度。

#### 【解答】

(a) 是错误的，类可以不提供任何构造函数，这时编译器自动实现一个合成的默认构造函数。

(b) 是错误的，如果某个构造函数包含若干形参，但是同时为这些形参都提供了默认实参，则该构造函数也具备默认构造函数的功能。

(c) 是错误的，因为如果一个类没有默认构造函数，也就是说我们定义了该类的某些构造函数但是没有为其设计默认构造函数，则当编译器确实需要隐式地使用默认构造函数时，该类无法使用。所以一般情况下，都应该为类构建一个默认构造函数。

(d) 是错误的，对于编译器合成的默认构造函数来说，类类型的成员执行各自所属类的默认构造函数，内置类型和复合类型的成员只对定义在全局作用域中的对象执行初始化。

**练习 7.47：**说明接受一个 `string` 参数的 `Sales_data` 构造函数是否应该是 `explicit` 的，并解释这样做的优缺点。

#### 【出题思路】

`explicit` 用于抑制类类型的隐式转换，读者需要知道 `explicit` 的长处和不足。

#### 【解答】

接受一个 `string` 参数的 `Sales_data` 构造函数应该是 `explicit` 的，否则，编译器就有可能自动把一个 `string` 对象转换成 `Sales_data` 对象，这种做法显得有些随意，某些时候会与程序员的初衷相违背。

使用 `explicit` 的优点是避免因隐式类类型转换而带来意想不到的错误，缺点是当用户的确需要这样的类类型转换时，不得不使用略显繁琐的方式来实现。

**练习 7.48：**假定 `Sales_data` 的构造函数不是 `explicit` 的，则下述定义将执行什么样的操作？

```
string null_isbn("9-999-99999-9");
Sales_data item1(null_isbn);
Sales_data item2("9-999-99999-9");
```

如果 `Sales_data` 的构造函数是 `explicit` 的，又会发生什么呢？

#### 【出题思路】

构造函数如果不是 `explicit` 的，则 `string` 对象隐式地转换成 `Sales_data` 对象；相反，构造函数如果是 `explicit` 的，则隐式类类型转换不会发生。

#### 【解答】

在本题给出的代码中，第一行创建了一个 `string` 对象，第二行和第三行都是调用 `Sales_data` 的构造函数（该构造函数接受一个 `string`）创建它的对象。此处无须任何类类型转换，所以不论 `Sales_data` 的构造函数是不是 `explicit` 的，`item1` 和 `item2` 都能被正确地创建，它们的 `bookNo` 成员都是 `9-999-99999-9`，其他成员都是 0。

**练习 7.49：**对于 `combine` 函数的三种不同声明，当我们调用 `i.combine(s)` 时分别发生什么情况？其中 `i` 是一个 `Sales_data`，而 `s` 是一个 `string` 对象。

- (a) `Sales_data &combine(Sales_data);`
- (b) `Sales_data &combine(Sales_data&);`
- (c) `Sales_data &combine(const Sales_data&) const;`

#### 【出题思路】

要想使用隐式的类类型转换，必须遵循一系列规定。如果我们试图在一行代码中使用两种转换规则，编译器将报错。

#### 【解答】

(a)是正确的，编译器首先用给定的 `string` 对象 `s` 自动创建一个 `Sales_data` 对象，然后这个新生成的临时对象传给 `combine` 的形参（类型是 `Sales_data`），函数正确执行并返回结果。

(b)无法编译通过，因为 `combine` 函数的参数是一个非常量引用，而 `s` 是一个 `string` 对象，编译器用 `s` 自动创建一个 `Sales_data` 临时对象，但是这个新生成的临时对象无法传递给 `combine` 所需的非常量引用。如果我们把函数声明修改为 `Sales_data &combine(const Sales_data&);` 就可以了。

(c)无法编译通过，因为我们把 `combine` 声明成了常量成员函数，所以该函数无法修改数据成员的值。

**练习 7.50：**确定在你的 `Person` 类中是否有一些构造函数应该是 `explicit` 的。

#### 【出题思路】

`explicit` 的优点是可以避免程序员不期望的隐式类类型转换。

#### 【解答】

我们之前定义的 `Person` 类含有 3 个构造函数，因为前两个构造函数接受的参数个数都不是 1，所以它们不存在隐式转换的问题，当然也不必指定 `explicit`。

`Person` 类的最后一个构造函数 `Person(std::istream &is);` 只接受一个参数，默认情况下它会把读入的数据自动转换成 `Person` 对象。我们更倾向于严格控制 `Person` 对象的生成过程，如果确实需要使用 `Person` 对象，可以明确指定；

在其他情况下则不希望自动类型转换的发生。所以应该把这个构造函数指定为 `explicit` 的。

**练习 7.51:** `vector` 将其单参数的构造函数定义成 `explicit` 的，而 `string` 则不是，你觉得原因何在？

#### 【出题思路】

从参数类型到类类型的自动转换是否有意义依赖于程序员的看法，如果这种转换是自然而然的，则不应该把它定义成 `explicit` 的；如果二者的语义距离较远，则为了避免不必要的转换，应该指定对应的构造函数是 `explicit` 的。

#### 【解答】

`string` 接受的单参数是 `const char*` 类型，如果我们得到了一个常量字符指针（字符数组），则把它看作 `string` 对象是自然而然的过程，编译器自动把参数类型转换成类类型也非常符合逻辑，因此我们无须指定为 `explicit` 的。

与 `string` 相反，`vector` 接受的单参数是 `int` 类型，这个参数的原意是指定 `vector` 的容量。如果我们在本来需要 `vector` 的地方提供一个 `int` 值并且希望这个 `int` 值自动转换成 `vector`，则这个过程显得比较牵强，因此把 `vector` 的单参数构造函数定义成 `explicit` 的更加合理。

**练习 7.52:** 使用 2.6.1 节（第 64 页）的 `Sales_data` 类，解释下面的初始化过程。如果存在问题，尝试修改它。

```
Sales_data item = {"978-0590353403", 25, 15.99};
```

#### 【出题思路】

熟悉聚合类的概念，理解聚合类初始化的过程及对数据成员的要求。

#### 【解答】

程序的意图是对 `item` 执行聚合类初始化操作，用花括号内的值初始化 `item` 的数据成员。然而实际过程与程序的原意不符合，编译器会报错。

这是因为聚合类必须满足一些非常苛刻的条件，其中一项就是没有类内初始值，而在 2.6.1 节给出的定义中，数据成员 `units_sold` 和 `revenue` 都包含类内初始值。只要去掉这两个类内初始值，程序就可以正常运行了。

```
struct Sales_data
{
    string bookNo;
    unsigned units_sold;
    double revenue;
};
```

**练习 7.53:** 定义你自己的 `Debug`。

#### 【出题思路】

本题旨在考查字面值常量类的用法。

### 【解答】

字面值常量类是一种非常特殊的类类型，聚合类是字面值常量类，某些类虽然不是聚合类但在满足书中所提要求的情况下也是字面值常量类。字面值常量类必须至少提供一个 `constexpr` 构造函数。

读者参考书中的例子定义 `Debug` 类即可，这里不再赘述。

**练习 7.54：** `Debug` 中以 `set_` 开头的成员应该被声明成 `constexpr` 吗？如果不，为什么？

### 【出题思路】

理解 `constexpr` 函数的用法。

### 【解答】

这些以 `set_` 开头的成员不能声明成 `constexpr`，这些函数的作用是设置数据成员的值，而 `constexpr` 函数只能包含 `return` 语句，不允许执行其他任务。

**练习 7.55：** 7.5.5 节（第 266 页）的 `Data` 类是字面值常量类吗？请解释原因。

### 【出题思路】

读者需要掌握字面值常量类的判断方法。

### 【解答】

因为 `Data` 类是聚合类，所以它也是一个字面值常量类。

**练习 7.56：** 什么是类的静态成员？它有何优点？静态成员与普通成员有何区别？

### 【出题思路】

本题考查静态成员的含义及用法。

### 【解答】

静态成员是指声明语句之前带有关键字 `static` 的类成员，静态成员不是任意单独对象的组成部分，而是由该类的全体对象所共享。

静态成员的优点包括：作用域位于类的范围之内，避免与其他类的成员或者全局作用域的名字冲突；可以是私有成员，而全局对象不可以；通过阅读程序可以非常容易地看出静态成员与特定类关联，使得程序的含义清晰明了。

静态成员与普通成员的区别主要体现在普通成员与类的对象关联，是某个具体对象的组成部分；而静态成员不从属于任何具体的对象，它由该类的所有对象共享。另外，还有一个细微的区别，静态成员可以作为默认实参，而普通数据成员不能作为默认实参。

**练习 7.57：**编写你自己的 Account 类。

**【出题思路】**

本题练习在自定义的类中使用静态成员。

**【解答】**

如果类的某些（某个）成员从逻辑上来说更应该与类本身关联，而不是与类的具体对象关联，则我们应该把这种成员声明成静态的。在 Account 类中，很明显利率是相对稳定和统一的，应该是静态成员；而开户人以及它的储蓄额则与对象息息相关，不能是静态的。

为了简便起见，我们只给出 Account 类的声明：

```
class Account
{
private:
    string strName;
    double dAmount = 0.0;
    static double dRate;
};
```

**练习 7.58：**下面的静态数据成员的声明和定义有错误吗？请解释原因。

```
// 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;
```

**【出题思路】**

本题旨在考查静态成员的用法。

**【解答】**

本题的程序存在以下几处错误：

在类的内部，rate 和 vec 的初始化是错误的，因为除了静态常量成员之外，其他静态成员不能在类的内部初始化。另外，example.C 文件的两条语句也是错误的，因为在这里我们必须给出静态成员的初始值。

# 第 8 章

# IO 库

## 导读

本章介绍了处理面向流的输入输出的 C++ 标准库类：

- `iostream` 处理控制台 IO
- `fstream` 处理命名文件 IO
- `stringstream` 完成内存 `string` 的 IO

本章的练习主要帮助读者熟悉基本的控制台输入输出、文件输入输出和字符串输入输出，以及条件状态检测和设置等。

**练习 8.1：**编写函数，接受一个 `istream&` 参数，返回值类型也是 `istream&`。此函数须从给定流中读取数据，直至遇到文件结束标识时停止。它将读取的数据打印在标准输出上。完成这些操作后，在返回流之前，对流进行复位，使其处于有效状态。

### 【出题思路】

本题是流的输入输出的基本练习。此外，本节介绍了流的条件状态，本题还对流的结束状态（本题是遇到文件结束符）、错误状态和数据错误状态（例如要求输入整数时输入了字符）的检测和处理进行了练习。

### 【解答】

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

istream & f(istream & in)
{
    int v;
```

```

while (in >> v, !in.eof()) { // 直到遇到文件结束符才停止读取
    if (in.bad())
        throw runtime_error("IO 流错误");
    if (in.fail()) {
        cerr << "数据错误, 请重试: " << endl;
        in.clear();
        in.ignore(100, '\n');
        continue;
    }
    cout << v << endl;
}
in.clear();
return in;
}

int main()
{
    cout << "请输入一些整数, 按 Ctrl+Z 结束" << endl;
    f(cin);
    return 0;
}

```

**练习 8.2:** 测试函数，调用参数为 cin。

**【出题思路】**

通过运行程序来观察流的状态如何产生变化。

**【解答】**

运行过程如下所示：

请输入一些整数, 按 Ctrl+Z 结束

1 2 3

1

2

3

7.6

7

数据错误, 请重试:

a

数据错误, 请重试:

4

4

^Z

```

Process returned 0 (0x0)  execution time : 20.635 s
Press any key to continue.

```

**练习 8.3:** 什么情况下，下面的 while 循环会终止？

```
while (cin >> i) /* ... */
```

**【出题思路】**

进一步理解流的状态的检测方式。

### 【解答】

遇到了文件结束符，或者遇到了IO流错误，或者读入了无效数据。

**练习 8.4：**编写函数，以读模式打开一个文件，将其内容读入到一个 `string` 的 `vector` 中，将每一行作为一个独立的元素存于 `vector` 中。

### 【出题思路】

本题练习文件输入和流的逐行输入，还练习了使用迭代器遍历容器中的元素。

### 【解答】

```
#include <iostream>
#include <fstream>
#include <string>
#include <vector>

using namespace std;

int main()
{
    ifstream in("data");           // 打开文件
    if (!in) {
        cerr << "无法打开输入文件" << endl;
        return -1;
    }

    string line;
    vector<string> words;
    while (getline(in, line)) {    // 从文件中读取一行
        words.push_back(line);     // 添加到 vector 中
    }

    in.close();                    // 输入完毕，关闭文件

    vector<string>::const_iterator it = words.begin(); // 迭代器
    while (it != words.end()) {   // 遍历 vector
        cout << *it << endl;       // 输出 vector 中的元素
        ++it;
    }

    return 0;
}
```

### 【其他解题思路】

`vector` 的遍历还可使用范围 `for` 循环来实现。

**练习 8.5：**重写上面的程序，将每个单词作为一个独立的元素进行存储。

### 【出题思路】

本题练习逐个数据的输入方式。

### 【解答】

将第 18 行的 `while (getline(in, line)) {` 改为 `while (in >> line) {` 即可。

**练习 8.6：**重写 7.1.1 节的书店程序（第 229 页），从一个文件中读取交易记录。

将文件名作为一个参数传递给 `main`（参见 6.2.5 节，第 196 页）。

### 【出题思路】

通过一个较大的例子继续练习文件输入，并练习从命令行获取参数及参数合法性的检测。

### 【解答】

```
#include <iostream>
#include <fstream>
#include "Sales_data.h"

using namespace std;

int main(int argc, char *argv[])
{
    if (argc != 2) {
        cerr << "请给出文件名" << endl;
        return -1;
    }
    ifstream in(argv[1]);
    if (!in) {
        cerr << "无法打开输入文件" << endl;
        return -1;
    }

    Sales_data total; // 保存当前求和结果的变量
    if (read(in, total)) { // 读入第一笔交易
        Sales_data trans; // 保存下一条交易数据的变量
        while(read(in, trans)) { // 读入剩余的交易
            if (total.isbn() == trans.isbn()) // 检查 isbn
                total.combine(trans); // 更新变量 total 当前的值
            else {
                print(cout, total) << endl; // 输出结果
                total = trans; // 处理下一本书
            }
        }
        print(cout, total) << endl; // 输出最后一条交易
    } else {
        cerr << "没有数据" << endl; // 通知用户
    }

    return 0;
}
```

**练习 8.7：**修改上一节的书店程序，将结果保存到一个文件中。将输出文件名作为第二个参数传递给 main 函数。

### 【出题思路】

本题练习文件输出。

### 【解答】

```
#include <iostream>
#include <fstream>
#include "Sales_data.h"

using namespace std;

int main(int argc, char *argv[])
{
    if (argc != 3) {
        cerr << "请给出输入、输出文件名" << endl;
        return -1;
    }
    ifstream in(argv[1]);
    if (!in) {
        cerr << "无法打开输入文件" << endl;
        return -1;
    }

    ofstream out(argv[2]);
    if (!out) {
        cerr << "无法打开输出文件" << endl;
        return -1;
    }

    Sales_data total;
    if (read(in, total)) { // 读入第一条交易
        Sales_data trans;
        while(read(in, trans)) { // 读入剩余的交易
            if (total.isbn() == trans.isbn()) // 检查 isbn
                total.combine(trans); // 更新变量 total 当前的值
            else {
                print(out, total) << endl; // 输出结果
                total = trans; // 处理下一本
            }
        }
        print(out, total); // 输出最后一条交易
    } else { // 没有输入任何信息
        cerr << "没有数据" << endl; // 通知用户
    }

    return 0;
}
```

**练习 8.8:** 修改上一题的程序，将结果追加到给定的文件末尾。对同一个输出文件，运行程序至少两次，检验数据是否得以保留。

**【出题思路】**

本题练习文件的追加模式。

**【解答】**

将上一题程序的第 19 行 `ofstream out l(argv[2]);` 改为 `ofstream out(argv[2], ofstream::app);`。

**练习 8.9:** 使用你为 8.1.2 节（第 281 页）第一个练习所编写的函数打印一个 `istringstream` 对象的内容。

**【出题思路】**

本题练习字符串流的输入。

**【解答】**

```
#include <iostream>
#include <sstream>
#include <string>
#include <stdexcept>

using namespace std;

istream & f(istream & in)
{
    string v;
    while (in >> v, !in.eof()) { // 直到遇到文件结束符才停止读取
        if (in.bad())
            throw runtime_error("IO 流错误");
        if (in.fail()) {
            cerr << "数据错误，请重试：" << endl;
            in.clear();
            in.ignore(100, '\n');
            continue;
        }
        cout << v << endl;
    }
    in.clear();
    return in;
}

int main()
{
    ostringstream msg;
    msg << "C++ Primer 第五版" << endl;
    istringstream in(msg.str());
    f(in);
    return 0;
}
```

**练习 8.10:** 编写程序, 将来自一个文件中的行保存在一个 `vector<string>` 中。然后使用一个 `istringstream` 从 `vector` 读取数据元素, 每次读取一个单词。

### 【出题思路】

本题继续练习字符串流的输入。

### 【解答】

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
#include <vector>

using namespace std;

int main()
{
    ifstream in("data");                                // 打开文件
    if (!in) {
        cerr << "无法打开输入文件" << endl;
        return -1;
    }

    string line;
    vector<string> words;
    while (getline(in, line)) {                          // 从文件中读取一行
        words.push_back(line);                          // 添加到 vector 中
    }

    in.close();   // 输入完毕, 关闭文件

    vector<string>::const_iterator it = words.begin();   // 迭代器
    while (it != words.end()) {                         // 遍历 vector
        istringstream line_str(*it);
        string word;
        while (line_str >> word)           // 通过 istringstream 从 vector 中读取数据
            cout << word << " ";
        cout << endl;
        ++it;
    }

    return 0;
}
```

**练习 8.11:** 本节的程序在外层 `while` 循环中定义了 `istringstream` 对象。如果 `record` 对象定义在循环之外, 你需要对程序进行怎样的修改? 重写程序, 将 `record` 的定义移到 `while` 循环之外, 验证你设想的修改方法是否正确。

### 【出题思路】

本题练习字符串流的重复使用, 每次通过 `str` 成员将流绑定到不同的字符串,

同时还要调用 `clear` 来重置流的状态。

### 【解答】

```
#include <iostream>
#include <sstream>
#include <string>
#include <vector>

using namespace std;

struct PersonInfo {
    string name;
    vector<string> phones;
};

int main()
{
    string line, word;           // 分别保存来自输入的一行和单词
    vector<PersonInfo> people;   // 保存来自输入的所有记录
    istringstream record;

    while (getline(cin, line)) {
        PersonInfo info;         // 创建一个保存此记录数据的对象
        record.clear();          // 重复使用字符串流时，每次都要调用 clear
        record.str(line);         // 将记录绑定到刚读入的行
        record >> info.name;      // 读取名字
        while (record >> word) // 读取电话号码
            info.phones.push_back(word); // 保持它们

        people.push_back(info);    // 将此记录追加到 people 末尾
    }

    return 0;
}
```

### 【其他解题思路】

还可通过 `rdbuf` 成员函数来获得 `record` 流的缓冲区（一个 `stringbuf` 对象的指针），然后利用 `stringbuf` 的 `str` 成员函数来直接设置缓冲区（字符串）内容。注意，此方法仍然需要调用 `clear` 来将流复位。

### 练习 8.12：我们为什么没有在 `PersonInfo` 中使用类内初始化？

#### 【出题思路】

体会根据应用特点决定程序设计策略。

#### 【解答】

由于每个人的电话号码数量不固定，因此更好的方式不是通过类内初始化指定人名和所有电话号码，而是在缺省初始化之后，在程序中设置人名并逐个添加电话号码。

**练习 8.13:** 重写本节的电话号码程序，从一个命名文件而非 cin 读取数据。

**【出题思路】**

本题练习文件流和字符串流输入输出的综合应用。

**【解答】**

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
#include <vector>

using namespace std;

struct PersonInfo {
    string name;
    vector<string> phones;
};

string format(const string &s) { return s; }

bool valid(const string &s)
{
    // 如何验证电话号码将在第 17 章介绍
    // 现在简单返回 true
    return true;
}

int main(int argc, char *argv[])
{
    string line, word;                                // 分别保存来自输入的一行和单词
    vector<PersonInfo> people;                         // 保存来自输入的所有记录
    istringstream record;

    if (argc != 2) {
        cerr << "请给出文件名" << endl;
        return -1;
    }
    ifstream in(argv[1]);
    if (!in) {
        cerr << "无法打开输入文件" << endl;
        return -1;
    }

    while (getline(in, line)) {
        PersonInfo info;                // 创建一个保存此记录数据的对象
        record.clear();                 // 重复使用字符串流时，每次都要调用 clear
        record.str(line);               // 将记录绑定到刚读入的行
        record >> info.name;            // 读取名字
        while (record >> word) // 读取电话号码
            info.phones.push_back(word); // 保持它们

        people.push_back(info);          // 将此记录追加到 people 末尾
    }
}
```

```

    }

ostringstream os;
for (const auto &entry : people) {           // 对 people 中每一项
    ostringstream formatted, badNums;          // 每个循环步创建的对象
    for (const auto &nums : entry.phones) {     // 对每个数
        if (!valid(nums)) {
            badNums << " " << nums; // 将数的字符串形式存入 badNums
        } else
            // 将格式化的字符串“写入”formatted
            formatted << " " << format(nums);
    }
    if (badNums.str().empty())                  // 没有错误的数
        os << entry.name << " "                // 打印名字
        << formatted.str() << endl;             // 和格式化的数
    else
        // 否则，打印名字和错误的数
        cerr << "input error: " << entry.name
        << " invalid number(s) " << badNums.str() << endl;
}
cout << os.str() << endl;

return 0;
}

```

**练习 8.14：**我们为什么将 `entry` 和 `nums` 定义为 `const auto&`？

**【出题思路】**

回顾范围 `for` 语句的相关内容。

**【解答】**

这两条语句分别使用范围 `for` 语句枚举 `people` 中所有项（人）和每项的 `phones` 中的所有项（电话号码）。使用 `const` 表明在循环中不会改变这些项的值；`auto` 是请求编译器依据 `vector` 元素类型来推断出 `entry` 和 `nums` 的类型，既简化代码又避免出错；使用引用的原因是，`people` 和 `phones` 的元素分别是结构对象和字符串对象，使用引用即可避免对象拷贝。

# 第 9 章

## 顺序容器

### 导读

本章介绍了标准库顺序容器，包括：

- 顺序容器的公共标准接口，如构造函数、添加/删除操作、大小操作、获取位置的操作等。
- 利用迭代器访问容器，及需要注意的一些问题。
- 不同顺序容器的一些差异。
- `string` 的特殊操作。
- 容器适配器，如栈、队列等。

本章的练习主要帮助读者熟练掌握顺序容器的构造、添加、删除、获取大小、获取位置等基本操作；熟悉通过迭代器访问顺序容器的方法，了解一些限制；了解不同顺序容器在操作、迭代器等方面的差异；掌握用 `string` 特殊操作进行字符串处理；以及用栈解决实际问题。

**练习 9.1：**对于下面的程序任务，`vector`、`deque` 和 `list` 哪种容器最为适合？解释你的选择的理由。如果没有哪一种容器优于其他容器，也请解释原因。

(a) 读取固定数量的单词，将它们按字典序插入到容器中。我们将在下一章中看到，关联容器更适合这个问题。

(b) 读取未知数量的单词，总是将新单词插入到末尾。删除操作在头部进行。

(c) 从一个文件读取未知数量的整数。将这些数排序，然后将它们打印到标准输出。

### 【出题思路】

学习数据结构很重要的一点是清晰地掌握数据结构之上各种操作的时间复杂性，并据此分析求解问题不同算法的优劣。本题即是熟悉几种常见容器的插入、删除等

基本操作的时间复杂性，并练习据此选择容器求解实际问题。

### 【解答】

(a) “按字典序插入到容器中”意味着进行插入排序操作，从而需要在容器内部频繁进行插入操作，`vector` 在尾部之外的位置插入和删除元素很慢，`deque` 在头尾之外的位置插入和删除元素很慢，而`list` 在任何位置插入、删除速度都很快。因此，这个任务选择`list`更为适合。当然，如果不是必须边读取单词边插入到容器中，可以使用`vector`，将读入的单词依次追加到尾部，读取完毕后，调用标准库到排序算法将单词重排为字典序。

(b) 由于需要在头、尾分别进行插入、删除操作，因此将`vector`排除在外，`deque`和`list`都可以达到很好的性能。如果还需要频繁进行随机访问，则`deque`更好。

(c) 由于整数占用空间很小，且快速的排序算法需频繁随机访问元素，将`list`排除在外。由于无须在头部进行插入、删除操作，因此使用`vector`即可，无须使用`deque`。

### 练习 9.2：定义一个`list`对象，其元素类型是`int`的`deque`。

#### 【出题思路】

本题练习容器的定义。

#### 【解答】

```
list<deque<int>> a;
```

### 练习 9.3：构成迭代器范围的迭代器有何限制？

#### 【出题思路】

准确理解怎样的迭代器可以构成迭代器范围。

#### 【解答】

两个迭代器`begin`和`end`必须指向同一个容器中的元素，或者是容器最后一个元素之后的位置；而且，对`begin`反复进行递增操作，可保证到达`end`，即`end`不在`begin`之前。

### 练习 9.4：编写函数，接受一对指向`vector<int>`的迭代器和一个`int`值。在两个迭代器指定的范围内查找给定的值，返回一个布尔值来指出是否找到。

#### 【出题思路】

本题练习遍历迭代器范围的方法。

#### 【解答】

```
#include <iostream>
#include <vector>
```

```

using namespace std;

bool search_vec(vector<int>::iterator beg,
                 vector<int>::iterator end, int val)
{
    for (;beg != end; beg++) // 遍历范围
        if (*beg == val) // 检查是否与给定值相等
            return true;
    return false;
}

int main()
{
    vector<int> ilist = {1, 2, 3, 4, 5, 6, 7};

    cout << search_vec(ilist.begin(), ilist.end(), 3) << endl;
    cout << search_vec(ilist.begin(), ilist.end(), 8) << endl;

    return 0;
}

```

**练习 9.5：**重写上一题的函数，返回一个迭代器指向找到的元素。注意，程序必须处理未找到给定值的情况。

### 【出题思路】

练习如何用迭代器表示搜索成功和搜索失败。

### 【解答】

```

#include <iostream>
#include <vector>

using namespace std;

vector<int>::iterator search_vec(vector<int>::iterator beg,
                                 vector<int>::iterator end, int val)
{
    for (;beg != end; beg++) // 遍历范围
        if (*beg == val) // 检查是否与给定值相等
            return beg; // 搜索成功，返回元素对应迭代器
    return end; // 搜索失败，返回尾后迭代器
}

int main()
{
    vector<int> ilist = {1, 2, 3, 4, 5, 6, 7};

    cout << search_vec(ilist.begin(), ilist.end(), 3) - ilist.begin()
        << endl;
    cout << search_vec(ilist.begin(), ilist.end(), 8) - ilist.begin()
        << endl;

    return 0;
}

```

**练习 9.6:** 下面程序有何错误？你应该如何修改它？

```
list<int> lst1;
list<int>::iterator iter1 = lst1.begin(),
                  iter2 = lst1.end();
while (iter1 < iter2) /* ... */
```

**【出题思路】**

理解不同类型容器的迭代器之间的差别。更深层次的，理解数据结构的实现如何导致迭代器的差别。

**【解答】**

与 `vector` 和 `deque` 不同，`list` 的迭代器不支持`<`运算，只支持递增、递减、`==`以及`!=`运算。

原因在于这几种数据结构实现上的不同。`vector` 和 `deque` 将元素在内存中连续保存，而 `list` 则是将元素以链表方式存储，因此前者可以方便地实现迭代器的大小比较（类似指针的大小比较）来体现元素的前后关系。而在 `list` 中，两个指针的大小关系与它们指向的元素的前后关系并不一定是吻合的，实现`<`运算将会非常困难和低效。

**练习 9.7:** 为了索引 `int` 的 `vector` 中的元素，应该使用什么类型？

**【出题思路】**

标准库容器定义了若干类型成员，对应容器使用中可能涉及的类型，如迭代器、元素引用等。本题和下一题帮助读者理解这些类型。

**【解答】**

使用迭代器类型 `vector<int>::iterator` 来索引 `int` 的 `vector` 中的元素。

**练习 9.8:** 为了读取 `string` 的 `list` 中的元素，应该使用什么类型？如果写入 `list`，又该使用什么类型？

**【出题思路】**

理解容器的类型成员。

**【解答】**

为了读取 `string` 的 `list` 中的元素，应使用 `list<string>::value_type`，因为 `value_type` 表示元素类型。

为了写入数据，需要（非常量）引用类型，因此应使用 `list<string>::reference`。

**练习 9.9:** `begin` 和 `cbegin` 两个函数有什么不同？

**【出题思路】**

`begin` 和 `end` 都有多个版本，分别返回普通、反向和 `const` 迭代器。本题帮助读者理解不同版本间的差异。

**【解答】**

`cbegin` 是 C++ 新标准引入的，用来与 `auto` 结合使用。它返回指向容器第一个元素的 `const` 迭代器，可以用来只读地访问容器元素，但不能对容器元素进行修改。因此，当不需要写访问时，应该使用 `cbegin`。

`begin` 则是被重载过的，有两个版本：其中一个是 `const` 成员函数，也返回 `const` 迭代器；另一个则返回普通迭代器，可以对容器元素进行修改。

**练习 9.10：**下面 4 个对象分别是什么类型？

```
vector<int> v1;
const vector<int> v2;
auto it1 = v1.begin(), it2 = v2.begin();
auto it3 = v1.cbegin(), it4 = v2.cbegin();
```

**【出题思路】**

继续熟悉 `begin` 不同版本的差异和不同类型迭代器的差异，特别是 `begin` 与 `auto` 配合使用时的细微差异。

**【解答】**

`v1` 是 `int` 的 `vector` 类型，我们可以修改 `v1` 的内容，包括添加、删除元素及修改元素值等操作。

`v2` 是 `int` 的常量 `vector` 类型，其内容不能修改，添加、删除元素及修改元素值等均不允许。

`begin` 与 `auto` 结合使用时，会根据调用对象的类型来决定迭代器的类型，因此 `it1` 是普通迭代器，可对容器元素进行读写访问，而 `it2` 是 `const` 迭代器，不能对容器元素进行写访问。

而 `cbegin` 则不管调用对象是什么类型，始终返回 `const` 迭代器，因此 `it3` 和 `it4` 都是 `const` 迭代器。

**练习 9.11：**对 6 种创建和初始化 `vector` 对象的方法，每一种都给出一个实例。解释每个 `vector` 包含什么值。**【出题思路】**

C++11 提供了丰富的容器初始化方式，理解各种初始化方式之间的不同，有助于我们在求解实际问题时选择恰当的方式。

**【解答】**

(1) `vector<int> ilist1;` // 默认初始化，`vector` 为空——`size` 返回 0，表明容器中尚未有元素；`capacity` 返回 0，意味着尚未分配存储空间。这种初始化方式适合于元素个数和值未知，需要在程序运行中动态添加的情况。

(2) `vector<int> ilist2(ilist);` // `ilist2` 初始化为 `ilist` 的拷贝，`ilist` 必须与 `ilist2` 类型相同，即也是 `int` 的 `vector` 类型，`ilist2` 将具有与 `ilist` 相同的容量和元素。

```
vector<int> ilist2_1=ilist; // 等价方式
(3) vector<int> ilist = {1, 2, 3.0, 4, 5, 6, 7}; // ilist
初始化为列表中元素的拷贝，列表中的元素类型必须与 ilist 的元素类型相容，在本例中必须是与整型相容的数值类型。对于整型，会直接拷贝其值，对于其他类型则需进行类型转换（如 3.0 转换为 3）。这种初始化方式适合元素数量和值预先可知的情况。
```

```
vector<int> ilist_1{1, 2, 3.0, 4, 5, 6, 7}; // 等价方式
(4) vector<int> ilist3(ilist.begin() + 2, ilist.end() - 1);
// ilist3 初始化为两个迭代器指定范围中的元素的拷贝，范围中的元素类型必须与 ilist3 的元素类型相容，在本例中 ilist3 被初始化为 {3, 4, 5, 6}。注意，由于只要求范围内元素类型与待初始化的容器的元素类型相容，因此，迭代器来自于不同类型的容器是可能的，例如，用一个 double 的 list 的范围来初始化 ilist3 是可行的。另外，由于构造函数只是读取范围中的元素并进行拷贝，因此使用普通迭代器还是 const 迭代器来指出范围并无区别。这种初始化方法特别适合于获取一个序列的子序列。
```

(5) `vector<int> ilist4(7); // 默认值初始化，ilist4 中将包含 7 个元素，每个元素进行缺省的值初始化，对于 int，也就是被赋值为 0，因此 ilist4 被初始化为包含 7 个 0。当程序运行初期元素大致数量可预知，而元素的值需动态获取时，可采用这种初始化方式。`

(6) `vector<int> ilist5(7, 3); // 指定值初始化，ilist5 被初始化为包含 7 个值为 3 的 int。`

**练习 9.12：**对于接受一个容器创建其拷贝的构造函数，和接受两个迭代器创建拷贝的构造函数，解释它们的不同。

#### 【出题思路】

继续熟悉不同的初始化方式。

#### 【解答】

接受一个已有容器的构造函数会拷贝此容器中的所有元素，这样，初始化完成后，我们得到此容器的一个一模一样的拷贝。当我们确实需要一个容器的完整拷贝时，这种初始化方式非常方便。

但当我们不需要已有容器中的全部元素，而只是想拷贝其中一部分元素时，可使用接受两个迭代器的构造函数。传递给它要拷贝的范围的起始和尾后位置的迭代器，即可令新容器对象包含所需范围中元素的拷贝。

**练习 9.13:** 如何从一个 `list<int>` 初始化一个 `vector<double>`? 从一个 `vector<int>` 又该如何创建? 编写代码验证你的答案。

### 【出题思路】

更深入地理解容器拷贝初始化和范围初始化两种方式的差异。

### 【解答】

由于 `list<int>` 与 `vector<double>` 是不同的容器类型, 因此无法采用容器拷贝初始化方式。但前者的元素类型是 `int`, 与后者的元素类型 `double` 是相容的, 因此可以采用范围初始化方式来构造一个 `vector<double>`, 令它的元素值与 `list<int>` 完全相同。对 `vector<int>` 也是这样的思路。

```
#include <iostream>
#include <vector>
#include <list>

using namespace std;

int main()
{
    list<int> ilist = {1, 2, 3, 4, 5, 6, 7};
    vector<int> ivec = {7, 6, 5, 4, 3, 2, 1};
    // 容器类型不同, 不能使用拷贝初始化
    // vector<double> dvec(ilist);
    // 元素类型相容, 因此可采用范围初始化
    vector<double> dvec(ilist.begin(), ilist.end());
    // 容器类型不同, 不能使用拷贝初始化
    // vector<double> dvec1(ivec);
    // 元素类型相容, 因此可采用范围初始化
    vector<double> dvec1(ivec.begin(), ivec.end());

    cout << dvec.capacity() << " " << dvec.size() << " "
        << dvec[0] << " " << dvec[dvec.size()-1] << endl;
    cout << dvec1.capacity() << " " << dvec1.size() << " "
        << dvec1[0] << " " << dvec1[dvec1.size()-1] << endl;

    return 0;
}
```

**练习 9.14:** 编写程序, 将一个 `list` 中的 `char *` 指针 (指向 C 风格字符串) 元素赋值给一个 `vector` 中的 `string`。

### 【出题思路】

容器有多种赋值操作, 本题帮助读者理解不同赋值方式的差异。

### 【解答】

由于 `list<char *>` 与 `vector<string>` 是不同类型的容器, 因此无法采用赋值运算符`=`来进行元素赋值。但 `char *` 可以转换为 `string`, 因此可以采用范围赋值方式来实现本题要求。

```
#include <iostream>
#include <vector>
#include <list>

using namespace std;

int main()
{
    list<char *> slist = {"hello", "world", "!"};
    vector<string> svec;

    // 容器类型不同，不能直接赋值
    // svec = slist;
    // 元素类型相容，可以采用范围赋值
    svec.assign(slist.begin(), slist.end());

    cout << svec.capacity() << " " << svec.size() << " "
        << svec[0] << " " << svec[svec.size()-1] << endl;

    return 0;
}
```

**练习 9.15：**编写程序，判定两个 `vector<int>` 是否相等。

#### 【出题思路】

练习容器的关系运算符的使用。

#### 【解答】

标准库容器支持关系运算符，比较两个 `vector` 是否相等使用`==`运算符即可。

当两个 `vector` 包含相同个数的元素，且对位元素都相等时，判定两个 `vector` 相等，否则不等。两个 `vector` 的 `capacity` 不会影响相等性判定，因此，当下面程序中 `ivec1` 在添加、删除元素导致扩容后，仍然与 `ivec` 相等。

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector<int> ivec = {1, 2, 3, 4, 5, 6, 7};
    vector<int> ivec1 = {1, 2, 3, 4, 5, 6, 7};
    vector<int> ivec2 = {1, 2, 3, 4, 5};
    vector<int> ivec3 = {1, 2, 3, 4, 5, 6, 8};
    vector<int> ivec4 = {1, 2, 3, 4, 5, 7, 6};

    cout << (ivec == ivec1) << endl;
    cout << (ivec == ivec2) << endl;
    cout << (ivec == ivec3) << endl;
    cout << (ivec == ivec4) << endl;
```

```

ivec1.push_back(8);
ivec1.pop_back();
cout << ivec1.capacity() << " " << ivec1.size() << endl;
cout << (ivec == ivec1) << endl;

return 0;
}

```

**练习 9.16：**重写上一题的程序，比较一个 `list<int>` 中的元素和一个 `vector<int>` 中的元素。

#### 【出题思路】

首先，理解容器关系运算符的限制——必须是相容类型的容器，且元素类型也必须相同才能比较。其次，练习自己编写程序来实现容器内容的比较。

#### 【解答】

两个容器相等的充分条件是包含相同个数的元素，且对位元素的值都相等。因此，首先判断两个容器是否包含相同个数的元素，若不等，则两个容器也不等。否则，遍历两个容器中的元素，两两比较对位元素的值，若有元素不相等，则容器不等。否则，两个容器相等。

```

#include <iostream>
#include <vector>
#include <list>

using namespace std;

bool l_v_equal(vector<int> &ivec, list<int> &ilist) {
    // 比较 list 和 vector 元素个数
    if (ilist.size() != ivec.size())
        return false;

    auto lb = ilist.cbegin();    // list 首元素
    auto le = ilist.cend();      // list 尾后位置
    auto vb = ivec.cbegin();     // vector 首元素
    for (;lb != le; lb++, vb++)
        if (*lb != *vb)          // 元素不等，容器不等
            return false;
    return true;                  // 容器相等
}

int main()
{
    vector<int> ivec = {1, 2, 3, 4, 5, 6, 7};
    list<int> ilist = {1, 2, 3, 4, 5, 6, 7};
    list<int> ilist1 = {1, 2, 3, 4, 5};
    list<int> ilist2 = {1, 2, 3, 4, 5, 6, 8};
    list<int> ilist3 = {1, 2, 3, 4, 5, 7, 6};

    cout << l_v_equal(ivec, ilist) << endl;
    cout << l_v_equal(ivec, ilist1) << endl;
}

```

```

    cout << l_v_equal(ivec, ilist2) << endl;
    cout << l_v_equal(ivec, ilist3) << endl;

    return 0;
}

```

**【其他解题思路】**

首先利用范围初始化创建一个与 `list<int>` 内容相同的 `vector<int>`，然后用`==`运算符比较此 `vector` 和给定的 `vector`。与上一种方法相比，这种方法的优点是直接利用标准库的功能实现，比较简单，且不必为不同容器类型实现不同的版本，但创建容器的临时拷贝会占用额外的内存，且需要额外时间拷贝数据。

**练习 9.17：假定 c1 和 c2 是两个容器，下面的比较操作有何限制(如果有的话)？**

```
if (c1 < c2)
```

**【出题思路】**

理解容器关系运算符对容器类型和元素类型的限制。

**【解答】**

首先，容器类型必须相同，元素类型也必须相同。

其次，元素类型必须支持`<`运算符。

**练习 9.18：编写程序，从标准输入读取 `string` 序列，存入一个 `deque` 中。编写一个循环，用迭代器打印 `deque` 中的元素。****【出题思路】**

本题练习向容器中添加元素，继续练习遍历容器中的元素。重点：不同容器在不同位置添加元素的性能是有差异的。

**【解答】**

对 `deque` 来说，在首尾位置添加新元素性能最佳，在中间位置插入新元素性能会很差。对遍历操作，可高效完成。

```

#include <iostream>
#include <deque>

using namespace std;

int main()
{
    deque<string> sd;      // string 的 deque

    string word;
    while (cin >> word)   // 读取字符串，直至遇到文件结束符
        sd.push_back(word);

    // 用 cbegin() 获取 deque 首元素迭代器，遍历 deque 中所有元素
    for (auto si = sd.cbegin(); si != sd.cend(); si++)
        cout << *si << endl;
}

```

```

    return 0;
}

```

### 【其他解题思路】

由于在 `deque` 的首尾位置添加新元素性能很好，因此可以用 `push_front` 替换 `push_back`，性能不变，但元素在 `deque` 中的顺序将与输入顺序相反。若需保持相同顺序，应使用 `push_back`。

**练习 9.19：**重写上题的程序，用 `list` 替代 `deque`。列出程序要做出哪些改变。

### 【出题思路】

练习不同容器的添加操作的异同。

### 【解答】

对 `list` 来说，在任何位置添加新元素都有很好的性能，遍历操作也能高效完成，因此程序与上一题并无太大差异。

```

#include <iostream>
#include <list>

using namespace std;

int main()
{
    list<string> sl;           // string 的 list

    string word;
    while (cin >> word)      // 读取字符串，直至遇到文件结束符
        sl.push_back(word);

    // 用 cbegin() 获取 deque 首元素迭代器，遍历 deque 中的所有元素
    for (auto si = sl.cbegin(); si != sl.cend(); si++)
        cout << *si << endl;

    return 0;
}

```

### 【其他解题思路】

与上一题一样，可以用 `push_front` 替换 `push_back`。甚至可以用 `insert` 等更复杂的操作，可获得相同的性能，但对本题的简单要求来说，并无必要。

**练习 9.20：**编写程序，从一个 `list<int>` 拷贝元素到两个 `deque` 中。值为偶数的所有元素都拷贝到一个 `deque` 中，而奇数值元素都拷贝到另一个 `deque` 中。

### 【出题思路】

这是一个很简单的数据处理问题的练习。读者可练习多个容器间数据的处理、拷贝。

### 【解答】

通过遍历 `list<int>`, 可检查其中每个元素的奇偶性, 并用 `push_back` 分别添加到目的 `deque` 的末尾。程序中用位与运算检查元素最低位的值, 若为 1, 表明是奇数, 否则即为偶数。

```
#include <iostream>
#include <list>
#include <deque>

using namespace std;

int main()
{
    list<int> ilist = {1, 2, 3, 4, 5, 6, 7, 8}; // 初始化整数 list
    deque<int> odd_d, even_d;

    // 遍历整数 list
    for (auto iter = ilist.cbegin(); iter != ilist.cend(); iter++)
        if (*iter & 1) // 查看最低位, 1: 奇数, 0: 偶数
            odd_d.push_back(*iter);
        else even_d.push_back(*iter);

    cout << "奇数值有: ";
    for (auto iter = odd_d.cbegin(); iter != odd_d.cend(); iter++)
        cout << *iter << " ";
    cout << endl;

    cout << "偶数值有: ";
    for (auto iter = even_d.cbegin(); iter != even_d.cend(); iter++)
        cout << *iter << " ";
    cout << endl;

    return 0;
}
```

### 【其他解题思路】

出于简单和保持顺序考虑, 不必用其他操作代替 `push_back`。对于奇偶性的判定, 可用模 2 运算 “%2” 代替位与运算, 两者是等价的。

**练习 9.21:** 如果我们将第 308 页中使用 `insert` 返回值将元素添加到 `list` 中的循环程序改写为将元素插入到 `vector` 中, 分析循环将如何工作。

### 【出题思路】

本题练习用 `insert` 向容器中添加元素的方法。理解这是最通用的方法, 可以实现 `push_back` 和 `push_front` 这些特殊插入操作的效果。

### 【解答】

在循环之前, `vector` 为空, 此时将 `iter` 初始化为 `vector` 首位置, 与初始化为尾后位置效果是一样的。循环中第一次调用 `insert` 会将读取的第一个 `string` 插入到 `iter` 指向位置之前的位置, 即, 令新元素成为 `vector` 的首元素。而 `insert`

的返回指向此元素的迭代器，我们将它赋予 `iter`，从而使得 `iter` 始终指向 `vector` 的首元素。接下来的每个循环步均是如此，将新 `string` 插入到 `vector` 首元素之前的位置，成为新的首元素，并使 `iter` 始终指向 `vector` 首。这样，`string` 在 `vector` 排列的顺序将与它们的输入顺序恰好相反。整个循环执行的过程和最后的结果都与 `list` 版本没有什么区别。但要注意，在 `list` 首元素之前插入新元素性能很好，但对于 `vector`，这样的操作需要移动所有现有元素，导致性能很差。

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector<string> svec;           // string 的 vector

    string word;
    auto iter = svec.begin();      // 获取 vector 首位置迭代器
    while (cin >> word)          // 读取字符串，直至遇到文件结束符
        iter = svec.insert(iter, word);

    // 用 cbegin() 获取 vector 首元素迭代器，遍历 vector 中所有元素
    for (auto iter = svec.cbegin(); iter != svec.cend(); iter++)
        cout << *iter << endl;

    return 0;
}
```

**练习 9.22：**假定 `iv` 是一个 `int` 的 `vector`，下面的程序存在什么错误？你将如何修改？

```
vector<int>::iterator iter = iv.begin(),
                     mid = iv.begin() + iv.size()/2;
while (iter != mid)
    if (*iter == some_val)
        iv.insert(iter, 2 * some_val);
```

#### 【出题思路】

首先，理解容器插入操作的副作用——向一个 `vector`、`string` 或 `deque` 插入元素会使现有指向容器的迭代器、引用和指针失效。其次，练习如何利用 `insert` 返回的迭代器，使得在向容器插入元素后，仍能正确在容器中进行遍历。

#### 【解答】

循环中未对 `iter` 进行递增操作，`iter` 无法向中点推进。其次，即使加入了 `iter++` 语句，由于向 `iv` 插入元素后，`iter` 已经失效，`iter++` 也不能起到将迭代器向前推进一个元素的作用。修改方法如下：

首先，将 `insert` 返回的迭代器赋予 `iter`，这样，`iter` 将指向新插入的元素 `y`。我们知道，`insert` 将 `y` 插入到 `iter` 原来指向的元素 `x` 之前的位置，因此，接

下来我们需要进行两次 `iter++` 才能将 `iter` 推进到 `x` 之后的位置。

其次，`insert()` 也会使 `mid` 失效，因此，只正确设置 `iter` 仍不能令循环在正确的时候结束，我们还需设置 `mid` 使之指向 `iv` 原来的中央元素。在未插入任何新元素之前，此位置是 `iv.begin() + iv.size() / 2`，我们将此时的 `iv.size()` 的值记录在变量 `org_size` 中。然后在循环过程中统计新插入的元素的个数 `new_ele`，则在任何时候，`iv.begin() + org_size / 2 + new_ele` 都能正确指向 `iv` 原来的中央元素。

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector<int> iv = {1, 1, 2, 1};           // int 的 vector
    int some_val = 1;

    vector<int>::iterator iter = iv.begin();
    int org_size = iv.size(), new_ele = 0;    // 原大小和新元素个数

    // 每个循环步都重新计算“mid”，保证正确指向 iv 原中央元素
    while (iter != (iv.begin() + org_size / 2 + new_ele))
        if (*iter == some_val) {
            iter = iv.insert(iter, 2 * some_val); // iter 指向新元素
            new_ele++;
            iter++; iter++;                  // 将 iter 推进到旧元素的下一个位置
        } else iter++;                      // 简单推进 iter

    // 用 begin() 获取 vector 首元素迭代器，遍历 vector 中的所有元素
    for (iter = iv.begin(); iter != iv.end(); iter++)
        cout << *iter << endl;

    return 0;
}
```

### 【其他解题思路】

由于程序的意图是检查 `iv` 原来的前一半元素，也就是说，循环次数是预先可知的。因此，我们可以通过检测循环变量来控制循环执行次数，这要比比较“当前”迭代器和“中央迭代器”的方式简单一些：

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector<int> iv = {1, 1, 1, 1, 1};      // int 的 vector
    int some_val = 1;
```

```

vector<int>::iterator iter = iv.begin();
int org_size = iv.size(), i = 0;           // 原大小

// 用循环变量控制循环次数
while (i <= org_size / 2) {
    if (*iter == some_val) {
        iter = iv.insert(iter, 2 * some_val); // iter 指向新元素
        iter++; iter++;                  // 将 iter 推进到旧元素的下一个位置
    } else iter++;
    i++;
}

// 用 begin() 获取 vector 首元素迭代器，遍历 vector 中的所有元素
for (iter = iv.begin(); iter != iv.end(); iter++)
    cout << *iter << endl;

return 0;
}

```

**练习 9.23：**在本节第一个程序（第 309 页）中，若 `c.size()` 为 1，则 `val`、`val2`、`val3` 和 `val4` 的值会是什么？

**【出题思路】**

理解获取容器首、尾元素的不同方法。

**【解答】**

4 个变量的值会一样，都等于容器中唯一一个元素的值。

**练习 9.24：**编写程序，分别使用 `at`、下标运算符、`front` 和 `begin` 提取一个 `vector` 中的第一个元素。在一个空 `vector` 上测试你的程序。

**【出题思路】**

练习获取容器首元素的不同方法，以及如何安全访问容器元素。

**【解答】**

下面的程序会异常终止。因为 `vector` 为空，此时用 `at` 访问容器的第一个元素会抛出一个 `out_of_range` 异常，而此程序未捕获异常，因此程序会因异常退出。正确的编程方式是，捕获可能的 `out_of_range` 异常，进行相应的处理。

但对于后三种获取容器首元素的方法，当容器为空时，不会抛出 `out_of_range` 异常，而是导致程序直接退出（注释掉前几条语句即可看到后面语句的执行效果）。因此，正确的编程方式是，在采用这几种获取容器的方法时，检查下标的合法性（对 `front` 和 `begin` 只需检查容器是否为空），确定没有问题后再获取元素。当然这种方法对 `at` 也适用。

```
#include <iostream>
#include <vector>
```

```

using namespace std;

int main()
{
    vector<int> iv;      // int 的 vector

    cout << iv.at(0) << endl;
    cout << iv[0] << endl;
    cout << iv.front() << endl;
    cout << *(iv.begin()) << endl;

    return 0;
}

```

**练习 9.25:** 对于第 312 页中删除一个范围内的元素的程序, 如果 elem1 与 elem2 相等会发生什么? 如果 elem2 是尾后迭代器, 或者 elem1 和 elem2 皆为尾后迭代器, 又会发生什么?

#### 【出题思路】

理解范围删除操作的两个迭代器参数如何决定删除操作的结果。

#### 【解答】

如果两个迭代器 elem1 和 elem2 相等, 则什么也不会发生, 容器保持不变。哪怕两个迭代器是指向尾后位置 (例如 end() +1), 也是如此, 程序也不会出错。

因此 elem1 和 elem2 都是尾后迭代器时, 容器保持不变。

如果 elem2 为尾后迭代器, elem1 指向之前的合法位置, 则会删除从 elem1 开始直至容器末尾的所有元素。

**练习 9.26:** 使用下面代码定义的 ia, 将 ia 拷贝到一个 vector 和一个 list 中。使用单迭代器版本的 erase 从 list 中删除奇数元素, 从 vector 中删除偶数元素。

```
int ia[] = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 55, 89 };
```

#### 【出题思路】

练习删除指定位置元素的操作, 理解操作对迭代器的影响。

#### 【解答】

当从 vector 中删除元素时, 会导致删除点之后位置的迭代器、引用和指针失效。而 erase 返回的迭代器指向删除元素之后的位置。因此, 将 erase 返回的迭代器赋予 iiv, 使其正确向前推进。且尾后位置每个循环步中都用 end 重新获得, 保证其有效。

对于 list, 删除操作并不会令迭代器失效, 但上述方法仍然是适用的。

```
#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> iv;
    list<int> il;

    iv.assign(ia, ia + 11);           // 将数据拷贝到 vector
    il.assign(ia, ia + 11);           // 将数据拷贝到 list

    vector<int>::iterator iiv = iv.begin();
    while (iiv != iv.end())
        if (!(*iiv & 1))           // 偶数
            iiv = iv.erase(iiv);     // 删除偶数, 返回下一位置迭代器
        else iiv++;                // 推进到下一位置

    list<int>::iterator iil = il.begin();
    while (iil != il.end())
        if (*iil & 1)             // 奇数
            iil = il.erase(iil);     // 删除奇数, 返回下一位置迭代器
        else iil++;                // 推进到下一位置

    for (iiv = iv.begin(); iiv != iv.end(); iiv++)
        cout << *iiv << " ";
    cout << endl;
    for (iil = il.begin(); iil != il.end(); iil++)
        cout << *iil << " ";
    cout << endl;

    return 0;
}

```

**练习 9.27:** 编写程序, 查找并删除 `forward_list<int>` 中的奇数元素。

#### 【出题思路】

练习 `forward_list` 特殊的删除操作。

#### 【解答】

关键点是理解 `forward_list` 其实是单向链表数据结构, 只有前驱节点指向后继节点的指针, 而没有反向的指针。因此, 在 `forward_list` 中可以高效地从前驱转到后继, 但无法从后继转到前驱。而当我们删除一个元素后, 应该调整被删元素的前驱指针指向被删元素的后继, 起到将该元素从链表中删除的效果。因此, 在 `forward_list` 中插入、删除元素既需要该元素的迭代器, 也需要前驱迭代器。为此, `forward_list` 提供了 `before_begin` 来获取首元素之前位置的迭代器, 且插入、删除都是 `_after` 形式, 即, 删除 (插入) 给定迭代器的后继。

```

#include <iostream>
#include <forward_list>

```

```

using namespace std;

int main()
{
    forward_list<int> iflst = {1, 2, 3, 4, 5, 6, 7, 8};

    auto prev = iflst.before_begin();           // 前驱元素
    auto curr = iflst.begin();                 // 当前元素

    while (curr != iflst.end())
        if (*curr & 1)                      // 奇数
            curr = iflst.erase_after(prev);    // 删除，移动到下一元素
        else {
            prev = curr;                   // 前驱和当前迭代器都向前推进
            curr++;
        }
    for (curr = iflst.begin(); curr != iflst.end(); curr++)
        cout << *curr << " ";
    cout << endl;

    return 0;
}

```

**练习 9.28：**编写函数，接受一个 `forward_list<string>` 和两个 `string` 共三个参数。函数应在链表中查找第一个 `string`，并将第二个 `string` 插入到紧接着第一个 `string` 之后的位置。若第一个 `string` 未在链表中，则将第二个 `string` 插入到链表末尾。

#### 【出题思路】

练习 `forward_list` 特殊的添加操作。

#### 【解答】

与删除相同的是，`forward_list` 的插入操作也是在给定元素之后。不同的是，插入一个新元素后，只需将其后继修改为给定元素的后继，然后修改给定元素的后继为新元素即可，不需要前驱迭代器参与。但对于本题，当第一个 `string` 不在链表中时，要将第二个 `string` 插入到链表末尾。因此仍然需要维护前驱迭代器，当遍历完链表时，“前驱”指向尾元素，“当前”指向尾后位置。若第一个 `string` 不在链表中，此时只需将第二个 `string` 插入到“前驱”之后即可。

总体来说，单向链表由于其数据结构上的局限，为实现正确插入、删除操作带来了困难。标准库的 `forward_list` 容器为我们提供了一些特性，虽然（与其他容器相比）我们仍需维护一些额外的迭代器，但已经比直接用指针来实现链表的插入、删除方便了许多。

```

#include <iostream>
#include <forward_list>

using namespace std;

```

```

void test_and_insert(forward_list<string> &sflst, const string &s1,
const string &s2)
{
    auto prev = sflst.before_begin();           // 前驱元素
    auto curr = sflst.begin();                  // 当前元素
    bool inserted = false;

    while (curr != sflst.end()) {
        if (*curr == s1) {                      // 找到给定字符串
            curr = sflst.insert_after(curr, s2); // 插入新字符串, curr 指向它
            inserted = true;
        }
        prev = curr;                          // 前驱迭代器向前推进
        curr++;                             // 当前迭代器向前推进
    }

    if (!inserted)
        sflst.insert_after(prev, s2);          // 未找到给定字符串, 插入尾后
}

int main()
{
    forward_list<string> sflst = {"Hello", "!", "world", "!"};

    test_and_insert(sflst, "Hello", "你好");
    for (auto curr = sflst.cbegin(); curr != sflst.cend(); curr++)
        cout << *curr << " ";
    cout << endl;

    test_and_insert(sflst, "!", "?");
    for (auto curr = sflst.cbegin(); curr != sflst.cend(); curr++)
        cout << *curr << " ";
    cout << endl;

    test_and_insert(sflst, "Bye", "再见");
    for (auto curr = sflst.cbegin(); curr != sflst.cend(); curr++)
        cout << *curr << " ";
    cout << endl;

    return 0;
}

```

**练习 9.29:** 假定 vec 包含 25 个元素, 那么 `vec.resize(100)` 会做什么? 如果接下来调用 `vec.resize(10)` 会做什么?

#### 【出题思路】

本题练习改变容器大小的操作。

#### 【解答】

调用 `vec.resize(100)` 会向 vec 末尾添加 75 个元素, 这些元素将进行值初始化。

接下来调用 `vec.resize(10)` 会将 `vec` 末尾的 90 个元素删除。

**练习 9.30:** 接受单个参数的 `resize` 版本对元素类型有什么限制（如果有的话）？

**【出题思路】**

更深入理解改变容器大小的操作。

**【解答】**

对于元素是类类型，则单参数 `resize` 版本要求该类型必须提供一个默认构造函数。

**练习 9.31:** 第 316 页中删除偶数值元素并复制奇数值元素的程序不能用于 `list` 或 `forward_list`。为什么？修改程序，使之也能用于这些类型。

**【出题思路】**

本题继续练习 `list` 和 `forward_list` 的插入、删除操作，理解与其他容器的不同，理解对迭代器的影响。

**【解答】**

`list` 和 `forward_list` 与其他容器的一个不同是，迭代器不支持加减运算，究其原因，链表中元素并非在内存中连续存储，因此无法通过地址的加减在元素间远距离移动。因此，应多次调用`++`来实现与迭代器加法相同的效果。

```
#include <iostream>
#include <list>

using namespace std;

int main()
{
    list<int> ilst = {0,1,2,3,4,5,6,7,8,9};
    auto curr = ilst.begin(); // 首节点

    while (curr != ilst.end()) {
        if (*curr & 1) { // 奇数
            curr = ilst.insert(curr, *curr); // 插入到当前元素之前
            curr++; curr++; // 移动到下一元素
        } else { // 偶数
            curr = ilst.erase(curr); // 删除，指向下一元素
        }
    }

    for (curr = ilst.begin(); curr != ilst.end(); curr++)
        cout << *curr << " ";
    cout << endl;

    return 0;
}
```

对于 `forward_list`，由于是单向链表结构，删除元素时，需将前驱指针调整

为指向下一个节点，因此需维护“前驱”、“后继”两个迭代器。

```
#include <iostream>
#include <forward_list>

using namespace std;

int main()
{
    forward_list<int> iflst = {0,1,2,3,4,5,6,7,8,9};
    auto prev = iflst.before_begin();           // 前驱节点
    auto curr = iflst.begin();                  // 首节点

    while (curr != iflst.end()) {
        if (*curr & 1) {                      // 奇数
            curr = iflst.insert_after(curr, *curr); // 插入到当前元素之后
            prev = curr;                      // prev 移动到新插入元素
            curr++;                          // curr 移动到下一元素
        } else {                            // 偶数
            curr = iflst.erase_after(prev);   // 删除, curr 指向下一元素
        }
    }

    for (curr = iflst.begin(); curr != iflst.end(); curr++)
        cout << *curr << " ";
    cout << endl;
}

return 0;
}
```

**练习 9.32：**在第 316 页的程序中，像下面语句这样调用 `insert` 是否合法？如果不合法，为什么？

```
iter = vi.insert(iter, *iter++);
```

#### 【出题思路】

本题复习实参与形参的关系，进一步熟悉迭代器的处理对容器操作的关键作用。

#### 【解答】

很多编译器（例如作者所使用的 tdm-gcc）对实参数值、向形参传递的处理顺序是由右至左的。这意味着，编译器在编译上述代码时，首先对`*iter++`求值，传递给 `insert` 第二个形参，此时 `iter` 已指向当前奇数的下一个元素，因此传递给 `insert` 的第一个参数的迭代器指向的是错误位置，程序执行会发生混乱，最终崩溃。

因此，若将代码改为 `iter = vi.insert(iter++, *iter);`，或是使用由左至右求值、传递参数的编译器，代码的运行结果是正确的。当然，这样的代码在逻辑上是毫无道理的。

**练习 9.33：**在本节最后一个例子中，如果不将 `insert` 的结果赋予 `begin`，将会发生什么？编写程序，去掉此赋值语句，验证你的答案。

**【出题思路】**

进一步理解容器插入、删除操作会使迭代器失效。

**【解答】**

向vector中插入新元素后，原有迭代器都会失效。因此，不将insert()返回的迭代器赋予begin，会使begin失效。继续使用begin会导致程序崩溃。对此程序，保存尾后迭代器和不向begin赋值两个错误存在其一，程序都会崩溃。

**练习 9.34：**假定vi是一个保存int的容器，其中有偶数值也有奇数值，分析下面循环的行为，然后编写程序验证你的分析是否正确。

```
iter = vi.begin();
while (iter != vi.end())
    if (*iter % 2)
        iter = vi.insert(iter, *iter);
    ++iter;
```

**【出题思路】**

继续熟悉容器插入、删除操作与迭代器的关系，以及编程中容易出现的错误。

**【解答】**

此段代码的第一个错误是忘记使用花括号，使得`++iter`变成循环后的第一条语句，而非所期望的循环体的最后一条语句。因此，除非容器为空，否则程序会陷入死循环：

1. 若容器的第一个元素是偶数，布尔表达式为假，if语句真分支不会被执行，`iter`保持不变。循环继续执行，真分支仍然不会执行，`iter`继续保持不变，如此陷入死循环。

2. 若容器的第一个元素是奇数，`insert`语句被调用，将该值插入到首元素之前，并将返回的迭代器（指向新插入元素）赋予`iter`，因此`iter`指向新首元素。继续执行循环，会继续将首元素复制到容器首位置，并令`iter`指向它，如此陷入死循环。

示例（粗体代表迭代器位置）：

```
初始: {1,2,3,4,5,6,7,8,9}
第一步: {1,1,2,3,4,5,6,7,8,9}
第二步: {1,1,1,2,3,4,5,6,7,8,9}
.....
```

下面的程序可展示程序执行效果。其中，我们在循环体最后加入了一个循环，打印容器中所有元素，即可观察程序执行效果。这是一种简单的程序调试方法。`cout >> tmp`是为了让程序暂停，程序员有时间观察输出，需要继续执行程序时，随意输入一个字符串即可。

```
#include <iostream>
#include <vector>

using namespace std;
```

```

int main()
{
    vector<int> vi = {1,2,3,4,5,6,7,8,9};
    auto iter = vi.begin();
    string tmp;
    while (iter != vi.end()) {
        if (*iter % 2)
            iter = vi.insert(iter, *iter);
        for (auto begin = vi.begin(); begin != vi.end(); begin++)
            cout << *begin << " ";
        cout << endl;
        cin >> tmp;
    }
    ++iter;

    return 0;
}

```

当我们将`++iter`放入循环体后，程序仍然是错误的，除非容器为空或仅包含偶数，否则程序仍然会陷入死循环。原因是，当遍历到奇数时，执行`insert`将该值插入到旧元素之前，将返回指向新元素的迭代器赋予`iter`，再递增`iter`，此时`iter`将指向旧元素。继续执行循环仍会重复这几个步骤，程序陷入死循环。正确的程序应该是将`++iter`移入循环体，再增加一个`++iter`，令`iter`指向奇数之后的元素。

**示例**（粗体代表迭代器位置）：

```

初始: {0,1,2,3,4,5,6,7,8,9}
第一步: {0,1,1,2,3,4,5,6,7,8,9}
第二步: {0,1,1,1,2,3,4,5,6,7,8,9}
.....

```

```

#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector<int> vi = {1,2,3,4,5,6,7,8,9};
    auto iter = vi.begin();
    string tmp;
    while (iter != vi.end()) {
        if (*iter % 2)
            iter = vi.insert(iter, *iter);
        ++iter;
        for (auto begin = vi.begin(); begin != vi.end(); begin++)
            cout << *begin << " ";
        cout << endl;
        cin >> tmp;
    }

    return 0;
}

```

**练习 9.35:** 解释一个 `vector` 的 `capacity` 和 `size` 有何区别。

**【解答】**

`capacity` 返回已经为 `vector` 分配了多大内存空间（单位是元素大小），也就是在不分配新空间的情况下，容器可以保存多少个元素。而 `size` 则返回容器当前已经保存了多少个元素。

**练习 9.36:** 一个容器的 `capacity` 可能小于它的 `size` 吗？

**【解答】**

由上一题解答可知，这是不可能的。

**练习 9.37:** 为什么 `list` 或 `array` 没有 `capacity` 成员函数？

**【出题思路】**

理解 `list` 和 `array` 与 `vector` 在数据结构上的差异导致内存分配方式的不同。

**【解答】**

`list` 是链表，当有新元素加入时，会从内存空间中分配一个新节点保存它；当从链表中删除元素时，该节点占用的内存空间会被立刻释放。因此，一个链表占用的内存空间总是与它当前保存的元素所需空间相等（换句话说，`capacity` 总是等于 `size`）。

而 `array` 是固定大小数组，内存一次性分配，大小不变，不会变化。

因此它们均不需要 `capacity`。

**练习 9.38:** 编写程序，探究在你的标准库实现中，`vector` 是如何增长的。

**【解答】**

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector<int> vi;
    int s, c;

    for (s = vi.size(), c = vi.capacity(); s <= c; s++)
        vi.push_back(1);
    cout << "空间: " << vi.capacity() << "元素数: " << vi.size() << endl;

    for (s = vi.size(), c = vi.capacity(); s <= c; s++)
        vi.push_back(1);
    cout << "空间: " << vi.capacity() << "元素数: " << vi.size() << endl;
```

```

for (s = vi.size(), c = vi.capacity(); s <= c; s++)
    vi.push_back(1);
cout << "空间: " << vi.capacity() << "元素数: " << vi.size() << endl;

for (s = vi.size(), c = vi.capacity(); s <= c; s++)
    vi.push_back(1);
cout << "空间: " << vi.capacity() << "元素数: " << vi.size() << endl;

for (s = vi.size(), c = vi.capacity(); s <= c; s++)
    vi.push_back(1);
cout << "空间: " << vi.capacity() << "元素数: " << vi.size() << endl;

return 0;
}

```

用作者的编译器编译执行这个程序，输出结果是：

```

空间: 1 元素数: 1
空间: 2 元素数: 2
空间: 4 元素数: 3
空间: 8 元素数: 5
空间: 16 元素数: 9

```

说明作者的编译器是成倍增长 vector 空间的。

**练习 9.39：**解释下面程序片段做了什么：

```

vector<string> svec;
svec.reserve(1024);
string word;
while (cin >> word)
    svec.push_back(word);
svec.resize(svec.size() + svec.size() / 2);

```

**【出题思路】**

继续熟悉 vector 空间分配。

**【解答】**

首先，reserve 为 svec 分配了 1024 个元素（字符串）的空间。

随后，循环会不断读入字符串，添加到 svec 末尾，直至遇到文件结束符。这个过程中，如果读入的字符串数量不多于 1024，则 svec 的容量 (capacity) 保持不变，不会分配新的内存空间。否则，会按一定规则分配更大的内存空间，并进行字符串的移动。

接下来，resize 将向 svec 末尾添加当前字符串数量一半那么多的新字符串，它们的值都是空串。若空间不够，会分配足够容纳这些新字符串的内存空间。

**练习 9.40：**如果上一题中的程序读入了 256 个词，在 resize 之后容器的 capacity 可能是多少？如果读入了 512 个、1000 个或 1048 个词呢？

**【解答】**

根据上题解答中对程序的分析：

若读入了 256 个词，则 `resize` 之后容器的 `capacity` 将是 384（假定使用 tdm-gcc 4.8.1）。

若读入了 512 个词，则 `resize` 之后容器的 `capacity` 将是 768。

若读入了 1000 个词，则 `resize` 之后容器的 `capacity` 将是 2048。

若读入了 1048 个词，则 `resize` 之后容器的 `capacity` 将是 2048。

**练习 9.41：**编写程序，从一个 `vector<char>` 初始化一个 `string`。

#### 【出题思路】

本题练习从字符数组初始化 `string`。

#### 【解答】

`vector` 提供了 `data` 成员函数，返回其内存空间的首地址。将此返回值作为 `string` 的构造函数的第一个参数，将 `vector` 的 `size` 返回值作为第二个参数，即可获取 `vector<char>` 中的数据，将其看作一个字符数组来初始化 `string`。

```
#include <iostream>
#include <vector>
#include <string>

using namespace std;

int main()
{
    vector<char> vc = {'H', 'e', 'l', 'l', 'o'};
    string s(vc.data(), vc.size());
    cout << s << endl;

    return 0;
}
```

**练习 9.42：**假定你希望每次读取一个字符存入一个 `string` 中，而且知道最少需要读取 100 个字符，你应该如何提高程序的性能？

#### 【出题思路】

本题练习高效地处理动态增长的 `string`。

#### 【解答】

由于知道至少读取 100 个字符，因此可以用 `reserve` 先为 `string` 分配 100 个字符的空间，然后逐个读取字符，用 `push_back` 添加到 `string` 末尾。

```
#include <iostream>
#include <vector>
#include <string>

using namespace std;

void input_string(string &s)
```

```

{
    s.reserve(100);
    char c;
    while (cin >> c)
        s.push_back(c);
}
int main()
{
    string s;
    input_string(s);
    cout << s << endl;
}

return 0;
}

```

**练习 9.43:** 编写函数，接受三个 string 参数 s、oldVal 和 newVal。使用迭代器及 insert 和 erase 函数将 s 中所有 oldVal 替换为 newVal。测试你的程序，用它替换通用的简写形式，如，将 “tho” 替换为 “though”，将 “thru” 替换为 “through”。

### 【出题思路】

本题练习较为复杂的 string 操作。

### 【解答】

由于要求使用迭代器，因此使用如下算法：

1. 用迭代器 `iter` 遍历字符串 `s`。注意，对于 `s` 末尾少于 `oldVal` 长度的部分，已不可能与 `oldVal` 相等，因此无须检查。
  2. 对每个位置，用一个循环检查 `s` 中字符是否与 `oldVal` 中的字符都相等。
  3. 若循环是因为 `iter2 == oldVal.end` 而退出，表明 `s` 中 `iter` 开始的子串与 `oldVal` 相等。则调用 `erase` 将此子串删除，接着用一个循环将 `newVal` 复制到当前位置（tdm-gcc 4.8.1 中，返回迭代器的 `insert` 只支持单个字符插入）。由于 `insert` 将新字符插入到当前位置之前，并返回指向新字符的迭代器，因此，逆序插入 `newVal` 字符即可。最后将 `iter` 移动到新插入内容之后，继续遍历 `s`。
  4. 否则，`iter` 开始的子串与 `oldVal` 不等，递增 `iter`，继续遍历 `s`。

```
#include <iostream>
```

```
#include <vector>
#include <string>

using namespace std;

void replace_string(string &s, const string &oldVal, const string &newVal)
{
    auto l = newVal.size();
    if (!l) // 要查找的字符串为空
        return;
```

```

auto iter = s.begin();
while (iter <= s.end()-1) {           // 末尾少于 oldVal 长度的部分无须检查
    auto iter1 = iter;
    auto iter2 = oldVal.begin();
    // s 中 iter 开始的子串必须每个字符都与 oldVal 相同
    while (iter2 != oldVal.end() && *iter1 == *iter2) {
        iter1++;
        iter2++;
    }
    if (iter2 == oldVal.end()) {      // oldVal 耗尽——字符串相等
        iter = s.erase(iter, iter1); // 删除 s 中与 oldVal 相等部分
        if (newVal.size()) {         // 替换子串是否为空
            iter2 = newVal.end();   // 由后至前逐个插入 newVal 中的字符
            do {
                iter2--;
                iter = s.insert(iter, *iter2);
            } while (iter2 > newVal.begin());
        }
        iter += newVal.size();       // 迭代器移动到新插入内容之后
    } else iter++;
}
int main()
{
    string s="tho thru tho!";
    replace_string(s, "thru", "through");
    cout << s << endl;

    replace_string(s, "tho", "though");
    cout << s << endl;

    replace_string(s, "through", "");
    cout << s << endl;

    return 0;
}

```

**练习 9.44：**重写上一题的函数，这次使用一个下标和 replace。

### 【出题思路】

本题练习使用标准库提供的特性更简单地实现 string 操作。

### 【解答】

由于可以使用下标和 replace，因此可以更为简单地实现上一题的目标。通过 find 成员函数（只支持下标参数）即可找到 s 中与 oldVal 相同的子串，接着用 replace 即可将找到的子串替换为新内容。可以看到，使用下标而不是迭代器，通常可以更简单地实现字符串操作。

```

#include <iostream>
#include <vector>
#include <string>

```

```

using namespace std;

void replace_string(string &s, const string &oldVal, const string
&newVal)
{
    int p = 0;
    while ((p=s.find(oldVal, p)) != string::npos) { // 在 s 中查找 oldVal
        s.replace(p, oldVal.size(), newVal); // 将找到的子串替换为 newVal 的内容
        p += newVal.size(); // 下标调整到新插入内容之后
    }
}
int main()
{
    string s="tho thru tho!";
    replace_string(s, "thru", "through");
    cout << s << endl;

    replace_string(s, "tho", "though");
    cout << s << endl;

    replace_string(s, "through", "");
    cout << s << endl;

    return 0;
}

```

**【其他解题思路】**

我们当然可以像上一题的解答一样来查找 s 中与 oldVal 相同的子串，但显然，无论从编程效率还是目标代码效率考虑，这样做都是毫无意义的。除非你确定目标程序会有效率上的问题，否则应尽量使用标准库功能来编写你的程序，而非自己从头编写代码。

**练习 9.45：**编写函数，接受一个表示名字的 string 参数和两个分别表示前缀（如“Mr.”或“Ms.”）和后缀（如“Jr.”或“III”）的字符串。使用迭代器及 insert 和 append 函数将前缀和后缀添加到给定的名字中，将生成的新 string 返回。

**【出题思路】**

本题练习 string 的追加操作。

**【解答】**

通过 insert 插入到首位置之前，即可实现前缀插入。通过 append 即可实现将后缀追加到字符串末尾。

```

#include <iostream>
#include <vector>
#include <string>

using namespace std;

void name_string(string &name, const string &prefix, const string
&suffix)

```

```

{
    name.insert(name.begin(), 1, ' ');
    name.insert(name.begin(), prefix.begin(), prefix.end()); // 插入前缀
    name.append(" ");
    name.append(suffix.begin(), suffix.end()); // 插入后缀
}

int main()
{
    string s="James Bond";
    name_string(s, "Mr.", "II");
    cout << s << endl;

    s = "M";
    name_string(s, "Mrs.", "III");
    cout << s << endl;

    return 0;
}

```

**练习 9.46:** 重写上一题的函数，这次使用位置和长度来管理 `string`，并只使用 `insert`。

### 【出题思路】

本题继续练习基于位置的 `string` 操作。

### 【解答】

使用 `insert, 0` 等价于 `begin()`，都是在当前首字符之前插入新字符串；`size()` 等价于 `end()`，都是在末尾追加新字符串。

```

#include <iostream>
#include <vector>
#include <string>

using namespace std;

void name_string(string &name, const string &prefix, const string &suffix)
{
    name.insert(0, " ");
    name.insert(0, prefix); // 插入前缀
    name.insert(name.size(), " ");
    name.insert(name.size(), suffix); // 插入后缀
}

int main()
{
    string s="James Bond";
    name_string(s, "Mr.", "II");
    cout << s << endl;

    s = "M";
    name_string(s, "Mrs.", "III");

```

```

    cout << s << endl;

    return 0;
}

```

**练习 9.47：**编写程序，首先查找 string "ab2c3d7R4E6" 中的每个数字字符，然后查找其中每个字母字符。编写两个版本的程序，第一个要使用 `find_first_of`，第二个要使用 `find_first_not_of`。

### 【出题思路】

本题练习 `string` 的搜索操作的基本用法。

### 【解答】

`find_first_of` 在字符串中查找给定字符集合中任一字符首次出现的位置。若查找数字字符，则“给定字符集合”应包含所有 10 个数字；若查找字母，则要包含所有大小写字母——`abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ`。

```

#include <iostream>
#include <string>

using namespace std;

void find_char(string &s, const string &chars)
{
    cout << "在" << s << "中查找" << chars << "中字符" << endl;
    string::size_type pos = 0;
    while ((pos = s.find_first_of(chars, pos)) !=
           string::npos) { // 找到字符
        cout << "pos: " << pos << ", char: " << s[pos] << endl;
        pos++; // 移动到下一字符
    }
}

int main()
{
    string s="ab2c3d7R4E6";
    cout << "查找所有数字" << endl;
    find_char(s, "0123456789");
    cout << endl << "查找所有字母" << endl;
    find_char(s, "abcdefghijklmnopqrstuvwxyz" \
              "ABCDEFGHIJKLMNOPQRSTUVWXYZ");

    return 0;
}

```

`find_first_not_of` 查找第一个不在给定字符集合中出现的字符，若用它查找某类字符首次出现的位置，则应使用补集。若查找数字字符，则“给定字符集合”应包含 10 个数字之外的所有字符——`abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ`；若查找字母，则要包含所有非字母字符。注意，这一设定仅对此

问题要查找的字符串有效——它只包含字母和数字。因此，字母和数字互为补集。若字符串包含任意 ASCII 字符，可以想见，正确的“补集”可能非常冗长。

```
#include <iostream>
#include <string>

using namespace std;

void find_not_char(string &s, const string &chars)
{
    cout << "在" << s << "中查找不在" << chars << "中字符" << endl;
    string::size_type pos = 0;
    while ((pos = s.find_first_not_of(chars, pos)) !=
           string::npos) { // 找到字符
        cout << "pos: " << pos << ", char: " << s[pos] << endl;
        pos++; // 移动到下一字符
    }
}

int main()
{
    string s="ab2c3d7R4E6";
    cout << "查找所有数字" << endl;
    find_not_char(s, "abcdefghijklmnopqrstuvwxyz" \
                  "ABCDEFGHIJKLMNOPQRSTUVWXYZ");
    cout << endl << "查找所有数字" << endl;
    find_not_char(s, "0123456789");

    return 0;
}
```

**练习 9.48:** 假定 name 和 numbers 的定义如 325 页所示, numbers.find(name) 返回什么?

#### 【出题思路】

理解 find 与 find\_first\_of、find\_first\_not\_of 的区别。

#### 【解答】

s.find(args) 查找 s 中 args 第一次出现的位置，即第一个与 args 匹配的字符串的位置。args 是作为一个字符串整体在 s 中查找，而非一个字符集合在 s 中查找其中字符。因此，对 325 页给定的 name 和 numbers 值，在 numbers 中不存在与 name 匹配的字符串，find 会返回 npos。

**练习 9.49:** 如果一个字母延伸到中线之上，如 d 或 f，则称其有上出头部分(ascender)。如果一个字母延伸到中线之下，如 p 或 g，则称其有下出头部分(descender)。编写程序，读入一个单词文件，输出最长的既不包含上出头部分，也不包含下出头部分的单词。

**【出题思路】**

本题练习用搜索操作做一些更复杂的事情。

**【解答】**

查找既不包含上出头字母，也不包含下出头字母的单词，等价于“排除包含上出头字母或下出头字母的单词”。因此，用 `find_first_of` 在单词中查找上出头字母或下出头字母是否出现。若出现（返回一个合法位置，而非 `npos`），则丢弃此单词，继续检查下一个单词。否则，表明单词符合要求，检查它是否比之前的最长合法单词更长，若是，记录其长度和内容。文件读取完毕后，输出最长的合乎要求的单词。

```
#include <iostream>
#include <fstream>
#include <string>

using namespace std;

void find_longest_word(ifstream &in)
{
    string s, longest_word;
    int max_length = 0;

    while (in >> s) {
        if (s.find_first_of("bdfghjklpqty") != string::npos)
            continue; // 包含上出头或下出头字母
        cout << s << " ";
        if (max_length < s.size()) { // 新单词更长
            max_length = s.size(); // 记录长度和单词
            longest_word = s;
        }
    }
    cout << endl << "最长字符串：" << longest_word << endl;
}

int main(int argc, char *argv[])
{
    ifstream in(argv[1]); // 打开文件
    if (!in) {
        cerr << "无法打开输入文件" << endl;
        return -1;
    }

    find_longest_word(in);

    return 0;
}
```

**练习 9.50：**编写程序处理一个 `vector<string>`，其元素都表示整型值。计算 `vector` 中所有元素之和。修改程序，使之计算表示浮点值的 `string` 之和。

**【出题思路】**

本题练习简单的字符串到数值的类型转换，这在开发实际应用程序时是非常常见的操作，是很有用的基本编程技巧。

### 【解答】

标准库提供了将字符串转换为整型函数 `stoi`。如果希望转换为不同整型类型，如长整型、无符号整型等，标准库也都提供了相应的版本。

```
#include <iostream>
#include <string>
#include <vector>

using namespace std;

int main()
{
    vector<string> vs = {"123", "+456", "-789"};
    int sum = 0;

    for (auto iter = vs.begin(); iter != vs.end(); iter++)
        sum += stoi(*iter);

    cout << "和: " << sum << endl;
    return 0;
}
```

标准库也提供了将字符串转换为浮点数的函数，其中 `stof` 是转换为单精度浮点数。简单修改上面的程序即可实现本题的第二问。

注意，当给定的字符串不能转换为数值时（不是所需类型数值的合法表示），这些转换函数会抛出 `invalid_argument` 异常；如果表示的值超出类型所能表达的范围，则抛出一个 `out_of_range` 异常。这两个程序均未捕获、处理这两个异常，读者可尝试编写捕获并处理异常的版本，并用不合要求的字符串进行测试。

```
#include <iostream>
#include <string>
#include <vector>

using namespace std;

int main()
{
    vector<string> vs = {"12.3", "-4.56", "+7.8e-2"};
    float sum = 0;

    for (auto iter = vs.begin(); iter != vs.end(); iter++)
        sum += stof(*iter);

    cout << "和: " << sum << endl;
    return 0;
}
```

**练习 9.51:** 设计一个类，它有三个 `unsigned` 成员，分别表示年、月和日。为其编写构造函数，接受一个表示日期的 `string` 参数。你的构造函数应该能处理不同数据格式，如 January 1,1900、1/1/1900、Jan 1 1900 等。

### 【出题思路】

本题看似简单，但实际上较为复杂。在实际应用程序开发中，编写从文本中提取格式数据的程序片段，是非常烦琐、很容易出错的工作。因为这部分程序不能只会解析格式正确的数据，还应检查格式错误，给出错误信息。

### 【解答】

在头文件中定义了 `date` 类。构造函数 `date(string &ds)` 从字符串中解析出年、月、日的值，大致步骤如下：

1. 若首字符是数字，则为格式 2，用 `stoi` 提取月份值，若月份值不合法，抛出异常，否则转到步骤 6。
2. 若首字符不是数字，表明是格式 1 或 3，首先提取月份值。
3. 将 `ds` 开始的子串与月份简称进行比较，若均不等，抛出异常（若与简称不等，则不可能与全称相等）。
4. 若与第 `i` 个月简称相等，且下一个字符是合法间隔符，返回月份值。
5. 否则，检查接下来的子串是否与全称剩余部分相等，若不等，抛出异常；否则，返回月份值。
6. 用 `stoi` 提取日期值和年份值，如需要，检查间隔符合法性。

读者需要特别注意的是，在解析过程中，如何调整偏移量 `p`。

此程序已经较为复杂，但显然离“完美”还差很远，只能解析 3 种格式，且进行了很多简化。程序中已经给出了几种格式错误，读者可尝试构造其他可能的格式错误。并尝试补充程序，支持其他格式，如“2006 年 7 月 12 日”。

此外，程序中也涉及类、异常等相关的编程知识，读者可自行分析。

头文件 `date.h` 如下所示：

```
#ifndef DATE_H_INCLUDED
#define DATE_H_INCLUDED

#include <iostream>
#include <string>
#include <stdexcept>

using namespace std;

class date {
public:
    friend ostream& operator<<(ostream&, const date&);

    date() = default;
    date(string &ds);

    unsigned y() const { return year; }
    unsigned m() const { return month; }
}
```

```

unsigned d() const { return day; }

private:
    unsigned year, month, day;
};

// 月份全称
const string month_name[] = {"January", "February", "March",
    "April", "May", "June", "July", "August", "September",
    "October", "November", "December"};

// 月份简写
const string month_abbr[] = {"Jan", "Feb", "Mar", "Apr", "May",
    "Jun", "Jul", "Aug", "Sept", "Oct", "Nov", "Dec"};

// 每月天数
const int days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

inline int get_month(string &ds, int &end_of_month)
{
    int i, j;
    for (i = 0; i < 12; i++) {
        // 检查每个字符是否与月份简写相等
        for (j = 0; j < month_abbr[i].size(); j++)
            if (ds[j] != month_abbr[i][j])      // 不是此月简写
                break;
        if (j == month_abbr[i].size())      // 与简写匹配
            break;
    }

    if (i == 12)                      // 与所有月份名都不相同
        throw invalid_argument("不是合法月份名");

    if (ds[j] == ' ') {               // 空白符, 仅是月份简写
        end_of_month = j + 1;
        return i + 1;
    }

    for (; j < month_name[i].size(); j++)
        if (ds[j] != month_name[i][j])
            break;

    if (j == month_name[i].size() && ds[j] == ' ') { // 月份全称
        end_of_month = j + 1;
        return i + 1;
    }

    throw invalid_argument("不是合法月份名");
}

inline int get_day(string &ds, int month, int &p)
{
    size_t q;

```

```

int day = stoi(ds.substr(p), &q); // 从 p 开始的部分转换为日期值
if (day < 1 || day > days[month])
    throw invalid_argument("不是合法日期值");
    p += q; // 移动到日期值之后
return day;
}

inline int get_year(string &ds, int &p)
{
    size_t q;
    int year = stoi(ds.substr(p), &q); // 从 p 开始的部分转换为年
    if (p + q < ds.size())
        throw invalid_argument("非法结尾内容");
    return year;
}

date::date(string &ds)
{
    int p;
    size_t q;

    if ((p = ds.find_first_of("0123456789")) == string::npos)
        throw invalid_argument("没有数字，非法日期");

    if (p > 0) { // 月份名格式
        month = get_month(ds, p);
        day = get_day(ds, month, p);
        if (ds[p] != ' ' && ds[p] != ',')
            throw invalid_argument("非法间隔符");
        p++;
        year = get_year(ds, p);
    } else { // 月份值格式
        month = stoi(ds, &q);
        p = q;
        if (month < 1 || month > 12)
            throw invalid_argument("不是合法月份值");
        if (ds[p++] != '/')
            throw invalid_argument("非法间隔符");
        day = get_day(ds, month, p);
        if (ds[p++] != '/')
            throw invalid_argument("非法间隔符");
        year = get_year(ds, p);
    }
}

ostream & operator<<(ostream& out, const date& d)
{
    out << d.y() << "年" << d.m() << "月" << d.d() << "日" << endl;
    return out;
}
#endif // DATE_H_INCLUDED

```

主程序如下所示：

```
#include <iostream>
#include <string>
#include "date.h"

using namespace std;

int main()
{
    string dates[] = {"Jan 1,2014", "February 1 2014", "3/1/2014",
                      // "Jcn 1,2014",
                      // "January 1,2014",
                      // "Jan 32,2014",
                      // "Jan 1/2014",
                      "3 1 2014 ",
                      };
    try {
        for (auto ds : dates) {
            date d1(ds);
            cout << d1;
        }
    } catch (invalid_argument e) {
        cout << e.what() << endl;
    }

    return 0;
}
```

**练习 9.52：**使用 stack 处理括号化的表达式。当你看到一个左括号，将其记录下来。当你在一个左括号之后看到一个右括号，从 stack 中 pop 对象，直至遇到左括号，将左括号也一起弹出栈。然后将一个值（括号内的运算结果）push 到栈中，表示一个括号化的（子）表达式已经处理完毕，被其运算结果所替代。

### 【出题思路】

本题作为栈数据结构的基本练习有些复杂。原因在于，表达式的解析算法对于编程初学者来说相对复杂。解答本题更多的工作是设计表达式解析的逻辑，栈的操作变为次要地位。

### 【解答】

如下所示，本题的解答已经较为复杂，但这已是将问题大幅简化之后的解答了。我们假定表达式中只有加、减两种运算，由于所有运算优先级相同，无须进行复杂的优先级判断。由于加、减运算都是左结合，因此，遇到一个数值，直接与之前的运算数进行它们中间的运算即可。唯一的例外是括号，它将改变计算次序——括号里的运算将先于括号前的运算执行。不过，我们可以将它看作“阻断”了括号前的表达式，将括号内的部分看作一个独立的表达式优先处理，计算结果作为一个运算数参与之前的运算即可。

表达式解析的逻辑大致如下：

1. 读入了一个运算数 v。

a)若栈空或栈顶是左括号，则 v 是第一个运算数，直接压栈即可。

b)否则，v 前必须是一个运算符，再之前是另一个运算数 v'，从栈顶弹出这两项，将计算结果压栈即可；否则，就抛出一个“缺少运算符”异常。

2. 读入了一个左括号，直接压栈。

3. 读入了一个运算符，

a)若栈空或栈顶不是一个运算数，则抛出一个“缺少运算数”异常。注意，若运算符之前是一个右括号，之前也已处理完毕，栈顶是其计算结果，仍应该是运算数，不影响此逻辑。

b)否则，运算符压栈。

4. 读入了一个右括号，

a)若栈空，表明之前没有与之配对的左括号，抛出“未匹配右括号”异常。

b)若栈顶是左括号，表明括号对之间无表达式，抛出“空括号”异常。

c)若栈顶不是运算数，表明括号内缺少一个运算数，抛出一个异常。

d)弹出此运算数 v，若栈空或栈顶不是左括号，仍抛出“未匹配右括号”异常；否则弹出左括号，把 v 作为新运算数，执行 1 中的逻辑。

5. 以上均不是，则出现了非法输入，会在转换为数值时产生异常。

6. 当字符串处理完毕后，判断栈中是否有且仅有一个运算数，若是，此值即为表达式运算结果，输出它；否则，表达式非法。

值得注意的是，为了在栈中保存括号、运算符和运算数三类对象，程序中定义了枚举类型 obj\_type。栈中每个对象都保存了类型 t 和数值 v（如果 t 是 VAL 的话）。

```
#include <iostream>
#include <string>
#include <deque>
#include <stack>
#include <stdexcept>

using namespace std;

// 表示栈中对象的不同类型
enum obj_type { LP, RP, ADD, SUB, VAL };
struct obj {
    obj(obj_type type, double val = 0) { t = type; v = val; }
    obj_type t;
    double v;
};

inline void skipws(string &exp, size_t &p)
{
    p = exp.find_first_not_of(" ", p);
}
```

```

inline void new_val(stack<obj> &so, double v)
{
    if (so.empty() || so.top().t == LP) {           // 空栈或左括号
        so.push(obj(VAL, v));
        // cout << "push " << v << endl;
    } else if (so.top().t == ADD || so.top().t == SUB) {
        // 之前是运算符
        obj_type type = so.top().t;
        so.pop();
        /*if (type == ADD)
            cout << "pop +" << endl;
        else cout << "pop -" << endl;*/
        // cout << "pop " << so.top().v << endl;
        // 执行加减法
        if (type == ADD)
            v += so.top().v;
        else v = so.top().v - v;
        so.pop();
        so.push(obj(VAL, v));                         // 运算结果压栈
        // cout << "push " << v << endl;
    } else throw invalid_argument("缺少运算符");
}

int main()
{
    stack<obj> so;
    string exp;
    size_t p = 0, q;
    double v;

    cout << "请输入表达式: ";
    getline(cin, exp);

    while (p < exp.size()) {
        skipws(exp, p);                           // 跳过空格
        if (exp[p] == '(') {                      // 左括号直接压栈
            so.push(obj(LP));
            p++;
            // cout << "push LP" << endl;
        } else if (exp[p] == '+' || exp[p] == '-') {
            // 新运算符
            if (so.empty() || so.top().t != VAL)
                // 空栈或之前不是运算数
                throw invalid_argument("缺少运算数");

            if (exp[p] == '+')                     // 运算符压栈
                so.push(obj(ADD));
            else so.push(obj(SUB));
            p++;
            // cout << "push " << exp[p - 1] << endl;
        } else if (exp[p] == ')') {               // 右括号
            p++;
        }
    }
}

```

```

if (so.empty())                                // 之前无配对的左括号
    throw invalid_argument("未匹配右括号");

if (so.top().t == LP)                          // 一对括号之间无内容
    throw invalid_argument("空括号");

if (so.top().t == VAL) {                      // 正确：括号内运算结果
    v = so.top().v;
    so.pop();
    // cout << "pop " << v << endl;

    if (so.empty() || so.top().t != RP)
        throw invalid_argument("未匹配右括号");

    so.pop();
    // cout << "pop RP" << endl;
    new_val(so, v);                         // 与新运算数逻辑一致
} else   // 栈顶必定是运算符
    throw invalid_argument("缺少运算数");
} else {                                       // 应该是运算数
    v = stod(exp.substr(p), &q);
    p += q;
    new_val(so, v);
}
}

if (so.size() != 1 || so.top().t != VAL)
    throw invalid_argument("非法表达式");

cout << so.top().v << endl;

return 0;
}

```

### 【解答解题思路】

其实，如果不是强制使用栈来手工处理解析过程的数据，我们可以写出一个递归算法（其实是编译器隐含使用栈为我们保存了相关数据）来解析表达式，会更为简洁、清晰。

# 第 10 章

# 泛型算法

## 导读

本章介绍了标准库算法，包括：

- 基础的只读算法、写容器元素的算法。
- 利用 lambda、bind 等技术手段定制操作。
- 更深入的迭代器知识，如插入器、IO 流迭代器、反向迭代器。
- 泛型算法对迭代器的分类和算法命名规范，以及特定容器算法。

本章练习的最重要目的是让读者深入理解“泛型”思想，体会标准库是如何通过算法和数据结构的分离来实现泛型的，以及如何通过迭代器在分离的算法和数据结构间架起桥梁，达到算法“不知”数据结构，但又能操纵数据元素的效果。具体内容包括基础算法使用的练习、lambda 和定制操作的练习、插入器/IO 流迭代器/反向迭代器的练习等。

**练习 10.1:** 头文件 `algorithm` 中定义了一个名为 `count` 的函数，它类似 `find`，接受一对迭代器和一个值作为参数。`count` 返回给定值在序列中出现的次数。编写程序，读取 `int` 序列存入 `vector` 中，打印有多少个元素的值等于给定值。

### 【出题思路】

本题练习泛型算法的简单使用。

### 【解答】

泛型算法的使用其实很简单，记住关键一点：泛型算法不会直接调用容器的操作，而是通过迭代器来访问、修改、移动元素。对于本题，将 `vector` 的 `begin` 和 `end` 传递给 `count`，并将要查找的值作为第三个参数传递给它即可。

```
#include <iostream>
```

```

#include <fstream>
#include <vector>
#include <algorithm>

using namespace std;

int main(int argc, char *argv[])
{
    ifstream in(argv[1]);
    if (!in) {
        cout << "打开输入文件失败！" << endl;
        exit(1);
    }

    vector<int> vi;
    int val;
    while (in >> val)
        vi.push_back(val);

    cout << "请输入要搜索的整数：" ;
    cin >> val;

    cout << "序列中包含" << count(vi.begin(), vi.end(), val)
        << "个" << val;

    return 0;
}

```

### 【其他解题思路】

读者可以尝试自己编写计数整数出现次数的代码，并尝试回答如下问题：

1. 与使用 `count` 相比，编程效率如何？代码复杂度如何？若问题更为复杂呢？
2. 若需要计数给定的是单/双精度浮点数、字符串……在 `vector`、`deque`、`list`……中出现的次数，自己编写程序的方式会怎样，使用 `count` 的方式又如何？

通过回答这两个问题来理解泛型算法的特点。

```

#include <iostream>
#include <fstream>
#include <vector>
#include <algorithm>

using namespace std;

int main(int argc, char *argv[])
{
    ifstream in(argv[1]);
    if (!in) {
        cout << "打开输入文件失败！" << endl;
        exit(1);
    }

    vector<int> vi;
    int val;
    while (in >> val)

```

```

    vi.push_back(val);

    cout << "请输入要搜索的整数: ";
    cin >> val;

    int c = 0;
    for (auto iter = vi.begin(); iter != vi.end(); iter++)
        if (*iter == val)
            c++;
    cout << "序列中包含" << c << "个" << val;

    return 0;
}

```

**练习 10.2：**重做上一题，但读取 string 序列存入 list 中。

### 【出题思路】

理解“泛型”的优点。

### 【解答】

可以看到，与上一题对比，程序的变化只在不同类型变量的声明上，而算法的使用部分几乎没有任何差异。

```

#include <iostream>
#include <fstream>
#include <string>
#include <list>
#include <algorithm>

using namespace std;

int main(int argc, char *argv[])
{
    ifstream in(argv[1]);
    if (!in) {
        cout << "打开输入文件失败!" << endl;
        exit(1);
    }

    list<string> ls;
    string word;
    while (in >> word)
        ls.push_back(word);

    cout << "请输入要搜索的字符串: ";
    cin >> word;

    cout << "序列中包含" << count(ls.begin(), ls.end(), word)
        << "个" << word;

    return 0;
}

```

**练习 10.3:** 用 accumulate 求一个 vector<int> 中的元素之和。

**【出题思路】**

练习泛型算法的使用。

**【解答】**

Accumulate 的前两个参数仍然是指定范围的一对迭代器，第三个参数是和的初值。

```
#include <iostream>
#include <fstream>
#include <vector>
#include <algorithm>

using namespace std;

int main(int argc, char *argv[])
{
    ifstream in(argv[1]);
    if (!in) {
        cout << "打开输入文件失败!" << endl;
        exit(1);
    }

    vector<int> vi;
    int val;
    while (in >> val)
        vi.push_back(val);

    cout << "序列中整数之和为" << accumulate(vi.begin(), vi.end(), 0) << endl;

    return 0;
}
```

**练习 10.4:** 假定 v 是一个 vector<double>，那么调用 accumulate(v.cbegin(), v.cend(), 0) 有何错误（如果存在的话）？

**【出题思路】**

理解 accumulate 的第三个参数。

**【解答】**

accumulate 的第三个参数是和的初值，它还决定了函数的返回类型，以及函数中使用哪个加法运算符。

因此，本题中的调用是错误的，第三个参数 0 告知 accumulate，和是整型的，使用整型加法运算符。读者可尝试输入带小数的值，函数返回的是一个整数。

正确的调用方法是将 0.0 作为第三个参数传递给 accumulate。

读者可以修改、运行程序，观察运行结果。

```
#include <iostream>
#include <fstream>
#include <vector>
```

```

#include <algorithm>

using namespace std;

int main(int argc, char *argv[])
{
    ifstream in(argv[1]);
    if (!in) {
        cout << "打开输入文件失败！" << endl;
        exit(1);
    }

    vector<double> vd;
    double val;
    while (in >> val)
        vd.push_back(val);

    cout << "序列中浮点数之和为" << accumulate(vd.begin(), vd.end(), 0) << endl;

    return 0;
}

```

**练习 10.5：**在本节对名册 (roster) 调用 equal 的例子中，如果两个名册中保存的都是 C 风格字符串而不是 string，会发生什么？

### 【出题思路】

理解 equal 如何比较元素。

### 【解答】

equal 使用==运算符比较两个序列中的元素。string 类重载了==，可比较两个字符串是否长度相等且其中元素对位相等。而 C 风格字符串本质是 char\* 类型，用==比较两个 char\* 对象，只是检查两个指针值是否相等，即地址是否相等，而不会比较其中字符是否相同。所以，只有当两个序列中的指针都指向相同的地址时，equal 才会返回 true。否则，即使字符串内容完全相同，也会返回 false。

如下面的程序，q 中的每个字符串是 p 中字符串的拷贝，虽然内容相同，但是不同对象指向不同地址，因此 equal 判定它们不等。而 r 中每个指针都是 p 中指针的拷贝，指向相同的地址，因此 equal 判定它们相等。

```

#include <iostream>
#include <algorithm>
#include <string.h>

using namespace std;

int main(int argc, char *argv[])
{
    char *p[] = { "Hello", "World", "!" };
    char *q[] = { strdup(p[0]), strdup(p[1]), strdup(p[2]) };
    char *r[] = { p[0], p[1], p[2] };

    cout << "q == r: " << equal(q, q + 3, r, r + 3) << endl;
    cout << "p == r: " << equal(p, p + 3, r, r + 3) << endl;
}

```

```

cout << equal(begin(p), end(p), q) << endl;
cout << equal(begin(p), end(p), r) << endl;

return 0;
}

```

**练习 10.6：**编写程序，使用 `fill_n` 将一个序列中的 `int` 值都设置为 0。

**【出题思路】**

练习使用 `fill_n`。

**【解答】**

`fill_n` 接受一个迭代器，指出起始位置，还接受一个整数，指出设置的元素数目，第三个参数则是要设置的值。

```

#include <iostream>
#include <fstream>
#include <vector>
#include <algorithm>

using namespace std;

int main(int argc, char *argv[])
{
    ifstream in(argv[1]);
    if (!in) {
        cout << "打开输入文件失败！" << endl;
        exit(1);
    }

    vector<int> vi;
    int val;
    while (in >> val) {
        vi.push_back(val);
        cout << val << " ";
    }
    cout << endl;

    fill_n(vi.begin(), vi.size(), 0);
    for (auto iter = vi.begin(); iter != vi.end(); iter++)
        cout << *iter << " ";

    return 0;
}

```

**练习 10.7：**下面程序是否有错误？如果有，请改正。

```

(a) vector<int> vec; list<int> lst; int i;
    while (cin >> i)
        lst.push_back(i);
    copy(lst.cbegin(), lst.cend(), vec.begin());

```

```
(b) vector<int> vec;
    vec.reserve(10);      // reverse 将在 9.4 节(第 318 页)介绍
    fill_n(vec.begin(), 10, 0);
```

### 【出题思路】

进一步理解泛型算法的特点。

### 【解答】

(a) 是错误的。因为泛型算法的一个基本特点是：算法总是通过迭代器操作容器，因此不能直接向/从容器添加、删除元素，无法改变容器大小。因此，对于 copy 算法，要求目标序列至少要包含与源序列一样多的元素。而此程序中，vec 进行缺省初始化，它是空的，copy 无法进行。如需改变容器大小，需要使用一类特殊的称为插入器的迭代器。我们可以将第三个参数改为 back\_inserter(vec)，通过它，copy 算法即可将 lst 中元素的拷贝插入到 vec 的末尾。

```
#include <iostream>
#include <fstream>
#include <vector>
#include <list>
#include <algorithm>

using namespace std;

int main(int argc, char *argv[])
{
    ifstream in(argv[1]);
    if (!in) {
        cout << "打开输入文件失败！" << endl;
        exit(1);
    }

    list<int> lst;
    vector<int> vec;
    int val;
    while (in >> val)
        lst.push_back(val);

    copy(lst.begin(), lst.end(), back_inserter(vec));

    cout << equal(lst.begin(), lst.end(), vec.begin()) << endl;
    for (auto iter = vec.begin(); iter != vec.end(); iter++)
        cout << *iter << " ";
    cout << endl;

    return 0;
}
```

(b) 这段程序仍然是错误的。粗看起来，reserve 为 vec 分配了至少能容纳 10 个 int 的内存空间，调用 fill\_n 时，vec 已有足够空间。但泛型算法对于容器的要求并不是有足够的空间，而是足够的元素。此时 vec 仍然为空，没有任何元素。而算法又不具备向容器添加元素的能力，因此 fill\_n 仍然失败。这里，我们还是

需要使用 `back_inserter` 来让 `fill_n` 有能力向 `vec` 添加元素。其实，只有 0 有能力做到这一点，空间大小并不是问题，容器都能根据需要自动扩容。

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main(int argc, char *argv[])
{
    vector<int> vec;
    vec.reserve(10);
    fill_n(back_inserter(vec), 10, 0);

    for (auto iter = vec.begin(); iter != vec.end(); iter++)
        cout << *iter << " ";
    cout << endl;

    return 0;
}
```

**练习 10.8：**本节提到过，标准库算法不会改变它们所操作的容器的大小。为什么使用 `back_inserter` 不会使这一断言失效？

#### 【出题思路】

深入理解泛型算法的这一特点。

#### 【解答】

严格来说，标准库算法根本不知道有“容器”这个东西。它们只接受迭代器参数，运行于这些迭代器之上，通过这些迭代器来访问元素。

因此，当你传递给算法普通迭代器时，这些迭代器只能顺序或随机访问容器中的元素，造成的效果就是算法只能读取元素、改变元素值、移动元素，但无法添加或删除元素。

但当我们传递给算法插入器，例如 `back_inserter` 时，由于这类迭代器能调用下层容器的操作来向容器插入元素，造成的算法执行的效果就是向容器中添加了元素。

因此，关键要理解：标准库算法从来不直接操作容器，它们只操作迭代器，从而间接访问容器。能不能插入和删除元素，不在于算法，而在于传递给它们的迭代器是否具有这样的能力。

**练习 10.9：**实现你自己的 `elimDups`。测试你的程序，分别在读取输入后、调用 `unique` 后以及调用 `erase` 后打印 `vector` 的内容。

#### 【出题思路】

本题练习重排元素的算法。

**【解答】**

`unique` “消除” 重复值的方式并不是删除值重复的元素，执行 `unique` 后，容器的元素数目并未改变。不重复元素之后位置上的元素的值是未定义的。可以观察在你的编译器上这些元素是什么值，从而推测 `unique` 的实现方式。在作者的 tdm-gcc 8.1 上，这些元素的值是被“删除”的重复值 `red` 和 `the`，而且两者的顺序与原来是相反的。因此推测，`unique` 的实现方式并不是删除重复值，而是将它们交换到了容器的末尾。

```
#include <iostream>
#include <fstream>
#include <vector>
#include <string>
#include <algorithm>

using namespace std;

inline void output_words(vector<string> &words)
{
    for (auto iter = words.begin(); iter != words.end(); iter++)
        cout << *iter << " ";
    cout << endl;
}

void elimDups(vector<string> &words)
{
    output_words(words);

    sort(words.begin(), words.end());
    output_words(words);

    auto end_unique = unique(words.begin(), words.end());
    output_words(words);
}

int main(int argc, char *argv[])
{
    ifstream in(argv[1]);
    if (!in) {
        cout << "打开输入文件失败！" << endl;
        exit(1);
    }

    vector<string> words;
    string word;
    while (in >> word)
        words.push_back(word);

    elimDups(words);
}
```

```

    return 0;
}

```

**练习 10.10：**你认为算法不改变容器大小的原因是什么？

**【出题思路】**

让读者对语言的设计有自己的思考。

**【解答】**

泛型算法的一大优点是“泛型”，也就是一个算法可用于多种不同的数据类型，算法与所操作的数据结构分离。这对编程效率的提高是非常巨大的。

要做到算法与数据结构分离，重要的技术手段就是使用迭代器作为两者的桥梁。算法从不操作具体的容器，从而也就不存在与特定容器绑定，不适用于其他容器的问题。算法只操作迭代器，由迭代器真正实现对容器的访问。不同容器实现自己特定的迭代器（但不同迭代器是相容的），算法操作不同迭代器就实现了对不同容器的访问。

因此，并不是算法应该改变或不该改变容器的问题。为了实现与数据结构的分离，为了实现通用性，**算法根本就不该知道容器的存在**。算法访问数据的唯一通道是迭代器。是否改变容器大小，完全是迭代器的选择和责任。当我们向 `fill_n` 传递 `back_inserter` 时，虽然最终效果是向容器添加了新的元素，但对 `fill_n` 来说，根本不知道这回事儿。它仍然像往常一样（通过迭代器）向元素赋予新值，只不过这次是通过 `back_inserter` 来赋值，而 `back_inserter` 选择将新值添加到了容器而已。

**练习 10.11：**编写程序，使用 `stable_sort` 和 `isShorter` 将传递给你的 `elimDups` 版本的 `vector` 排序。打印 `vector` 的内容，验证你的程序的正确性。

**【出题思路】**

练习向算法传递谓词来定制操作，理解稳定排序的概念。

**【解答】**

按书中所述实现即可。

```

#include <iostream>
#include <fstream>
#include <vector>
#include <string>
#include <algorithm>

using namespace std;

inline void output_words(vector<string> &words)
{
    for (auto iter = words.begin(); iter != words.end(); iter++)
        cout << *iter << " ";
    cout << endl;
}

```

```

}

bool isShorter(const string &s1, const string &s2)
{
    return s1.size() < s2.size();
}

void elimDups(vector<string> &words)
{
    output_words(words);

    sort(words.begin(), words.end());
    output_words(words);

    auto end_unique = unique(words.begin(), words.end());
    output_words(words);

    words.erase(end_unique, words.end());
    output_words(words);

    stable_sort(words.begin(), words.end(), isShorter);
    output_words(words);
}

int main(int argc, char *argv[])
{
    ifstream in(argv[1]);
    if (!in) {
        cout << "打开输入文件失败！" << endl;
        exit(1);
    }

    vector<string> words;
    string word;
    while (in >> word)
        words.push_back(word);

    elimDups(words);

    return 0;
}

```

**练习 10.12：**编写名为 compareIsbn 的函数，比较两个 Sales\_data 对象的 isbn() 成员。使用这个函数排序一个保存 Sales\_data 对象的 vector。

### 【出题思路】

练习定义和使用谓词。

### 【解答】

我们将 compareIsbn 定义为一个二元谓词，接受两个 Sales\_data 对象，通过 isbn 成员函数获取 ISBN 编号，若前者小于后者返回真，否则返回假。

```
inline bool compareIsbn(const Sales_data &lhs, const Sales_data &rhs)
```

```

{
    return lhs.isbn() < rhs.isbn();
}

```

在主程序中，将 `compareIsbn` 作为第三个参数传递给 `sort`，即可实现销售数据按 ISBN 号排序(注，此程序的主程序 `12.cpp` 要与 `Sales_data.cpp` 一起编译，即，要将 `Sales_data.cpp` 添加到开发环境的项目中)。

```

#include <iostream>
#include <fstream>
#include <vector>
#include "Sales_data.h"
#include <algorithm>

using namespace std;

int main(int argc, char *argv[])
{
    ifstream in(argv[1]);
    if (!in) {
        cout << "打开输入文件失败！" << endl;
        exit(1);
    }

    vector<Sales_data> sds;
    Sales_data sd;
    while (read(in, sd))
        sds.push_back(sd);

    sort(sds.begin(), sds.end(), compareIsbn);

    for (const auto &s : sds) {
        print(cout, s);
        cout << endl;
    }

    return 0;
}

```

**练习 10.13：**标准库定义了名为 `partition` 的算法，它接受一个谓词，对容器内容进行划分，使得谓词为 `true` 的值会排在容器的前半部分，而使谓词为 `false` 的值会排在后半部分。算法返回一个迭代器，指向最后一个使谓词为 `true` 的元素之后的位置。编写函数，接受一个 `string`，返回一个 `bool` 值，指出 `string` 是否有 5 个或更多字符。使用此函数划分 `words`。打印出长度大于等于 5 的元素。

### 【出题思路】

练习定义和使用一元谓词。

### 【解答】

本题要求谓词判断一个 `string` 对象的长度是否大于等于 5，而不是比较两个 `string` 对象，因此它应是一个一元谓词。其他与上一题基本类似。但需要注意，我

我们应该保存 partition 返回的迭代器 iter，打印范围 [words.begin(), iter) 中的字符串。

```
#include <iostream>
#include <fstream>
#include <vector>
#include <string>
#include <algorithm>

using namespace std;

inline void output_words(vector<string>::iterator beg, vector <string>::iterator end)
{
    for (auto iter = beg; iter != end; iter++)
        cout << *iter << " ";
    cout << endl;
}

bool five_or_more(const string &s1)
{
    return s1.size() >= 5;
}

int main(int argc, char *argv[])
{
    ifstream in(argv[1]);
    if (!in) {
        cout << "打开输入文件失败！" << endl;
        exit(1);
    }

    vector<string> words;
    string word;
    while (in >> word)
        words.push_back(word);
    output_words(words.begin(), words.end());

    auto iter = partition(words.begin(), words.end(), five_or_more);
    output_words(words.begin(), iter);

    return 0;
}
```

**练习 10.14：**编写一个 lambda，接受两个 int，返回它们的和。

### 【出题思路】

练习定义和使用简单的 lambda。

### 【解答】

由于此 lambda 无须使用所在函数中定义的局部变量，所以捕获列表为空。参数列表为两个整型。返回类型由函数体唯一的语句——返回语句推断即可。

```
#include <iostream>
```

```

using namespace std;

int main(int argc, char *argv[])
{
    auto sum = [] (int a, int b) { return a + b; };

    cout << sum(1, 1) << endl;

    return 0;
}

```

**练习 10.15:** 编写一个 lambda，捕获它所在函数的 int，并接受一个 int 参数。lambda 应该返回捕获的 int 和 int 参数的和。

#### 【出题思路】

练习定义和使用简单的 lambda。

#### 【解答】

由于需要计算所在函数的局部 int 和自己的 int 参数的和，该 lambda 需捕获所在函数的局部 int。参数列表为一个整型列表。返回类型仍由返回语句推断。

```

#include <iostream>

using namespace std;

void add(int a)
{
    auto sum = [a] (int b) { return a + b; };

    cout << sum(1) << endl;
}

int main(int argc, char *argv[])
{
    add(1);
    add(2);

    return 0;
}

```

**练习 10.16:** 使用 lambda 编写你自己版本的 biggies。

#### 【出题思路】

继续练习 lambda。

#### 【解答】

按书中代码片段完成整个程序即可。注意，要用到第 6 章的 make\_plural。

```

#include <iostream>
#include <fstream>
#include <vector>
#include <string>

```

```

#include <algorithm>
#include "make_plural.h"

using namespace std;

inline void output_words(vector<string> &words)
{
    for (auto iter = words.begin(); iter != words.end(); iter++)
        cout << *iter << " ";
    cout << endl;
}

void elimDups(vector<string> &words)
{
    sort(words.begin(), words.end());
    auto end_unique = unique(words.begin(), words.end());
    words.erase(end_unique, words.end());
}

void biggies(vector<string> &words,
             vector<string>::size_type sz)
{
    elimDups(words);           // 将 words 按字典序排序，删除重复单词
    // 按长度排序，长度相同的单词维持字典序
    stable_sort(words.begin(), words.end(),
                [] (const string &a, const string &b)
                { return a.size() < b.size(); });
    // 获取一个迭代器，指向第一个满足 size()>= sz 的元素
    auto wc = find_if(words.begin(), words.end(),
                       [sz] (const string &a)
                       { return a.size() >= sz; });
    // 计算满足 size >= sz 的元素的数目
    auto count = words.end() - wc;
    cout << count << " " << make_plural(count, "word", "s")
        << " of length " << sz << " or longer" << endl;
    // 打印长度大于等于给定值的单词，每个单词后面接一个空格
    for_each(wc, words.end(),
             [] (const string &s) {cout << s << " ";});
    cout << endl;
}

int main(int argc, char *argv[])
{
    ifstream in(argv[1]);
    if (!in) {
        cout << "打开输入文件失败！" << endl;
        exit(1);
    }

    vector<string> words;
    string word;
    while (in >> word)

```

```

    words.push_back(word);

    biggies(words, 4);

    return 0;
}

```

**练习 10.17：**重写 10.3.1 节练习 10.12（第 345 页）的程序，在对 sort 的调用中使用 lambda 来代替函数 compareIsbn。

### 【出题思路】

继续练习 lambda，体会其与谓词的区别。

### 【解答】

此 lambda 比较两个给定的 Sales\_data 对象，因此捕获列表为空，有两个 Sales\_data 对象引用的参数。函数体则与 compareIsbn 相同。

```

#include <iostream>
#include <fstream>
#include <vector>
#include "Sales_data.h"
#include <algorithm>

using namespace std;

int main(int argc, char *argv[])
{
    ifstream in(argv[1]);
    if (!in) {
        cout << "打开输入文件失败！" << endl;
        exit(1);
    }

    vector<Sales_data> sds;
    Sales_data sd;
    while (read(in, sd))
        sds.push_back(sd);

    sort(sds.begin(), sds.end(),
        [] (const Sales_data &lhs, const Sales_data &rhs)
        { return lhs.isbn() < rhs.isbn(); });

    for (const auto &s : sds) {
        print(cout, s);
        cout << endl;
    }

    return 0;
}

```

**练习 10.18:** 重写 `biggies`, 用 `partition` 代替 `find_if`。我们在 10.3.1 节练习 10.13 (第 345 页) 中介绍了 `partition` 算法。

### 【出题思路】

理解 `find_if` 和 `partition` 的不同。

### 【解答】

对于本题而言, 若使用 `find_if`, 要求序列已按字符串长度递增顺序排好序。`find_if` 返回第一个长度 $\geq sz$  的字符串的位置 `wc`, 则所有满足长度 $\geq sz$  的字符串位于范围  $[wc, end)$  之间。

而 `partition` 不要求序列已排序, 它对所有字符串检查长度是否 $\geq sz$ , 将满足条件的字符串移动到序列前端, 不满足条件的字符串都移动到满足条件的字符串之后, 返回满足条件的范围的尾后迭代器。因此满足条件的字符串位于范围  $[begin, wc)$  之间。

因此, 在 `partition` 之前不再需要 `stable_sort`, 计数语句和打印语句也都需要进行相应修改。

```
#include <iostream>
#include <fstream>
#include <vector>
#include <string>
#include <algorithm>
#include "make_plural.h"

using namespace std;

void elimDups(vector<string> &words)
{
    sort(words.begin(), words.end());

    auto end_unique = unique(words.begin(), words.end());

    words.erase(end_unique, words.end());
}

void biggies(vector<string> &words,
             vector<string>::size_type sz)
{
    elimDups(words); // 将 words 按字典序排序, 删除重复单词
    for_each(words.begin(), words.end(),
              [] (const string &s){cout << s << " ";});
    cout << endl;
    // 获取一个迭代器, 指向最后一个满足 size()>= sz 的元素之后的位置
    auto wc = partition(words.begin(), words.end(),
                        [sz] (const string &a)
                            { return a.size() >= sz; });

    // 计算满足 size >= sz 的元素的数目
    auto count = wc - words.begin();
    cout << count << " " << make_plural(count, "word", "s")
        << " of length " << sz << " or longer" << endl;
    // 打印长度大于等于给定值的单词, 每个单词后面接一个空格
}
```

```

for_each(words.begin(), wc,
    [](const string &s){cout << s << " ";});
cout << endl;
}

int main(int argc, char *argv[])
{
    ifstream in(argv[1]);
    if (!in) {
        cout << "打开输入文件失败! " << endl;
        exit(1);
    }

    vector<string> words;
    string word;
    while (in >> word)
        words.push_back(word);

    biggies(words, 4);

    return 0;
}

```

**练习 10.19:** 用 `stable_partition` 重写前一题的程序，与 `stable_sort` 类似，在划分后的序列中维持原有元素的顺序。

**【出题思路】**

理解 `stable_partition` 和 `partition` 的不同。

**【解答】**

将上一题程序中的 `partition` 换为 `stable_partition` 即可。在输入文件上观察两个程序输出的不同。

**练习 10.20:** 标准库定义了一个名为 `count_if` 的算法。类似 `find_if`，此函数接受一对迭代器，表示一个输入范围，还接受一个谓词，会对输入范围内每个元素执行。`count_if` 返回一个计数值，表示谓词有多少次为真。使用 `count_if` 重写我们程序中统计多少单词长度超过 6 的部分。

**【出题思路】**

练习 `count_if` 算法的使用。

**【解答】**

若只统计容器中满足一定条件的元素的个数，而不打印或者获取这些元素的话。直接使用 `count_if` 即可，无须进行 `unique`、`sort` 等操作。

```

#include <iostream>
#include <fstream>
#include <vector>
#include <string>
#include <algorithm>

```

```

#include "make_plural.h"

using namespace std;

inline void output_words(vector<string> &words)
{
    for (auto iter = words.begin(); iter != words.end(); iter++)
        cout << *iter << " ";
    cout << endl;
}

void biggies(vector<string> &words,
             vector<string>::size_type sz)
{
    output_words(words);

    // 统计满足 size()>= sz 的元素的个数
    auto bc = count_if(words.begin(), words.end(),
                        [sz](const string &a)
    {
        return a.size() >= sz; });
    cout << bc << " " << make_plural(bc, "word", "s")
        << " of length " << sz << " or longer" << endl;
}

int main(int argc, char *argv[])
{
    ifstream in(argv[1]);
    if (!in) {
        cout << "打开输入文件失败!" << endl;
        exit(1);
    }

    vector<string> words;
    string word;
    while (in >> word)
        words.push_back(word);

    biggies(words, 6);

    return 0;
}

```

**练习 10.21：**编写一个 lambda，捕获一个局部 int 变量，并递减变量值，直至它变为 0。一旦变量变为 0，再调用 lambda 应该不再递减变量。lambda 应该返回一个 bool 值，指出捕获的变量是否为 0。

### 【出题思路】

练习 lambda 改变捕获变量值的方法。

### 【解答】

若 lambda 需要改变捕获的局部变量的值，需要在参数列表之后、函数体之前

使用 `mutable` 关键字。对于本题，由于 `lambda` 有两个返回语句（`i` 大于 0 时返回 `false`，等于 0 时返回 `true`），还需要显式指定 `lambda` 的返回类型——使用尾置返回类型，在参数列表后使用 `->bool`。注意，正确的顺序是 `mutable -> bool`。由于 `i` 的初始值为 5，程序执行后会打印 5 个 0 和 1 个 1。

```
#include <iostream>
#include <algorithm>

using namespace std;

void mutable_lambda(void)
{
    int i = 5;
    auto f = [i] () mutable -> bool{ if (i > 0) { i--; return false;} else
        return true; };

    for (int j = 0; j < 6; j++)
        cout << f() << " ";
    cout << endl;
}

int main(int argc, char *argv[])
{
    mutable_lambda();

    return 0;
}
```

**练习 10.22:** 重写统计长度小于等于 6 的单词数量的程序，使用函数代替 `lambda`。

#### 【出题思路】

本题练习用函数代替 `lambda` 的方法。

#### 【解答】

当 `lambda` 不捕获局部变量时，用函数代替它是很容易的。但当 `lambda` 捕获局部变量时就不那么简单了。因为在这种情况下，通常是算法要求可调用对象（`lambda`）接受的参数个数少于函数所需的参数个数，`lambda` 通过捕获的局部变量来弥补这个差距，而普通函数是无法办到的。

解决方法是使用标准库中的 `bind` 函数在实际工作函数外做一层“包装”——它接受一个可调用对象 A，即实际的工作函数，返回一个新的可调用对象 B，供算法使用。A 后面是一个参数列表，即传递给它的参数列表。其中有一些名字形如 `_n` 的参数，表示程序调用 B 时传递给它的第 `n` 个参数。也就是说，算法调用 B 时传递较少的参数，B 再补充其他一些值，形成更长的参数列表，从而解决算法要求的参数个数比实际工作函数所需参数个数少的问题。

注意，`_n` 定义在命名空间 `std::placeholders` 中。

```
#include <iostream>
#include <fstream>
#include <vector>
```

```

#include <string>
#include <algorithm>
#include <functional>
#include "make_plural.h"

using namespace std;
using namespace std::placeholders;

inline void output_words(vector<string> &words)
{
    for (auto iter = words.begin(); iter != words.end(); iter++)
        cout << *iter << " ";
    cout << endl;
}

bool check_size(const string &s, string::size_type sz)
{
    return s.size() >= sz;
}

void biggies(vector<string> &words,
             vector<string>::size_type sz)
{
    output_words(words);

    // 统计满足 size()>= sz 的元素的个数
    auto bc = count_if(words.begin(), words.end(),
                        bind(check_size, _1, sz));

    cout << bc << " " << make_plural(bc, "word", "s")
        << " of length " << sz << " or longer" << endl;
}

int main(int argc, char *argv[])
{
    ifstream in(argv[1]);
    if (!in) {
        cout << "打开输入文件失败!" << endl;
        exit(1);
    }

    vector<string> words;
    string word;
    while (in >> word)
        words.push_back(word);

    biggies(words, 6);

    return 0;
}

```

**练习 10.23:** bind 接受几个参数?**【出题思路】**

理解 bind 函数的使用。

**【解答】**

bind 是可变参数的。它接受的第一个参数是一个可调用对象，即实际工作函数 A，返回供算法使用的新的可调用对象 B。若 A 接受  $x$  个参数，则 bind 的参数个数应该是  $x+1$ ，即除了 A 外，其他参数应一一对应 A 所接受的参数。这些参数中有一部分来自于 B ( $_n$ )，另外一些来自于所处函数的局部变量。

**练习 10.24:** 给定一个 string，使用 bind 和 check\_size 在一个 int 的 vector 中查找第一个大于 string 长度的值。**【出题思路】**

本题继续练习 bind 的使用。

**【解答】**

解题思路与练习 10.22 类似。有两点需要注意：

1. 本题中 check\_size 应该检查 string 的长度是否小于等于长度值，而不是大于等于。
2. 对于 bind 返回的可调用对象，其唯一参数是 vector 中的元素，因此  $_1$  应该是 bind 的第三个参数，即 check\_size 的第二个参数（长度值），而给定 string 应作为 bind 的第三个即 check\_size 的第二个参数。

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#include <functional>
#include "make_plural.h"

using namespace std;
using namespace std::placeholders;

bool check_size(const string &s, string::size_type sz)
{
    return s.size() <= sz;
}

void biggies(vector<int> &vc, const string &s)
{
    // 查找第一个大于等于 s 长度的数值
    auto p = find_if(vc.begin(), vc.end(),
                      bind(check_size, s, _1));

    // 打印结果
    cout << "第" << p-vc.begin()+1 << "个数" << *p
        << "大于等于" << s << "的长度" << endl;
}
```

```

}

int main(int argc, char *argv[])
{
    vector<int> vc = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

    biggies(vc, "Hello");
    biggies(vc, "everyone");
    biggies(vc, "!");

    return 0;
}

```

**练习 10.25:** 在 10.3.2 节（第 349 页）的练习中，编写了一个使用 `partition` 的 `biggies` 版本。使用 `check_size` 和 `bind` 重写此函数。

### 【出题思路】

本题继续练习 `bind` 的使用。

### 【解答】

解题思路与练习 10.22 类似。

```

#include <iostream>
#include <fstream>
#include <vector>
#include <string>
#include <algorithm>
#include <functional>
#include "make_plural.h"

using namespace std;
using namespace std::placeholders;

void elimDups(vector<string> &words)
{
    sort(words.begin(), words.end());

    auto end_unique = unique(words.begin(), words.end());
    words.erase(end_unique, words.end());
}

bool check_size(const string &s, string::size_type sz)
{
    return s.size() >= sz;
}

void biggies(vector<string> &words,
             vector<string>::size_type sz)
{
    elimDups(words); // 将 words 按字典序排序，删除重复单词
    for_each(words.begin(), words.end(),
              [] (const string &s) {cout << s << " ";});
}

```

```

cout << endl;
// 获取一个迭代器，指向最后一个满足 size()>= sz 的元素之后的位置
auto wc = partition(words.begin(), words.end(),
                     bind(check_size, _1, sz));
// 计算满足 size >= sz 的元素的数目
auto count = wc - words.begin();
cout << count << " " << make_plural(count, "word", "s")
    << " of length " << sz << " or longer" << endl;
// 打印长度大于等于给定值的单词，每个单词后面接一个空格
for_each(words.begin(), wc,
         [] (const string &s){cout << s << " ";});
cout << endl;
}

int main(int argc, char *argv[])
{
    ifstream in(argv[1]);
    if (!in) {
        cout << "打开输入文件失败！" << endl;
        exit(1);
    }

    vector<string> words;
    string word;
    while (in >> word)
        words.push_back(word);

    biggies(words, 6);

    return 0;
}

```

**练习 10.26：**解释三种插入迭代器的不同之处。

**【出题思路】**

理解插入迭代器的概念，以及几种插入迭代器的不同。

**【解答】**

在书中前文，我们已经学习了一种插入迭代器 `back_inserter`。插入迭代器又称插入器，本质上是一种迭代器适配器。如前所述，标准库算法为了保证通用性，并不直接操作容器，而是通过迭代器来访问容器元素。因此，算法不具备直接向容器插入元素的能力。而插入器正是帮助算法实现向容器插入元素的机制。

除了 `back_inserter`，标准库还提供了另外两种插入器：`front_inserter` 和 `inserter`。三者的差异在于如何向容器插入元素：`back_inserter` 调用 `push_back`，`front_inserter` 调用 `push_front`，`inserter` 则调用 `insert`。显然，这也决定了它们插入元素位置的不同。`back_inserter` 总是插入到容器尾元素之后，`front_inserter` 总是插入到容器首元素之前，而 `inserter` 则是插入到给定位置（作为 `inserter` 的第二个参数传递给它）之前。因此，需要注意这些

特点带来的元素插入效果的差异。例如，使用 `front_inserter` 向容器插入一些元素，元素最终在容器中的顺序与插入顺序相反，但 `back_inserter` 和 `inserter` 则不会有这个问题。

**练习 10.27：**除了 `unique`（参见 10.2.3 节，第 343 页）之外，标准库还定义了名为 `unique_copy` 的函数，它接受第三个迭代器，表示拷贝不重复元素的目的位置。编写一个程序，使用 `unique_copy` 将一个 `vector` 中不重复的元素拷贝到一个初始为空的 `list` 中。

#### 【出题思路】

本题练习 `unique_copy` 的使用，以及使用插入迭代器帮助算法(`unique_copy`)实现向容器插入新元素。

#### 【解答】

本题要求将 `vector` 中不重复元素按原顺序拷贝到空 `list` 中，因此使用 `back_inserter` 即可。需要注意的是，与 `unique` 一样，`unique_copy` 也要求在源容器中重复元素是相邻存放的。因此，若 `vector` 未排序且重复元素未相邻存放，`unique_copy` 就会失败。稳妥的方法是先对 `vector` 排序。

```
#include <iostream>
#include <vector>
#include <list>
#include <algorithm>

using namespace std;

int main(int argc, char *argv[])
{
    vector<int> vi = { 1, 2, 2, 3, 4, 5, 5, 6 };
    list<int> li;

    unique_copy(vi.begin(), vi.end(), back_inserter(li));

    for (auto v : li)
        cout << v << " ";
    cout << endl;

    return 0;
}
```

#### 【其他解题思路】

由于要保持原顺序，显然使用 `inserter` 也是可以的，将 `back_inserter(li)` 替换为 `inserter(li, li.begin())` 即可。

**练习 10.28：**一个 `vector` 中保存 1 到 9 共 9 个值，将其拷贝到三个其他容器中。分别使用 `inserter`、`back_inserter` 和 `front_inserter` 将元素添加到三个容器中。对每种 `inserter`，估计输出序列是怎样的，运行程序验证你的估计是否

正确。

### 【出题思路】

进一步理解三种插入迭代器的差异，并练习使用它们。

### 【解答】

若三个目的容器均为空，则显然 inserter 和 back\_inserter 的输出结果是 "1 2 3 4 5 6 7 8 9"，而 front\_inserter 的结果是 "9 8 7 6 5 4 3 2 1"。但如果目的容器不空，则 inserter 的结果取决于传递给它的第二个参数（一个迭代器）指向什么位置。

```
#include <iostream>
#include <vector>
#include <list>
#include <algorithm>

using namespace std;

int main(int argc, char *argv[])
{
    vector<int> vi = { 1, 2, 3, 4, 5, 6, 7, 8, 9};
    list<int> li1, li2, li3;

    unique_copy(vi.begin(), vi.end(), inserter(li1, li1.begin()));
    for (auto v : li1)
        cout << v << " ";
    cout << endl;

    unique_copy(vi.begin(), vi.end(), back_inserter(li2));
    for (auto v : li2)
        cout << v << " ";
    cout << endl;

    unique_copy(vi.begin(), vi.end(), front_inserter(li3));
    for (auto v : li3)
        cout << v << " ";
    cout << endl;

    return 0;
}
```

**练习 10.29：**编写程序，使用流迭代器读取一个文本文件，存入一个 vector 中的 string 里。

### 【出题思路】

本题练习流迭代器的简单使用。

### 【解答】

虽然流不是容器，但标准库提供了通过迭代器访问流的方法。声明一个流迭代器时，需指出所绑定的流。对于本题，首先打开一个文本文件，将此文件的流作为参数提供给流迭代器的构造函数即可。当默认构造流迭代器时，得到一个尾后迭代

器，对应文件结束。

```
#include <iostream>
#include <fstream>
#include <vector>
#include <string>
#include <iterator>
#include <algorithm>

using namespace std;

int main(int argc, char *argv[])
{
    ifstream in(argv[1]);
    if (!in) {
        cout << "打开输入文件失败！" << endl;
        exit(1);
    }

    // 创建流迭代器从文件读入字符串
    istream_iterator<string> in_iter(in);
    // 尾后迭代器
    istream_iterator<string> eof;
    vector<string> words;
    while (in_iter != eof)
        words.push_back(*in_iter++); // 存入 vector 并递增迭代器

    for (auto word : words)
        cout << word << " ";
    cout << endl;

    return 0;
}
```

**练习 10.30：**使用流迭代器、`sort` 和 `copy` 从标准输入读取一个整数序列，将其排序，并将结果写到标准输出。

#### 【出题思路】

本题练习输出流迭代器的使用。

#### 【解答】

使用流迭代器从标准输入读取整数序列的程序与上一题类似，创建流迭代器时指出是 `int`，并用 `cin` 代替文件流对象即可。

用 `copy` 将整数写到标准输出，需要声明一个输出流迭代器，作为第三个参数传递给 `copy`。将 `cin` 传递给输出流迭代器的构造函数，`copy` 即可将整数写到标准输出。将" "作为第二个参数传递给输出流迭代器的构造函数，表示在每个输出之后写一个空格，从而将整数分隔开来输出。

```
#include <iostream>
#include <fstream>
#include <vector>
```

```

#include <iterator>
#include <algorithm>

using namespace std;

int main(int argc, char *argv[])
{
    // 创建流迭代器从标准输入读入整数
    istream_iterator<int> in_iter(cin);
    // 尾后迭代器
    istream_iterator<int> eof;
    vector<int> vi;
    while (in_iter != eof)
        vi.push_back(*in_iter++); // 存入 vector 并递增迭代器

    sort(vi.begin(), vi.end());

    ostream_iterator<int> out_iter(cout, " ");
    copy(vi.begin(), vi.end(), out_iter);

    return 0;
}

```

**练习 10.31：**修改前一题的程序，使其只打印不重复的元素。你的程序应使用 unique\_copy（参见 10.4.1 节，第 359 页）。

#### 【出题思路】

继续练习输出流迭代器的使用，并复习 unique\_copy 的使用。

#### 【解答】

用 unique\_copy 替代上题的 copy 即可。

```

#include <iostream>
#include <fstream>
#include <vector>
#include <iterator>
#include <algorithm>

using namespace std;

int main(int argc, char *argv[])
{
    // 创建流迭代器从标准输入读入整数
    istream_iterator<int> in_iter(cin);
    // 尾后迭代器
    istream_iterator<int> eof;
    vector<int> vi;
    while (in_iter != eof)
        vi.push_back(*in_iter++); // 存入 vector 并递增迭代器

    sort(vi.begin(), vi.end());

    ostream_iterator<int> out_iter(cout, " ");

```

```

unique_copy(vi.begin(), vi.end(), out_iter);

return 0;
}

```

**练习 10.32：**重写 1.6 节（第 21 页）中的书店程序，使用一个 `vector` 保存交易记录，使用不同算法完成处理。使用 `sort` 和 10.3.1 节（第 345 页）中的 `compareIsbn` 函数来排序交易记录，然后使用 `find` 和 `accumulate` 求和。

### 【出题思路】

继续练习流迭代器的使用，并复习算法的使用。

### 【解答】

读取交易记录存入 `vector` 的程序与前两题类似。

将 `compareIsbn` 作为第三个参数传递给 `sort`，即可实现将交易记录按 ISBN 排序。

本题要求将 ISBN 相同的交易记录累加，而 `find` 算法查找与给定值相同的容器元素，需要逐个查找需要累加的元素，显然性能太差。我们使用 `find_if`，并构造如下 `lambda`：

```
[item] (const Sales_item &item1) { return item1.isbn() != item.isbn(); }
```

作为第三个参数传递给 `find_if`，从而查找到第一个 ISBN 与当前交易 1 记录不同的记录 `r`（ISBN 相同的范围的尾后位置）。

接下来调用 `accumulate` 即可实现范围 `[l, r)` 间交易记录的累加。

最后将 `l` 移动到 `r`，继续循环，计算下一段交易记录的查找和累加。

```

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

using namespace ::std;

int main()
{
    vector<Sales_item> vs;
    istream_iterator<Sales_item> in_iter(cin);
    istream_iterator<Sales_item> eof;

    // 读入交易记录，存入 vector
    while (in_iter != eof)
        vs.push_back(*in_iter++);

    if (vs.empty()) {
        // 没有输入！警告读者
        std::cerr << "No data?!" << std::endl;
        return -1; // 表示失败
    }
}
```

```

// 将交易记录按 ISBN 排序
sort(vs.begin(), vs.end(), compareIsbn);

auto l = vs.begin();
while (l != vs.end()) {
    auto item = *l; // 相同 ISBN 的交易记录中的第一个
    // 在后续记录中查找第一个 ISBN 与 item 不同者
    auto r = find_if(l + 1, vs.end(),
                      [item] (const Sales_item &item1)
                      { return item1.isbn() != item.isbn(); } );
    // 将范围[l, r)间的交易记录累加并输出
    cout << accumulate(l + 1, r, item) << endl;
    // l 指向下一段交易记录中的第一个
    l = r;
}

return 0;
}

```

**练习 10.33:** 编写程序，接受三个参数：一个输入文件和两个输出文件的文件名。输入文件保存的应该是整数。使用 `istream_iterator` 读取输入文件。使用 `ostream_iterator` 将奇数写入第一个输出文件，每个值之后都跟一个空格。将偶数写入第二个输出文件，每个值都独占一行。

#### 【出题思路】

本题通过一个稍大的例子巩固输入和输出流迭代器的使用。

#### 【解答】

程序从命令行接受三个文件名参数，因此程序首先判断参数数目是否为 4（包括程序名）。

然后依次打开输入文件和两个输出文件，再用这三个流初始化一个输入流迭代器和两个输出流迭代器。注意，第一个流迭代器输出以空格间隔的奇数，将“ ”作为第二个参数传递给构造函数；第二个流迭代器输出以换行间隔的偶数，将“\n”作为第二个参数传递给构造函数。

随后在循环中通过输入流迭代器读取文件中的整数，直至到达文件末尾。读入每个整数后，判断它是奇数还是偶数，分别写入两个输出文件。

```

#include <iostream>
#include <fstream>
#include <iterator>
#include <algorithm>

using namespace std;

int main(int argc, char *argv[])
{
    if (argc != 4) {
        cout << "用法: execise.exe in_file "

```

```

    "out_file1 out_file2" << endl;
    return -1;
}

ifstream in(argv[1]);
if (!in) {
    cout << "打开输入文件失败!" << endl;
    exit(1);
}

ofstream out1(argv[2]);
if (!out1) {
    cout << "打开输出文件 1 失败!" << endl;
    exit(1);
}

ofstream out2(argv[3]);
if (!out2) {
    cout << "打开输出文件 2 失败!" << endl;
    exit(1);
}

// 创建流迭代器从文件读入整数
istream_iterator<int> in_iter(in);
// 尾后迭代器
istream_iterator<int> eof;
// 第一个输出文件以空格间隔整数
ostream_iterator<int> out_iter1(out1, " ");
// 第二个输出文件以换行间隔整数
ostream_iterator<int> out_iter2(out2, "\n");
while (in_iter != eof) {
    if (*in_iter & 1) // 奇数写入第一个输出文件
        *out_iter1++ = *in_iter;
    else *out_iter2++ = *in_iter; // 偶数写入第二个输出文件
    in_iter++;
}

return 0;
}

```

**练习 10.34：**使用 `reverse_iterator` 逆序打印一个 `vector`。

#### 【出题思路】

本题练习反向迭代器的简单使用。

#### 【解答】

我们可以用 `(c) rbegin` 获取反向遍历的起始位置（其实是容器的末尾元素位置），用 `(c) rend` 获取尾后迭代器（首元素之前的位置）。通过这两个迭代器控制循环，即可实现对容器的反向遍历。注意，循环中向前移动迭代器仍然用`++`，而非`--`。

```
#include <iostream>
#include <fstream>
```

```

#include <vector>
#include <algorithm>

using namespace std;

int main(int argc, char *argv[])
{
    if (argc != 2) {
        cout << "用法: execise.exe in_file" << endl;
        return -1;
    }

    ifstream in(argv[1]);
    if (!in) {
        cout << "打开输入文件失败!" << endl;
        exit(1);
    }

    vector<int> vi;
    int v;
    while (in >> v) // 从文件中读取整数
        vi.push_back(v);

    for (auto r_iter = vi.crbegin(); r_iter != vi.crend();
         ++r_iter)
        cout << *r_iter << " ";
    cout << endl;

    return 0;
}

```

**练习 10.35:** 使用普通迭代器逆序打印一个 vector。

### 【出题思路】

体会反向迭代器和普通迭代器的差异。

### 【解答】

若使用普通迭代器反向遍历容器，首先通过 `cend` 获得容器的尾后迭代器，循环中递减该迭代器，直到它与 `begin` 相等为止。但需要注意的是，遍历所用迭代器的初值为尾后位置，终值为 `begin` 之后的位置。也就是说，在每个循环步中，它指向的都是我们要访问的元素之后的位置。因此，我们在循环中首先将其递减，然后通过它访问容器元素，而在循环语句的第三个表达式中就不再递减迭代器了。

显然，对于反向遍历容器，使用反向迭代器比普通迭代器更清晰易懂。

```

#include <iostream>
#include <fstream>
#include <vector>
#include <algorithm>

using namespace std;

```

```

int main(int argc, char *argv[])
{
    if (argc != 2) {
        cout << "用法: excise.exe in_file" << endl;
        return -1;
    }

    ifstream in(argv[1]);
    if (!in) {
        cout << "打开输入文件失败!" << endl;
        exit(1);
    }

    vector<int> vi;
    int v;
    while (in >> v) // 从文件中读取整数
        vi.push_back(v);

    for (auto r_iter = vi.cend(); r_iter != vi.begin();)
        cout << *(--r_iter) << " ";
    cout << endl;

    return 0;
}

```

**练习 10.36:** 使用 `find` 在一个 `int` 的 `list` 中查找最后一个值为 0 的元素。

### 【出题思路】

练习反向迭代器和算法的结合。

### 【解答】

借助反向迭代器，可以扩展算法的能力。例如，使用普通迭代器，`find` 能查找给定值在容器中第一次出现的位置。如果要查找最后一次出现的位置，还使用普通迭代器的话，代码会很复杂。但借助反向迭代器，`find` 可以逆序遍历容器中的元素，从而“第一次出现位置”实际上也就是正常顺序的最后一次出现位置了。

注意：

1. 由于 `list` 是链表数据结构，元素不连续存储，其迭代器不支持算术运算。因此，程序中用一个循环来计数位置编号。

2. 由于程序计数的是正向位置编号，因此，需要将 `find` 找到的反向迭代器 `last_z` 转换为普通迭代器（使用 `base` 成员函数）。但要注意，反向迭代器与普通迭代器的转换是左闭合区间的转换，而非精确位置的转换。`last_z.base()` 指向的并非最后一个 0，而是它靠近容器尾方向的邻居。因此，首先将 `last_z` 向容器首方向推进一个位置（`++`），然后再调用 `base`，得到的就是指向最后一个 0 的普通迭代器了。读者可尝试对本例画出类似图 10.2 所示的迭代器位置关系图。

```
#include <iostream>
```

```

#include <list>
#include <algorithm>

using namespace std;

int main(int argc, char *argv[])
{
    list<int> li = { 0, 1, 2, 0, 3, 4, 5, 0, 6 };
    // 利用反向迭代器查找最后一个 0
    auto last_z = find(li.rbegin(), li.rend(), 0);
    // 将迭代器向链表头方向推进一个位置
    // 转换为普通迭代器时，将回到最后一个 0 的位置
    last_z++;
    int p = 1;
    // 用 base 将 last_z 转换为普通迭代器
    // 从链表头开始遍历，计数最后一个 0 的编号
    for (auto iter = li.begin(); iter != last_z.base();
        iter++, p++) ;

    if (p >= li.size()) // 未找到 0
        cout << "容器中没有 0" << endl;
    else cout << "最后一个 0 在第" << p << "个位置" << endl;

    return 0;
}

```

### 【其他解题思路】

如果用普通迭代器求解本题，算法将会变得很复杂。

首先用 `find` 查找范围 `[begin, end)` 中的 0，若不存在，则输出序列中没有 0，算法结束。

否则，循环地查找下一个 0 的位置，直至不再有 0 为止，上一个 0 即为最后一个 0。在循环中，维护“上一个 0 的位置”和“当前 0 的位置”两个迭代器需要特别小心。

```

#include <iostream>
#include <list>
#include <algorithm>

using namespace std;

int main(int argc, char *argv[])
{
    list<int> li = { 0, 1, 2, 0, 3, 4, 5, 0, 6 };
    // 用普通迭代器查找第一个 0 的位置
    auto prev = find(li.begin(), li.end(), 0);
    if (prev == li.end()) // 未找到
        cout << "容器中没有 0" << endl;
    else {
        auto curr = prev;
        // 继续寻找后续 0，直到到达尾后位置
        while (curr != li.end()) {

```

```

        // 记住上一个0的位置
        prev = curr;
        curr++;
        // 寻找下一个0
        curr = find(curr, li.end(), 0);
    }
    // 从链表头开始计数 prev 的位置编号
    int p = 1;
    for (auto iter = li.begin(); iter != prev;
        iter++, p++) ;
    cout << "最后一个0在第" << p << "个位置" << endl;
}

return 0;
}

```

**练习 10.37:** 给定一个包含 10 个元素的 vector，将位置 3 到 7 之间的元素按逆序拷贝到一个 list 中。

### 【出题思路】

深入理解反向迭代器和普通迭代器间的差异及相互转换。

### 【解答】

反向迭代器和普通迭代器的转换是左闭合区间的转换。

对 10 个元素的 vector vi，包含位置 3~7 之间元素的迭代器区间如下所示：

|   |   |   |    |   |   |   |    |   |   |
|---|---|---|----|---|---|---|----|---|---|
| 0 | 1 | 2 | 3  | 4 | 5 | 6 | 7  | 8 | 9 |
|   |   |   | i1 |   |   |   | i2 |   |   |

第一个迭代器是 vi.begin() + 2，第二个迭代器指向位置 8，即 vi.begin() + 7。

当将这两个迭代器转换为反向迭代器时，位置如下：

|   |   |   |    |   |   |   |    |   |   |
|---|---|---|----|---|---|---|----|---|---|
| 0 | 1 | 2 | 3  | 4 | 5 | 6 | 7  | 8 | 9 |
|   |   |   | re |   |   |   | rb |   |   |

虽然与正向迭代器的位置不同，但左闭合区间 [rb, re) 仍然对应位置 3~7 之间的元素。显然，普通-反向迭代器间的这种错位，恰恰是因为标准库的范围概念是左闭合区间造成的。

另外，注意 back\_inserter 和流迭代器的使用。

```

#include <iostream>
#include <vector>
#include <list>
#include <iterator>
#include <algorithm>

using namespace std;

int main(int argc, char *argv[])
{
    ostream_iterator<int> out_iter(cout, " ");

```

```

vector<int> vi = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
// 用流迭代器和 copy 输出 int 序列
copy(vi.begin(), vi.end(), out_iter);
cout << endl;

list<int> li;
// 将 vi[2], 也就是第 3 个元素的位置转换为反向迭代器
vector<int>::reverse_iterator re(vi.begin() + 2);
// 将 vi[7], 也就是第 8 个元素的位置转换为反向迭代器
vector<int>::reverse_iterator rb(vi.begin() + 7);
// 用反向迭代器将元素逆序拷贝到 list
copy(rb, re, back_inserter(li));
copy(li.begin(), li.end(), out_iter);
cout << endl;
return 0;
}

```

**【其他解题思路】**

也可以通过 `rbegin()` 获得反向迭代器的首位置（正向的尾元素），然后正确计算偏移量，来获得正确的反向范围，但计算上需要非常小心。

```

#include <iostream>
#include <vector>
#include <list>
#include <iterator>
#include <algorithm>

using namespace std;

int main(int argc, char *argv[])
{
    ostream_iterator<int> out_iter(cout, " ");
    vector<int> vi = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    copy(vi.begin(), vi.end(), out_iter);
    cout << endl;

    list<int> li;
    copy(vi.rbegin() + vi.size() - 7, vi.rbegin() + vi.size() - 3 + 1,
         back_inserter(li));
    copy(li.begin(), li.end(), out_iter);
    cout << endl;
    return 0;
}

```

**练习 10.38：**列出 5 个迭代器类别，以及每类迭代器所支持的操作。

**【出题思路】**

理解 5 种迭代器及它们的差异。

**【解答】**

**输入迭代器：**只读，不写；单遍扫描，只能递增；还支持相等性判定运算符（`==`、`!=`）、解引用运算符（`*`）（只出现在赋值运算符右侧）和箭头运算符（`->`）。

输出迭代器：只写，不读；单遍扫描，只能递增，支持解引用运算符(\*)（只出现在赋值运算符左侧）。

前向迭代器：可读写；多遍扫描，只能递增，支持所有输入、输出迭代器的操作。

双向迭代器：可读写；多遍扫描，可递增递减，支持所有前向迭代器操作。

随机访问迭代器：可读写，多遍扫描，支持全部迭代器运算，除了上述迭代器类别支持的操作外，还有比较两个迭代器相对位置的关系运算符(<、<=、>和>=)、迭代器和一个整数值的加减运算(+、+=、-和-=)令迭代器在序列中前进或后退给定整数个元素、两个迭代器上的减法运算符(-)得到其距离以及下标运算符。

### 练习 10.39: list 上的迭代器属于哪类？vector 呢？

#### 【出题思路】

理解常用容器的迭代器类型。

#### 【解答】

list 上的迭代器是双向迭代器，vector 上的迭代器是随机访问迭代器。

### 练习 10.40: 你认为 copy 要求哪类迭代器？reverse 和 unique 呢？

#### 【出题思路】

理解算法对迭代器类型的要求。

#### 【解答】

copy 要求前两个参数至少是输入迭代器，表示一个输入范围。它读取这个范围中的元素，写入到第三个参数表示的输出序列中，因此第三个参数至少是输出迭代器。

reverse 要反向处理序列，因此它要求两个参数至少是双向迭代器。

unique 顺序扫描元素，覆盖重复元素，因此要求两个参数至少是前向迭代器。

“至少”意味着能力更强的迭代器是可接受的。

### 练习 10.41: 仅根据算法和参数的名字，描述下面每个标准库算法执行什么操作：

```
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);
```

#### 【出题思路】

理解标准库算法的命名规范。

#### 【解答】

1. 将范围[beg, end]间值等于 old\_val 的元素替换为 new\_val。
2. 将范围[beg, end]间满足谓词 pred 的元素替换为 new\_val。

3. 将范围 [beg, end) 间的元素拷贝到目的序列 dest 中，将其中值等于 old\_val 的元素替换为 new\_val。

4. 将范围 [beg, end) 间的元素拷贝到目的序列 dest 中，将其中满足谓词 pred 的元素替换为 new\_val。

**练习 10.42：**使用 list 代替 vector 重新实现 10.2.3 节（第 343 页）中的去除重复单词的程序。

#### 【出题思路】

练习链表的特殊操作。

#### 【解答】

本题要使用链表专用的 sort 和 unique 算法，与泛型算法的不同点有如下两点：

1. 它们是以链表类的成员函数形式实现的，因此使用方式是在链表对象上调用它们，也并不需要迭代器参数指出处理的序列。

2. 由于是以成员函数形式实现的，是直接操作容器而非通过迭代器访问容器元素，因此这些算法具有修改容器的能力（添加、删除元素）。例如，unique 会调用 erase 直接真正删除重复元素，容器的大小会变小，而不是像泛型 unique 算法那样只是覆盖重复元素，并不改变容器大小。因此程序已不再需要调用 erase 了。

建议读者好好体会泛型算法和专用算法之间的差异，包括上述使用方式上的差异，以及从库的开发者的角度思考两种方式的差异。

```
#include <iostream>
#include <fstream>
#include <list>
#include <string>
#include <algorithm>

using namespace std;

inline void output_words(list<string> &words)
{
    for (auto iter = words.begin(); iter != words.end(); iter++)
        cout << *iter << " ";
    cout << endl;
}

void elimDups(list<string> &words)
{
    output_words(words);

    words.sort();
    output_words(words);

    words.unique();
    output_words(words);
}
```

```
int main(int argc, char *argv[])
{
    ifstream in(argv[1]);
    if (!in) {
        cout << "打开输入文件失败！" << endl;
        exit(1);
    }

    list<string> words;
    string word;
    while (in >> word)
        words.push_back(word);

    elimDups(words);
    return 0;
}
```

# 第 11 章

# 关联容器

## 导读

本章介绍了标准库关联容器，包括：

- 关联容器的概念和简单使用。
- 关联容器涉及的类型和操作，特别是与顺序容器的差异。
- 无序关联容器，特别是与有序容器的差异。

本章练习的最重要目的是让读者理解关联容器的思想及其与顺序容器的差异，学会根据实际问题的特点选择适合的容器。具体内容除了关联容器基本类型和操作的练习之外，还有一些较大的练习以及与实际问题相关的练习。

**练习 11.1：**描述 map 和 vector 的不同。

### 【出题思路】

理解顺序容器和关联容器的不同。

### 【解答】

学习关联容器，理解与顺序容器的不同，最关键的是理解其基础的数据结构，随后它所表现出来的一些性质就很自然能够理解了。

两类容器的根本差别在于，顺序容器中的元素是“顺序”存储的（链表容器中的元素虽然不是在内存中“连续”存储的，但仍然是按“顺序”存储的）。理解“顺序”的关键，是理解容器支持的操作形式以及效率。

对于 vector 这样的顺序容器，元素在其中按顺序存储，每个元素有唯一对应的位置编号，所有操作都是按编号（位置）进行的。例如，获取元素（头、尾、用下标获取任意位置）、插入删除元素（头、尾、任意位置）、遍历元素（按元素位置顺序逐一访问）。底层的数据结构是数组、链表，简单但已能保证上述操作的高效。

而对于依赖值的元素访问，例如查找（搜索）给定值（`find`），在这种数据结构上的实现是要通过遍历完成的，效率不佳。

而`map`这种关联容器，就是为了高效实现“按值访问元素”这类操作而设计的。为了达到这一目的，容器中的元素是按关键字值存储的，关键字值与元素数据建立起对应关系，这就是“关联”的含义。底层数据结构是红黑树、哈希表等，可高效实现按关键字值查找、添加、删除元素等操作。

**练习 11.2：**分别给出最适合使用`list`、`vector`、`deque`、`map`以及`set`的例子。

#### 【出题思路】

理解顺序容器和关联容器的适用范围。

#### 【解答】

若元素很小（例如`int`），大致数量预先可知，在程序运行过程中不会剧烈变化，大部分情况下只在末尾添加或删除需要频繁访问任意位置的元素，则`vector`可带来最高的效率。若需要频繁在头部和尾部添加或删除元素，则`deque`是最好的选择。

如果元素较大（如大的类对象），数量预先不知道，或是程序运行过程中频繁变化，对元素的访问更多是顺序访问全部或很多元素，则`list`很适合。

`map`很适合对一些对象按它们的某个特征进行访问的情形。典型的例如按学生的名字来查询学生信息，即可将学生名字作为关键字，将学生信息作为元素值，保存在`map`中。

`set`，顾名思义，就是集合类型。当需要保存特定的值集合——通常是满足/不满足某种要求的值集合，用`set`最为方便。

**练习 11.3：**编写你自己的单词计数程序。

#### 【出题思路】

练习`map`的简单使用。

#### 【解答】

参照本节例子完成完整程序即可。

```
#include <iostream>
#include <fstream>
#include <map>
#include <string>
#include <algorithm>

using namespace std;

int main(int argc, char *argv[])
{
    ifstream in(argv[1]);
    if (!in) {

```

```

cout << "打开输入文件失败!" << endl;
exit(1);
}

map<string, size_t> word_count;           // string 到 count 的映射
string word;
while (in >> word)
    ++word_count[word];                  // 这个单词的出现次数加 1

for (const auto &w : word_count)          // 对 map 中的每个元素
    // 打印结果
    cout << w.first << "出现了" << w.second << "次" << endl;

return 0;
}

```

**练习 11.4:** 扩展你的程序，忽略大小写和标点。例如，example..、example，和 Example，应该递增相同的计数器。

#### 【出题思路】

此题并非练习 set 的使用，而是字符串的处理。

#### 【解答】

编写函数 trans，将单词中的标点去掉，将大写都转换为小写。具体方法是：遍历字符串，对每个字符首先检查是否是大写（ASCII 值在 A 和 Z 之间），若是，将其转换为小写（减去 A，加上 a）；否则，检查它是否带标点，若是，将其删除。最终，将转换好的字符串返回。

```

#include <iostream>
#include <fstream>
#include <map>
#include <string>
#include <algorithm>

using namespace std;

string &trans(string &s)
{
    for (int p = 0; p < s.size(); p++) {
        if (s[p] >= 'A' && s[p] <= 'Z')
            s[p] -= ('A' - 'a');
        else if (s[p] == ',' || s[p] == '.')
            s.erase(p, 1);
    }
    return s;
}

int main(int argc, char *argv[])
{
    ifstream in(argv[1]);
    if (!in) {

```

```

cout << "打开输入文件失败!" << endl;
exit(1);
}

map<string, size_t> word_count;           // string 到 count 的映射
string word;
while (in >> word)
    ++word_count[trans(word)];            // 这个单词的出现次数加 1

for (const auto &w : word_count)          // 对 map 中的每个元素
    // 打印结果
    cout << w.first << "出现了" << w.second << "次" << endl;

return 0;
}

```

**练习 11.5:** 解释 map 和 set 的差别。你如何选择使用哪个？

**【出题思路】**

理解两种关联容器的差别。

**【解答】**

当需要查找给定值所对应的数据时，应使用 map，其中保存的是<关键字，值>对，按关键字访问值。

如果只需判定给定值是否存在时，应使用 set，它是简单的值的集合。

**练习 11.6:** 解释 set 和 list 的差别。你如何选择使用哪个？

**【出题思路】**

理解关联容器和顺序容器的差别。

**【解答】**

两者都可以保存元素集合。

如果只需要顺序访问这些元素，或是按位置访问元素，那么应使用 list。

如果需要快速判定是否有元素等于给定值，则应使用 set。

**练习 11.7:** 定义一个 map，关键字是家庭的姓，值是一个 vector，保存家中孩子（们）的名。编写代码，实现添加新的家庭以及向已有家庭中添加新的孩子。

**【出题思路】**

理解 map 的稍复杂的使用。

**【解答】**

此 map 的关键字类型是 string，值类型是 vector<string>。

我们定义函数 add\_family 添加一个家庭，注意，必须先检查是否已有这个家庭，若不做这个检查，则可能将已有家庭的孩子名字清空（如 main 函数中的王姓家

庭的添加顺序)。若确实还没有这个家庭，则创建一个空的 `vector<string>`，表示这个家庭的孩子名字列表。

函数 `add_child` 向一个已有家庭添加孩子的名字：首先用 `[]` 运算符取出该家庭的 `vector`，然后调用 `push_back` 将名字追加到 `vector` 末尾。

```
#include <iostream>
#include <map>
#include <string>
#include <algorithm>

using namespace std;

void add_family(map<string, vector<string>> &families, const string &family)
{
    if (families.find(family) == families.end())
        families[family] = vector<string>();
}

void add_child(map<string, vector<string>> &families, const string &family, const string &child)
{
    families[family].push_back(child);
}

int main(int argc, char *argv[])
{
    map<string, vector<string>> families;

    add_family(families, "张");
    add_child(families, "张", "强");
    add_child(families, "张", "刚");
    add_child(families, "王", "五");
    add_family(families, "王");

    for (auto f : families) {
        cout << f.first << "家的孩子：" ;
        for (auto c : f.second)
            cout << c << " ";
        cout << endl;
    }

    return 0;
}
```

### 【其他解题思路】

`add_family` 的函数体其实可以有一个非常简单的实现：

```
families[family];
```

当该家庭已存在时，此语句只是获取其 `vector`，不会导致 `vector` 有任何变化；若该家庭不存在，标准库 `map` 的实现机制是在容器中为该关键字创建一个对象，进

行默认初始化，即构造一个空 vector。与 if 版本的效果完全一致。

这也是 add\_child 为什么不需要检查家庭是否存在原因，当家庭存在时，将孩子的名字追加到现有 vector 的末尾；若家庭不存在，标准库会先创建一个新的空 vector，然后我们的程序将孩子名字添加进去。

**练习 11.8：**编写一个程序，在一个 vector 而不是一个 set 中保存不重复的单词。使用 set 的优点是什么？

### 【出题思路】

通过实际编程理解 set 和 vector 的差别。

### 【解答】

使用 vector 保存不重复单词，需要用 find 查找新读入的单词是否已在 vector 中，若不在（返回尾后迭代器），才将单词加入 vector。

而使用 set，检查是否重复的工作是由 set 模板负责的，程序员无须编写对应代码，程序简洁很多。

更深层次的差别，vector 是无序线性表，find 查找指定值只能采用顺序查找方式，所花费的时间与 vector.size() 呈线性关系。而 set 是用红黑树实现的，花费的时间与 vector.size() 呈对数关系。当单词数量已经非常多时，set 的性能优势是巨大的。

当然，vector 也不是毫无用处。它可以保持单词的输入顺序，而 set 则不能，遍历 set，元素是按值的升序被遍历的。

vector 版本：

```
#include <iostream>
#include <fstream>
#include <vector>
#include <string>
#include <algorithm>

using namespace std;

string &trans(string &s)
{
    for (int p = 0; p < s.size(); p++) {
        if (s[p] >= 'A' && s[p] <= 'Z')
            s[p] -= ('A' - 'a');
        else if (s[p] == ',' || s[p] == '.')
            s.erase(p, 1);
    }
    return s;
}

int main(int argc, char *argv[])
{
    ifstream in(argv[1]);
    if (!in) {
```

```

cout << "打开输入文件失败!" << endl;
exit(1);
}

vector<string> unique_word;           // 不重复的单词
string word;
while (in >> word) {
    trans(word);
    if (find(unique_word.begin(), unique_word.end()
        , word) == unique_word.end())
        unique_word.push_back(word);      // 添加不重复单词
}

for (const auto &w : unique_word) // 打印不重复单词
    // 打印结果
    cout << w << " ";
cout << endl;

return 0;
}

```

set 版本:

```

#include <iostream>
#include <fstream>
#include <set>
#include <string>
#include <algorithm>

using namespace std;

string &trans(string &s)
{
    for (int p = 0; p < s.size(); p++) {
        if (s[p] >= 'A' && s[p] <= 'Z')
            s[p] -= ('A' - 'a');
        else if (s[p] == ',' || s[p] == '.')
            s.erase(p, 1);
    }
    return s;
}

int main(int argc, char *argv[])
{
    ifstream in(argv[1]);
    if (!in) {
        cout << "打开输入文件失败!" << endl;
        exit(1);
    }

    set<string> unique_word;           // 不重复的单词
    string word;
    while (in >> word) {
        trans(word);

```

```

        unique_word.insert(word);           // 添加不重复单词
    }

    for (const auto &w : unique_word)   // 打印不重复单词
        // 打印结果
        cout << w << " ";
    cout << endl;

    return 0;
}

```

**练习 11.9：**定义一个 map，将单词与一个行号的 list 关联，list 中保存的是单词所出现的行号。

### 【出题思路】

练习 map 的使用。

### 【解答】

map 的定义为：

```
map<string, list<int>> word_lineno;
```

完整程序如下所示。其中用 `getline` 读取一行，统计行号。再用字符串流读取这行中所有单词，记录单词行号。参见第 8 章内容。

```

#include <iostream>
#include <fstream>
#include <sstream>
#include <map>
#include <list>
#include <string>
#include <algorithm>

using namespace std;

string &trans(string &s)
{
    for (int p = 0; p < s.size(); p++) {
        if (s[p] >= 'A' && s[p] <= 'Z')
            s[p] -= ('A' - 'a');
        else if (s[p] == ',' || s[p] == '.')
            s.erase(p, 1);
    }
    return s;
}

int main(int argc, char *argv[])
{
    ifstream in(argv[1]);
    if (!in) {
        cout << "打开输入文件失败！" << endl;
        exit(1);
    }

```

```

map<string, list<int>> word_lineno; // 单词到行号的映射
string line;
string word;
int lineno = 0;
while (getline(in, line)) { // 读取一行
    lineno++; // 行号递增
    istringstream l_in(line); // 构造字符串流，读取单词
    while (l_in >> word) {
        trans(word);
        word_lineno[word].push_back(lineno); // 添加行号
    }
}
for (const auto &w : word_lineno) { // 打印单词行号
    cout << w.first << "所在行: ";
    for (const auto &i : w.second)
        cout << i << " ";
    cout << endl;
}
return 0;
}

```

**练习 11.10：**可以定义一个 `vector<int>::iterator` 到 `int` 的 `map` 吗?  
`list<int>::iterator` 到 `int` 的 `map` 呢？对于两种情况，如果不能，解释为什么。

#### 【出题思路】

理解关联容器对关键字类型的要求。

#### 【解答】

由于有序容器要求关键字类型必须支持比较操作`<`，因此

```
map<vector<int>::iterator, int> m1;
```

是可以的，因为 `vector` 的迭代器支持比较操作。而

```
map<list<int>::iterator, int> m2;
```

是不行的，因为 `list` 的元素不是连续存储，其迭代器不支持比较操作。

**练习 11.11：**不使用 `decltype` 重新定义 `bookstore`。

#### 【出题思路】

本题练习函数指针类型的定义。

#### 【解答】

首先用 `typedef` 定义与 `compareIsbn` 相容的函数指针类型，然后用此类型声明 `multiset` 即可。

```
typedef bool (*pf)(const Sales_data &, const Sales_data &);
multiset<Sales_data, pf> bookstore(compareIsbn);
```

**练习 11.12:** 编写程序，读入 string 和 int 的序列，将每个 string 和 int 存入一个 pair 中，pair 保存在一个 vector 中。

### 【出题思路】

本题练习 pair 的使用。

### 【解答】

```
#include <iostream>
#include <fstream>
#include <utility>
#include <vector>
#include <string>
#include <algorithm>

using namespace std;

int main(int argc, char *argv[])
{
    ifstream in(argv[1]);
    if (!in) {
        cout << "打开输入文件失败！" << endl;
        exit(1);
    }

    vector<pair<string, int>> data; // pair 的 vector
    string s;
    int v;
    while (in >> s && in >> v) // 读取一个字符串和一个整数
        data.push_back(pair<string, int>(s, v));

    for (const auto &d : data) // 打印单词行号
        cout << d.first << " " << d.second << endl;
}


```

### 【其他解题思路】

我们还可以使用更简洁的列表初始化方式，将 while 的循环体改为

```
    data.push_back({s, v});
```

即可。

还可使用 make\_pair：

```
    data.push_back(make_pair(s, v));
```

**练习 11.13:** 在上一题的程序中，至少有三种创建 pair 的方法。编写此程序的三个版本，分别采用不同的方法创建 pair。解释你认为哪种形式最易于编写和理解，为什么？

### 【出题思路】

熟悉 pair 的不同初始化方式。

### 【解答】

显然，列表初始化方式最为简洁易懂。

**练习 11.14：**扩展你在 11.2.1 节练习（第 378 页）中编写的孩子姓到名的 map，添加一个 pair 的 vector，保存孩子的名字和生日。

### 【出题思路】

本题练习稍复杂的 pair 和关联容器的结合使用。

### 【解答】

在本题中，我们将家庭的姓映射到孩子信息的列表，而不是简单的孩子名字的列表。因此，将在 vector 中的元素类型声明为 pair<string, string>，两个 string 分别表示孩子的名字和生日。在添加孩子信息时，用列表初始化创建名字和生日的 pair，添加到 vector 中即可。

```
#include <iostream>
#include <map>
#include <utility>
#include <string>
#include <algorithm>

using namespace std;

void add_family(map<string, vector<pair<string, string>>> &families,
                const string &family)
{
    families[family];
}

void add_child(map<string, vector<pair<string, string>>> &families,
               const string &family, const string &child, const string
&birthday)
{
    families[family].push_back({child, birthday});
}

int main(int argc, char *argv[])
{
    map<string, vector<pair<string, string>>> families;

    add_family(families, "张");
    add_child(families, "张", "强", "1970-1-1");
    add_child(families, "张", "刚", "1980-1-1");
    add_child(families, "王", "五", "1990-1-1");
    add_family(families, "王");

    for (auto f : families) {
        cout << f.first << "家的孩子：";
        for (auto c : f.second)
            cout << c.first << "(生日" << c.second << "), ";
        cout << endl;
    }

    return 0;
}
```

**练习 11.15:** 对一个 int 到 vector<int> 的 map，其 mapped\_type、key\_type 和 value\_type 分别是什么？

**【出题思路】**

理解关联容器的类型别名。

**【解答】**

mapped\_type 是 vector<int>。

key\_type 是 int。

value\_type 是 pair<const int, vector<int>>。

**练习 11.16:** 使用一个 map 迭代器编写一个表达式，将一个值赋予一个元素。

**【出题思路】**

理解 map 的迭代器解引用的类型。

**【解答】**

解引用关联容器的迭代器，得到的是 value\_type 的值的引用。因此对 map 而言，得到的是一个 pair 类型的引用，其 first 成员保存 const 的关键字，second 成员保存值。因此，通过迭代器只能修改值，而不能改变关键字。

```
map<int, int> m;
auto it = m.begin();
it->second = 0;
```

**练习 11.17:** 假定 c 是一个 string 的 multiset，v 是一个 string 的 vector，解释下面的调用。指出每个调用是否合法：

```
copy(v.begin(), v.end(), inserter(c, c.end()));
copy(v.begin(), v.end(), back_inserter(c));
copy(c.begin(), c.end(), inserter(v, v.end()));
copy(c.begin(), c.end(), back_inserter(v));
```

**【出题思路】**

理解 set 的迭代器的特点。

**【解答】**

set 的迭代器是 const 的，因此只允许访问 set 中的元素，而不能改变 set。与 map 一样，set 的关键字也是 const，因此也不能通过迭代器来改变 set 中元素的值。

因此，前两个调用试图将 vector 中的元素复制到 set 中，是非法的。

而后两个调用将 set 中的元素复制到 vector 中，是合法的。

**练习 11.18:** 写出第 382 页循环中 map\_it 的类型，不要使用 auto 或 decltype。

**【出题思路】**

理解 map 的迭代器。

**【解答】**

```
pair<const string, size_t>::iterator
```

**练习 11.19:** 定义一个变量，通过对 11.2.2 节（第 378 页）中的名为 bookstore 的 multiset 调用 begin() 来初始化这个变量。写出变量的类型，不要使用 auto 或 decltype。

**【出题思路】**

本题继续练习关联容器的迭代器。

**【解答】**

```
typedef bool (*pf)(const Sales_data &, const Sales_data &);  
multiset<Sales_data, pf> bookstore(compareIsbn);  
...  
pair<const Sales_data, pf>::iterator it = bookstore.begin();
```

**练习 11.20:** 重写 11.1 节练习（第 376 页）的单词计数程序，使用 insert 替代下标操作。你认为哪个程序更容易编写和阅读？解释原因。

**【出题思路】**

熟悉关联容器不同的插入方式。

**【解答】**

使用 insert 操作的方式是：构造一个 pair(单词,1)，用 insert 将其插入容器，返回一个 pair。若单词已存在，则返回 pair 的 second 成员为 false，表示插入失败，程序员还需通过返回 pair 的 first 成员（迭代器）递增已有单词的计数器。判断单词是否已存在，并进行相应操作的工作完全是由程序员负责的。

使用下标操作的方式是：以单词作为下标获取元素值，若单词已存在，则提取出已有元素的值；否则，下标操作将 pair(单词,0) 插入容器，提取出新元素的值。单词是否已存在的相关处理完全是由下标操作处理的，程序员不必关心，直接访问提取出的值就行了。

显然，对于单词计数问题来说，下标操作更简洁易读。

```
#include <iostream>  
#include <fstream>  
#include <map>  
#include <string>  
#include <algorithm>  
  
using namespace std;  
  
int main(int argc, char *argv[]){  
    ifstream in(argv[1]);  
    if (!in) {  
        cout << "打开输入文件失败！" << endl;  
        exit(1);  
    }
```

```

    }

map<string, size_t> word_count;           // string 到 count 的映射
string word;
while (in >> word) {
    auto ret = word_count.insert({word, 1}); // 插入单词，次数为 1
    if (!ret.second)                      // 插入失败，单词已存在
        ++ret.first->second;            // 已有单词的出现次数加 1
}

for (const auto &w : word_count) // 对 map 中的每个元素
    // 打印结果
    cout << w.first << "出现了" << w.second << "次" << endl;

return 0;
}

```

**练习 11.21:** 假定 `word_count` 是一个 `string` 到 `size_t` 的 `map`, `word` 是一个 `string`, 解释下面循环的作用:

```

while (cin >> word)
    ++word_count.insert({word, 0}).first->second;

```

#### 【出题思路】

继续熟悉关联容器的 `insert` 操作。

#### 【解答】

循环不断从标准输入读入单词（字符串），直至遇到文件结束或错误。

每读入一个单词，构造 `pair {word, 0}`，通过 `insert` 操作插入到 `word_count` 中。`insert` 返回一个 `pair`，其 `first` 成员是一个迭代器。若单词（关键字）已存在于容器中，它指向已有元素；否则，它指向新插入的元素。

因此，`.first` 会得到这个迭代器，指向 `word` 对应的元素。继续使用 `->second`，可获得元素的值的引用，即单词的计数。若单词是新的，则其值为 0，若已存在，则值为之前出现的次数。对其进行递增操作，即完成将出现次数加 1。

用这种方法，上一题可稍微简单些。

**练习 11.22:** 给定一个 `map<string, vector<int>>`，对此容器的插入一个元素的 `insert` 版本，写出其参数类型和返回类型。

#### 【出题思路】

继续熟悉关联容器的 `insert` 操作。

#### 【解答】

参数类型是一个 `pair`，`first` 成员的类型是 `map` 的关键字类型 `string`，`second` 成员的类型是 `map` 的值类型 `vector<int>`:

```

pair<string, vector<int>>

```

返回类型也是一个 pair，first 成员的类型是 map 的迭代器，second 成员的类型是布尔型：

```
pair<map<string, vector<int>>::iterator, bool>
```

**练习 11.23：**11.2.1 节练习（第 378 页）中的 map 以孩子的姓为关键字，保存他们的名的 vector，用 multimap 重写此 map。

### 【出题思路】

本题练习允许重复关键字的关联容器的 insert 操作。

### 【解答】

由于允许重复关键字，已经不需要 vector 保存同一家的孩子的名字的列表，直接保存每个孩子的(姓,名)pair 即可。容器类型变为 multimap<string, string>。

也不再需要 add\_family 添加家庭，只保留 add\_child 直接用 insert 操作添加孩子即可。

```
#include <iostream>
#include <map>
#include <string>
#include <algorithm>

using namespace std;

void add_child(multimap<string, string> &families, const string &family,
               const string &child)
{
    families.insert({family, child});
}

int main(int argc, char *argv[])
{
    multimap<string, string> families;

    add_child(families, "张", "强");
    add_child(families, "张", "刚");
    add_child(families, "王", "五");

    for (auto f : families)
        cout << f.first << "家的孩子：" << f.second << endl;

    return 0;
}
```

**练习 11.24：**下面的程序完成什么功能？

```
map<int, int> m;
m[0] = 1;
```

**【出题思路】**

本题继续熟悉 map 的下标操作。

**【解答】**

若 m 中已有关键字 0，下标操作提取出其值，赋值语句将值置为 1。

否则，下标操作会创建一个 pair (0, 0)，即关键字为 0，值为 0（值初始化），将其插入到 m 中，然后提取其值，赋值语句将值置为 1。

**练习 11.25：**对比下面程序与上一题程序

```
vector<int> v;
v[0] = 1;
```

**【出题思路】**

理解顺序容器和关联容器下标操作的不同。

**【解答】**

对于 m，“0”表示“关键字 0”。而对于 v，“0”表示“位置 0”。

若 v 中已有不少于一个元素，即，存在“位置 0”元素，则下标操作提取出此位置的元素（左值），赋值操作将其置为 1。而 map 的元素是 pair 类型，下标提取的不是元素，而是元素的第二个成员，即元素的值。

如 v 尚为空，则下标提取出的是一个非法左值（下标操作不做范围检查），向其赋值可能导致系统崩溃等严重后果。

**练习 11.26：**可以用什么类型来对一个 map 进行下标操作？下标运算符返回的类型是什么？请给出一个具体例子——即，定义一个 map，然后写出一个可以用来对 map 进行下标操作的类型以及下标运算符将会返回的类型。

**【出题思路】**

理解 map 的下标操作所涉及的各种类型。

**【解答】**

对 map 进行下标操作，应使用其 key\_type，即关键字的类型。

而下标操作返回的类型是 mapped\_type，即关键字关联的值的类型。

示例如下：

map 类型：map<string, int>

来进行下标操作的类型：string

下标操作返回的类型：int

**练习 11.27：**对于什么问题你会使用 count 来解决？什么时候你又会选择 find 呢？

**【出题思路】**

理解关联容器上不同算法的区别。

### 【解答】

`find` 查找关键字在容器中出现的位置，而 `count` 则还会统计关键字出现的次数。

因此，当我们希望知道（允许重复关键字的）容器中有多少元素的关键字与给定关键字相同时，使用 `count`。

当我们只关心关键字是否在容器中时，使用 `find` 就足够了。特别是，对于不允许重复关键字的关联容器，`find` 和 `count` 的效果没有什么区别，使用 `find` 就可以了。或者，当我们需要获取具有给定关键字的元素（不只是统计个数）时，也需要使用 `find`。

`find` 和下标操作有一个重要区别，当给定关键字不在容器中时，下标操作会插入一个具有该关键字的元素。因此，当我们想检查给定关键字是否存在时，应该用 `find` 而不是下标操作。

**练习 11.28：**对一个 `string` 到 `int` 的 `vector` 的 `map`，定义并初始化一个变量来保存在其上调用 `find` 所返回的结果。

### 【出题思路】

理解 `map` 上的 `find`。

### 【解答】

`find` 返回一个迭代器，指向具有给定关键字的元素（若不存在则返回尾后迭代器），因此其返回类型是容器的迭代器。

```
// map 类型
map<string, vector<int>> m;
// 保存 find 返回结果的变量
map<string, vector<int>>::iterator iter;
```

**练习 11.29：**如果给定的关键字不在容器中，`upper_bound`、`lower_bound` 和 `equal_range` 分别会返回什么？

### 【出题思路】

熟悉适合 `multimap` 和 `multiset` 的基于迭代器的关键字查找方法。

### 【解答】

`lower_bound` 返回第一个具有给定关键字的元素，`upper_bound` 则返回最后一个具有给定关键字的元素之后的位置。即，这两个迭代器构成包含所有具有给定关键字的元素的范围。若给定关键字不在容器中，两个操作显然应构成一个空范围，它们返回相当的迭代器，指出关键字的正确插入位置——不影响关键字的排序。如果给定关键字比容器中所有关键字都大，则此位置是容器的尾后位置 `end`。

`equal_range` 返回一个 `pair`，其 `first` 成员等价于 `lower_bound` 返回的

迭代器，second 成员等价于 upper\_bound 返回的迭代器。因此，若给定关键字不在容器中，first 和 second 都指向关键字的正确插入位置，两个迭代器构成一个空范围。

**练习 11.30：**对于本节最后一个程序中的输出表达式，解释运算对象 pos.first->second 的含义。

**【出题思路】**

熟悉 equal\_range 的使用。

**【解答】**

equal\_range 返回一个 pair，其 first 成员与 lower\_bound 的返回结果相同，即指向容器中第一个具有给定关键字的元素。因此，对其解引用会得到一个 value\_type 对象，即一个 pair，其 first 为元素的关键字，即给定关键字，而 second 为关键字关联的值。在本例中，关键字为作者，关联的值为著作的题目。因此 pos.first->second 即获得给定作者的第一部著作的题目。

**练习 11.31：**编写程序，定义一个作者及其作品的 multimap。使用 find 在 multimap 中查找一个元素并用 erase 删除它。确保你的程序在元素不在 map 中时也能正常运行。

**【出题思路】**

练习 multimap 的插入、查找和删除。

**【解答】**

将数据插入 multimap，需使用 insert 操作。

在 multimap 中查找具有给定关键字的元素，有几种方法：使用 find 只能查找第一个具有给定关键字的元素，要找到所有具有给定关键字的元素，需编写循环；lower\_bound 和 upper\_bound 配合使用，可找到具有给定关键字的元素的范围；equal\_range 最为简单，一次即可获得要查找的元素范围。

将找到的范围传递给 erase，即可删除指定作者的所有著作。

为了解决元素不在 multimap 中的情况，首先检查 equal\_range 返回的两个迭代器，若相等(空范围)，则什么也不做。范围不为空时，才将迭代器传递给 erase，删除所有元素。

```
#include <iostream>
#include <string>
#include <map>
#include <algorithm>

using namespace std;

void remove_author(multimap<string, string> &books,
                    const string &author)
```

```

{
    auto pos = books.equal_range(author); // 查找给定作者范围
    if (pos.first == pos.second) // 空范围, 没有该作者
        cout << "没有" << author << "这个作者" << endl << endl;
    else
        books.erase(pos.first, pos.second); // 删除该作者所有著作
}

void print_books(multimap<string, string> &books)
{
    cout << "当前书目包括: " << endl;
    for (auto &book : books) // 遍历所有书籍, 打印之
        cout << book.first << ", 《" << book.second << "》" << endl;
    cout << endl;
}

int main(int argc, char *argv[])
{
    multimap<string, string> books;
    books.insert({"Barth, John", "Sot-Weed Factor"});
    books.insert({"Barth, John", "Lost in the Funhouse"});
    books.insert({"金庸", "射雕英雄传"});
    books.insert({"金庸", "天龙八部"});

    print_books(books);

    remove_author(books, "张三");

    remove_author(books, "Barth, John");
    print_books(books);

    return 0;
}

```

### 【其他解题思路】

使用 `find` 或 `lower_bound+upper_bound`, 也可实现本题目标, 但复杂一些。

**练习 11.32:** 使用上一题定义的 `multimap` 编写一个程序, 按字典序打印作者列表和他们的作品。

### 【出题思路】

本题要求理解 `multimap` 数据结构中关键字的顺序, 以及利用它来实现关键字的有序输出。

### 【解答】

`multimap` 的数据结构是红黑树, 它维护了元素的关键字的默认序。例如, 对字符串关键字 (作者), 红黑树会维护它们的字典序。当我们遍历 `multimap` (如遍历 `[begin(), end()]`, 或更简单地使用范围 `for`) 时, 就是按关键字的字典序来

访问元素。

因此，上一题的 `print_books` 实际上已经实现了按字典序打印作者列表和他们的作品。

但是，当我们要求的不是关键字的默认序（运算符`<`定义的顺序）时，就要复杂一些。由于 `sort` 算法要求给定的两个迭代器是随机迭代器，关联容器的迭代器不符合这一要求，所以不能直接对其使用 `sort` 算法。其实这不难理解，关联容器的根本特征就是维护了关键字的默认序，从而实现了按关键字的插入、删除和查找。是不可能通过 `sort` 使其内部元素呈现出另外一种顺序的。只有本身不关心元素值的顺序容器，才可能随意安排元素顺序（位置）。我们可以在定义 `multimap` 时使用自己定义的比较操作所定义的关键字的序，而不是使用`<`定义的序，但这只是令 `multimap` 以另外一种序来维护关键字，仍然不可能在使用 `multimap` 的过程中来改变关键字顺序。为此，我们只能将 `multimap` 中的元素拷贝到一个顺序容器（如 `vector`）中，对顺序容器执行 `sort` 算法，来获得关键字的其他序。

### 练习 11.33：实现你自己版本的单词转换程序。

#### 【出题思路】

关联容器的综合练习。

#### 【解答】

本书配套网站提供了书中的全部源代码，其中就有单词转换程序。尝试编译、运行它即可。

结合本小节内容阅读、理解此程序，若仍不能理解，可利用集成开发环境的跟踪功能跟踪程序运行，观察变量值的变化，来帮助你理解程序。

### 练习 11.34：如果你将 `transform` 函数中的 `find` 替换为下标运算符，会发生什么情况？

#### 【出题思路】

继续理解 `find` 和下标操作的区别。

#### 【解答】

如前所述，`find` 仅查找给定关键字在容器中是否出现，若容器中不存在给定关键字，它返回尾后迭代器。当关键字存在时，下标运算符的行为与 `find` 类似，但当关键字不存在时，它会构造一个 `pair`（进行值初始化），将其插入到容器中。对于单词转换程序，这会将不存在的内容插入到输出文本中，这显然不是我们所期望的。

### 练习 11.35：在 `buildMap` 中，如果进行如下改写，会有什么效果？

```
trans_map[key] = value.substr(1);
```

改为 `trans_map.insert({key, value.substr(1)})`

**【出题思路】**

继续理解 `insert` 操作和下标操作的区别。

**【解答】**

当 `map` 中没有给定关键字时, `insert` 操作与下标操作+赋值操作的效果类似, 都是将关键字和值的 `pair` 添加到 `map` 中。

但当 `map` 中已有给定关键字, 也就是新的转换规则与一条已有规则要转换同一个单词时, 两者的行为是不同的。下标操作会获得具有该关键字的元素(也就是已有规则)的值, 并将新读入的值赋予它, 也就是用新读入的规则覆盖了容器中的已有规则。但 `insert` 操作遇到关键字已存在的情况, 则不会改变容器内容, 而是返回一个值指出插入失败。因此, 当规则文件中存在多条规则转换相同单词时, 下标+赋值的版本最终会用最后一条规则进行文本转换, 而 `insert` 版本则会用第一条规则进行文本转换。

**练习 11.36:** 我们的程序并没有检查输入文件的合法性。特别是, 它假定转换规则文件中的规则都是有意义的。如果文件中的某一行包含一个关键字、一个空格, 然后就结束了, 会发生什么? 预测程序的行为并进行验证, 再与你的程序进行比较。

**【出题思路】**

本题练习字符串处理的技巧。

**【解答】**

此题有误, 书中程序已经处理了这种情况。

在 `buildMap` 函数中, 当循环中读入要转换的单词和转换的内容后, 会检查是否存在转换的内容 (`value.size() > 1`), 若不存在, 则抛出一个异常。

当然, 程序只处理了这一种错误。你可以思考还有哪些错误, 编写程序完成处理。

**练习 11.37:** 一个无序容器与其有序版本相比有何优势? 有序版本有何优势?

**【出题思路】**

理解无序关联容器与有序版本的差异。

**【解答】**

无序版本通常性能更好, 使用也更为简单。有序版本的优势是维护了关键字的序。

当元素的关键字类型没有明显的序关系, 或是维护元素的序代价非常高时, 无序容器非常有用。

但当应用要求必须维护元素的序时, 有序版本就是唯一的选择。

**练习 11.38：**用 `unordered_map` 重写单词计数程序（参见 11.1 节，第 375 页）和单词转换程序（参见 11.3.6 节，第 391 页）。

### 【出题思路】

本题练习使用无序关联容器。

### 【解答】

对单词计数程序仅有的两处修改是将包含的头文件 `map` 改为 `unordered_map`，以及将 `word_count` 的类型由 `map` 改为 `unordered_map`。

尝试编译、运行此程序，你会发现，由于无序容器不维护元素的序，程序的输出结果与第 3 题的输出结果的顺序是不同的。

```
#include <iostream>
#include <fstream>
#include <unordered_map>
#include <string>
#include <algorithm>

using namespace std;

int main(int argc, char *argv[])
{
    ifstream in(argv[1]);
    if (!in) {
        cout << "打开输入文件失败！" << endl;
        exit(1);
    }

    unordered_map<string, size_t> word_count; // string 到 count 的映射
    string word;
    while (in >> word)                                // 这个单词的出现次数加 1
        ++word_count[word];

    for (const auto &w : word_count)                  // 对 map 中的每个元素
        // 打印结果
        cout << w.first << "出现了" << w.second << "次" << endl;

    return 0;
}
```

单词转换程序的修改类似。由于程序中不再有元素内容的顺序输出，因此输出结果与有序版本没有什么不同。

```
#include <unordered_map>
#include <vector>
#include <iostream>
#include <fstream>
#include <string>
#include <stdexcept>
#include <sstream>

using std::unordered_map; using std::string; using std::vector;
using std::ifstream; using std::cout; using std::endl;
using std::getline;
```

```

using std::runtime_error; using std::istringstream;

unordered_map<string, string> buildMap(ifstream &map_file)
{
    unordered_map<string, string> trans_map; // 保存转换规则
    string key; // 要转换的单词
    string value; // 用来替换的内容
    // 读取第一个单词存入 key, 这一行的剩余内容存入 value
    while (map_file >> key && getline(map_file, value))
        if (value.size() > 1) // 检查是否确实存在转换规则
            trans_map[key] = value.substr(1); // 跳过前导空白
        else
            throw runtime_error("no rule for " + key);
    return trans_map;
}

const string &
transform(const string &s, const unordered_map<string, string> &m)
{
    // 完成具体转换工作, 这个函数是程序的核心
    auto map_it = m.find(s);
    // 如果这个单词在转换映射表中
    if (map_it != m.cend())
        return map_it->second; // 用映射表指定内容替换单词
    else
        return s; // 否则原样返回单词
}

// 第一个参数为转换规则文件
// 第二个参数是要转换的文本文件
void word_transform(ifstream &map_file, ifstream &input)
{
    auto trans_map = buildMap(map_file); // 保存转换规则

    // 调试用: 映射表创建好后打印它
    cout << "Here is our transformation map: \n\n";
    for (auto entry : trans_map)
        cout << "key: " << entry.first
            << "\tvalue: " << entry.second << endl;
    cout << "\n\n";

    // 对给定文本进行转换
    string text; // 保存从输入读取的每一行
    while (getline(input, text)) { // 从输入读取一行
        istringstream stream(text); // 读取每个单词
        string word;
        bool firstword = true; // 控制是否打印空格
        while (stream >> word) {
            if (firstword)
                firstword = false;
            else

```

```
        cout << " "; // 在单词间打印空格
        // 转换结果可能是另一个字符串也可能是原单词
        cout << transform(word, trans_map); // 打印结果
    }
    cout << endl; // 当前行转换完毕，打印回车
}
}

int main(int argc, char **argv)
{
    // 打开两个文件并检查是否打开成功
    if (argc != 3)
        throw runtime_error("wrong number of arguments");

    ifstream map_file(argv[1]); // 打开转换规则文件
    if (!map_file) // 检查是否打开成功
        throw runtime_error("no transformation file");

    ifstream input(argv[2]); // 打开要转换的文件
    if (!input) // 检查是否打开成功
        throw runtime_error("no input file");

    word_transform(map_file, input);

    return 0; // 退出主函数时文件会自动关闭
}
```

# 第 12 章

## 动态内存

### 导读

本章介绍了智能指针与动态内存管理，包括：

- 智能指针的基本概念，特别是用它来管理动态内存的好处。
- 用 allocator 管理动态数组。
- 用文本查询这样一个较大的例子来展示动态内存管理。

本章练习的重点是让读者熟悉智能指针的使用，包括使用 `shared_ptr` 和 `unique_ptr` 管理动态内存时要注意的一些问题；用 allocator 管理动态数组；以及基于文本查询例子的一些较大的练习。

**练习 12.1：**在此代码的结尾，`b1` 和 `b2` 各包含多少个元素？

```
StrBlob b1;
{
    StrBlob b2 = {"a", "an", "the"};
    b1 = b2;
    b2.push_back("about");
}
```

#### 【出题思路】

理解智能指针的基本特点。

#### 【解答】

由于 `StrBlob` 的 `data` 成员是一个指向 `string` 的 `vector` 的 `shared_ptr`，因此 `StrBlob` 的赋值不会拷贝 `vector` 的内容，而是多个 `StrBlob` 对象共享同一个（创建于动态内存空间上）`vector` 对象。

代码第 3 行创建 `b2` 时提供了 3 个 `string` 的列表，因此会创建一个包含 3 个

string 的 vector 对象，并创建一个 shared\_ptr 指向此对象（引用计数为1）。

第4行将 b2 赋予 b1 时，创建一个 shared\_ptr 也指向刚才创建的 vector 对象，引用计数变为2。

因此，第4行向 b2 添加一个 string 时，会向两个 StrBlob 共享的 vector 中添加此 string。最终，在代码结尾，b1 和 b2 均包含4个 string。

**练习 12.2：**编写你自己的 StrBlob 类，包含 const 版本的 front 和 back。

### 【出题思路】

本题练习智能指针的简单使用。

### 【解答】

参考书中代码，并补充 front 和 back 对 const 的重载，即可完成自己的 StrBlob 类：

```
#ifndef MY_STRBLOB_H
#define MY_STRBLOB_H
#include <vector>
#include <string>
#include <initializer_list>
#include <memory>
#include <stdexcept>

using namespace std;

class StrBlob {
public:
    typedef vector<string>::size_type size_type;
    StrBlob();
    StrBlob(initializer_list<string> il);
    size_type size() const { return data->size(); }
    bool empty() const { return data->empty(); }
    // 添加和删除元素
    void push_back(const string &t) {data->push_back(t);}
    void pop_back();
    // 元素访问
    string& front();
    const string& front() const;
    string& back();
    const string& back() const ;
private:
    shared_ptr<std::vector<std::string>> data;
    // 如果 data[i] 不合法，抛出一个异常
    void check(size_type i, const std::string &msg) const;
};

StrBlob::StrBlob(): data(make_shared<vector<string>>()) { }
StrBlob::StrBlob(initializer_list<string> il):
    data(make_shared<vector<string>>(il)) { }

void StrBlob::check(size_type i, const string &msg) const
```

```

{
    if (i >= data->size())
        throw out_of_range(msg);
}

string& StrBlob::front()
{
    // 如果 vector 为空, check 会抛出一个异常
    check(0, "front on empty StrBlob");
    return data->front();
}

// const 版本 front
const string& StrBlob::front() const
{
    check(0, "front on empty StrBlob");
    return data->front();
}

string& StrBlob::back()
{
    check(0, "back on empty StrBlob");
    return data->back();
}

// const 版本 back
const string& StrBlob::back() const
{
    check(0, "back on empty StrBlob");
    return data->back();
}

void StrBlob::pop_back()
{
    check(0, "pop_back on empty StrBlob");
    data->pop_back();
}

#endif

```

再编写简单的 StrBlob 使用程序，测试类的正确性：

```

#include <iostream>

using namespace std;

#include "my_StrBlob.h"

int main(int argc, char **argv)
{
    StrBlob b1;
    {
        StrBlob b2 = { "a", "an", "the" };
        b1 = b2
        b2.push_back("about");
    }
}
```

```

        cout << b2.size() << endl;
    }
    cout << b1.size() << endl;
    cout << b1.front() << " " << b1.back() << endl;

    const StrBlob b3 = b1;
    cout << b3.front() << " " << b3.back() << endl;

    return 0;
}

```

**练习 12.3:** StrBlob 需要 const 版本的 push\_back 和 pop\_back 吗？如果需要，添加进去。否则，解释为什么不需要。

**【出题思路】**

理解 const 版本和非 const 版本的差别。

**【解答】**

push\_back 和 pop\_back 的语义分别是向 StrBlob 对象共享的 vector 对象添加元素和从其删除元素。因此，我们不应为其重载 const 版本，因为常量 StrBlob 对象是不应被允许修改共享 vector 对象内容的。

**练习 12.4:** 在我们的 check 函数中，没有检查 i 是否大于 0。为什么可以忽略这个检查？

**【出题思路】**

理解私有成员函数和公有成员函数的差别。

**【解答】**

我们将 check 定义为私有成员函数，亦即，它只会被 StrBlob 的成员函数调用，而不会被用户程序所调用。因此，我们可以很容易地保证传递给它的 i 的值符合要求，而不必进行检查。

**练习 12.5:** 我们未编写接受一个 initializer\_list explicit (参见 7.5.4 节，第 264 页) 参数的构造函数。讨论这个设计策略的优点和缺点。

**【出题思路】**

复习隐式类类型转换和显式转换的区别。

**【解答】**

未编写接受一个初始化列表参数的显式构造函数，意味着可以进行列表向 StrBlob 的隐式类型转换，亦即在需要 StrBlob 的地方（如函数的参数），可以使用户列表进行替代。而且，可以进行拷贝形式的初始化（如赋值）。这令程序编写更为简单方便。

但这种隐式转换并不总是好的。例如，列表中可能并非都是合法的值。再如，

对于接受 StrBlob 的函数，传递给它一个列表，会创建一个临时的 StrBlob 对象，用列表对其初始化，然后将其传递给函数，当函数完成后，此对象将被丢弃，再也无法访问了。对于这些情况，我们可以定义显式的构造函数，禁止隐式类类型转换。

**练习 12.6：**编写函数，返回一个动态分配的 int 的 vector。将此 vector 传递给另一个函数，这个函数读取标准输入，将读入的值保存在 vector 元素中。再将 vector 传递给另一个函数，打印读入的值。记得在恰当的时刻 delete vector。

### 【出题思路】

本题练习用 new 和 delete 直接管理内存。

### 【解答】

直接内存管理的关键是谁分配了内存谁就要记得释放。在此程序中，主函数调用分配函数在动态内存空间中创建 int 的 vector，因此在读入数据、打印数据之后，主函数应负责释放 vector 对象。

```
#include <iostream>
#include <vector>

using namespace std;

vector<int> *new_vector(void)
{
    return new (nothrow) vector<int>;
}

void read_ints(vector<int> *pv)
{
    int v;

    while (cin >> v)
        pv->push_back(v);
}

void print_ints(vector<int> *pv)
{
    for (const auto &v : *pv)
        cout << v << " ";
    cout << endl;
}

int main(int argc, char **argv)
{
    vector<int> *pv = new_vector();
    if (!pv) {
        cout << "内存不足！" << endl;
        return -1;
    }

    read_ints(pv);
}
```

```

    print_ints(pv);

    delete pv;
    pv = nullptr;

    return 0;
}

```

**练习 12.7：**重做上一题，这次使用 `shared_ptr` 而不是内置指针。

### 【出题思路】

本题练习用智能指针管理内存。

### 【解答】

与上一题相比，程序差别不大，主要是将 `vector<int> *类型` 变为 `shared_ptr<vector<int>>` 类型，空间分配不再用 `new` 而改用 `make_shared`，在主函数末尾不再需要主动释放内存。最后一点的意义对这个小程序还不明显，但对于大程序非常重要，它省去了程序员释放内存的工作，可以有效避免内存泄漏问题。

```

#include <iostream>
#include <vector>
#include <memory>

using namespace std;

shared_ptr<vector<int>> new_vector(void)
{
    return make_shared<vector<int>>();
}

void read_ints(shared_ptr<vector<int>> spv)
{
    int v;

    while (cin >> v)
        spv->push_back(v);
}

void print_ints(shared_ptr<vector<int>> spv)
{
    for (const auto &v : *spv)
        cout << v << " ";
    cout << endl;
}

int main(int argc, char **argv)
{
    auto spv = new_vector();

    read_ints(spv);

    print_ints(spv);
}

```

```

    print_ints(spv);

    return 0;
}

```

**练习 12.8:** 下面的函数是否有错误？如果有，解释错误原因。

```

bool b() {
    int* p = new int;
    // ...
    return p;
}

```

#### 【出题思路】

理解用 `new` 分配内存成功和失败的差别，以及复习类型转换。

#### 【解答】

从程序片段看，可以猜测程序员的意图是通过 `new` 返回的指针值来区分内存分配成功或失败——成功返回一个合法指针，转换为整型是一个非零值，可转换为 `bool` 值 `true`；分配失败，`p` 得到 `nullptr`，其整型值是 0，可转换为 `bool` 值 `false`。

但普通 `new` 调用在分配失败时抛出一个异常 `bad_alloc`，而不是返回 `nullptr`，因此程序不能达到预想的目的。

可将 `new int` 改为 `new (nothrow) int` 来令 `new` 在分配失败时不抛出异常，而是返回 `nullptr`。但这仍然不是一个好方法，应该通过捕获异常或是判断返回的指针来返回 `true` 或 `false`，而不是依赖类型转换。

**练习 12.9:** 解释下面代码执行的结果：

```

int *q = new int(42), *r = new int(100);
r = q;
auto q2 = make_shared<int>(42), r2 = make_shared<int>(100);
r2 = q2;

```

#### 【出题思路】

理解直接管理内存和智能指针的差别。

#### 【解答】

这段代码非常好地展示了智能指针在管理内存上的优点。

对于普通指针部分，首先分配了两个 `int` 型对象，指针分别保存在 `p` 和 `r` 中。接下来，将指针 `q` 的值赋予了 `r`，这带来了两个非常严重的内存管理问题：

1. 首先是一个直接的内存泄漏问题，`r` 和 `q` 一样都指向 42 的内存地址，而 `r` 中原来保存的地址——100 的内存再无指针管理，变成“孤儿内存”，从而造成内存泄漏。
2. 其次是一个“空悬指针”问题。由于 `r` 和 `q` 指向同一个动态对象，如果程序编写不当，很容易产生释放了其中一个指针，而继续使用另一个指针的问题。继续使用的指针指向的是一块已经释放的内存，是一个空悬指针，继续读写它指向的内

存可能导致程序崩溃甚至系统崩溃的严重问题。

而 `shared_ptr` 则可很好地解决这些问题。首先，分配了两个共享的对象，分别由共享指针 `p2` 和 `q2` 指向，因此它们的引用计数均为 1。接下来，将 `q2` 赋予 `r2`。赋值操作会将 `q2` 指向的对象地址赋予 `r2`，并将 `r2` 原来指向的对象的引用计数减 1，将 `q2` 指向的对象的引用计数加 1。这样，前者的引用计数变为 0，其占用的内存空间会被释放，不会造成内存泄漏。而后的引用计数变为 2，也不会因为 `r2` 和 `q2` 之一的销毁而释放它的内存空间，因此也不会造成空悬指针的问题。

**练习 12.10：**下面的代码调用了第 413 页中定义的 `process` 函数，解释此调用是否正确。如果不正确，应如何修改？

```
shared_ptr<int> p(new int(42));
process(shared_ptr<int>(p));
```

#### 【出题思路】

理解智能指针的使用。

#### 【解答】

此调用是正确的，利用 `p` 创建一个临时的 `shared_ptr` 赋予 `process` 的参数 `ptr`，`p` 和 `ptr` 都指向相同的 `int` 对象，引用计数被正确地置为 2。`process` 执行完毕后，`ptr` 被销毁，引用计数减 1，这是正确的——只有 `p` 指向它。

**练习 12.11：**如果我们像下面这样调用 `process`，会发生什么？

```
process(shared_ptr<int>(p.get()));
```

#### 【出题思路】

理解智能指针和普通指针不能混用。

#### 【解答】

此调用是错误的。`p.get()` 获得一个普通指针，指向 `p` 所共享的 `int` 对象。利用此指针创建一个 `shared_ptr`，而不是利用 `p` 创建一个 `shared_ptr`，将不会形成正确的动态对象共享。编译器会认为 `p` 和 `ptr` 是使用两个地址（虽然它们相等）创建的两个不相干的 `shared_ptr`，而非共享同一个动态对象。这样，两者的引用计数均为 1。当 `process` 执行完毕后，`ptr` 的引用计数减为 0，所管理的内存地址被释放，而此内存就是 `p` 所管理的。`p` 成为一个管理空悬指针的 `shared_ptr`。

**练习 12.12：**`p` 和 `sp` 的定义如下，对于接下来的对 `process` 的每个调用，如果合法，解释它做了什么，如果不合法，解释错误原因：

```
auto p = new int();
auto sp = make_shared<int>();
(a) process(sp);
(b) process(new int());
(c) process(p);
(d) process(shared_ptr<int>(p));
```

**【出题思路】**

理解智能指针和普通指针、new 混合使用应该注意的问题。

**【解答】**

(a)合法。sp 是一个共享指针，指向一个 int 对象。对 process 的调用会拷贝 sp，传递给 process 的参数 ptr，两者都指向相同的 int 对象，引用计数变为 2。当 process 执行完毕时，ptr 被销毁，引用计数变回 1。

(b)合法。new 创建了一个 int 对象，指向它的指针被用来创建了一个 shared\_ptr，传递给 process 的参数 ptr，引用计数为 1。当 process 执行完毕，ptr 被销毁，引用计数变为 0，临时 int 对象因而被销毁。不存在内存泄漏和空悬指针的问题。

(c)不合法。不能将 int\* 转换为 shared\_ptr<int>。

(d)合法，但是错误的程序。p 是一个指向 int 对象的普通指针，被用来创建一个临时 shared\_ptr，传递给 process 的参数 ptr，引用计数为 1。当 process 执行完毕，ptr 被销毁，引用计数变为 0，int 对象被销毁。p 变为空悬指针。

**练习 12.13：**如果执行下面的代码，会发生什么？

```
auto sp = make_shared<int>();
auto p = sp.get();
delete p;
```

**【出题思路】**

继续理解智能指针和普通指针使用上的问题。

**【解答】**

第二行用 get 获取了 sp 指向的 int 对象的地址，第三行用 delete 释放这个地址。这意味着 sp 的引用计数仍为 1，但其指向的 int 对象已经被释放了。sp 成为类似空悬指针的 shared\_ptr。

**练习 12.14：**编写你自己版本的用 shared\_ptr 管理 connection 的函数。**【出题思路】**

本题练习利用智能指针管理使用资源的类，避免内存泄漏等问题。

**【解答】**

参照本节内容设计函数即可。main 函数分别调用了未使用和使用了 shared\_ptr 的版本，根据输出可以看出，前者未调用 disconnect，而后者调用了。注意观察 f1 的输出中的换行，很明显，disconnect 是在 f1 结束后（最后一条输出换行的语句已经执行完），在销毁 p 时被调用的。

```
#include <iostream>
#include <memory>

using namespace std;
```

```
struct destination {};
struct connection {};

connection connect(destination *pd)
{
    cout << "打开连接" << endl;
    return connection();
}

void disconnect(connection c)
{
    cout << "关闭连接" << endl;
}

// 未使用 shared_ptr 的版本
void f(destination &d)
{
    cout << "直接管理 connect" << endl;
    connection c = connect(&d);
    // 忘记调用 disconnect 关闭连接

    cout << endl;
}

void end_connection(connection *p) { disconnect(*p); }

// 使用 shared_ptr 的版本
void f1(destination &d)
{
    cout << "用 shared_ptr 管理 connect" << endl;
    connection c = connect(&d);

    shared_ptr<connection> p(&c, end_connection);
    // 忘记调用 disconnect 关闭连接

    cout << endl;
}

int main(int argc, char **argv)
{
    destination d;

    f(d);

    f1(d);

    return 0;
}
```

**练习 12.15:** 重写上一题的程序，用 lambda（参见 10.3.2 节，第 346 页）代替 end\_connection 函数。

### 【出题思路】

复习 lambda。

### 【解答】

根据 end\_connection 的定义，lambda 不捕获局部变量，参数为 connection 指针，用该指针指向的对象调用 disconnect 即可：

```
[](connection *p) { disconnect(*p); }

#include <iostream>
#include <memory>

using namespace std;

struct destination {};
struct connection {};

connection connect(destination *pd)
{
    cout << "打开连接" << endl;
    return connection();
}

void disconnect(connection c)
{
    cout << "关闭连接" << endl;
}

// 未使用 shared_ptr 的版本
void f(destination &d)
{
    cout << "直接管理 connect" << endl;
    connection c = connect(&d);
    // 忘记调用 disconnect 关闭连接

    cout << endl;
}

// 使用 shared_ptr 的版本
void f1(destination &d)
{
    cout << "用 shared_ptr 管理 connect" << endl;
    connection c = connect(&d);

    shared_ptr<connection> p(&c,
        [](connection *p) { disconnect(*p); });
    // 忘记调用 disconnect 关闭连接

    cout << endl;
}
```

```

int main(int argc, char **argv)
{
    destination d;

    f(d);

    f1(d);

    return 0;
}

```

**练习 12.16:** 如果你试图拷贝或赋值 unique\_ptr，编译器并不总是能给出易于理解的错误信息。编写包含这种错误的程序，观察编译器如何诊断这种错误。

#### 【出题思路】

深入地理解 unique\_ptr 不能拷贝或赋值的限制。

#### 【解答】

用 tdm-gcc 8.1 编译本节开始的错误拷贝和赋值程序，会给出类似如下的错误信息：

```
error: use of deleted function 'std::unique_ptr<_Tp, _Dp>::unique_ptr(const std::unique_ptr<_Tp, _Dp>&) [with _Tp = std::basic_string<char>; _Dp = std::default_delete<std::basic_string<char> >]'
```

即，程序调用了删除的函数。原因是，标准库为了禁止 unique\_ptr 的拷贝和赋值，将其拷贝构造函数和赋值函数声明为了 delete 的：

```
unique_ptr(const unique_ptr&) = delete;
unique_ptr& operator=(const unique_ptr&) = delete;
```

**练习 12.17:** 下面的 unique\_ptr 声明中，哪些是合法的，哪些可能导致后续的程序错误？解释每个错误的问题在哪里。

```

int ix = 1024, *pi = &ix, *pi2 = new int(2048);
typedef unique_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());

```

#### 【出题思路】

继续熟悉 unique\_ptr 使用上应注意的问题。

#### 【解答】

- (a) 不合法。unique\_ptr 需要用一个指针初始化，无法将 int 转换为指针。
- (b) 合法。可以用一个 int \* 来初始化 IntP，但此程序逻辑上是错误的。它用一个普通 int 变量的地址初始化 p1，p1 销毁时会释放此内存，其行为是未定义的。
- (c) 合法。用一个指向动态分配的对象的指针来初始化 IntP 是正确的。
- (d) 合法。但存在与(b)相同的问题。
- (e) 合法。与(c)类似。

(f) 合法。但用 p2 管理的对象的地址来初始化 p5，造成两个 unique\_ptr 指向相同的内存地址。当其中一个 unique\_ptr 被销毁（或调用 reset 释放对象）时，该内存被释放，另一个 unique\_ptr 变为空悬指针。

**练习 12.18:** shared\_ptr 为什么没有 release 成员？

【出题思路】

理解 unique\_ptr 和 shared\_ptr 的差别。

【解答】

unique\_ptr “独占” 对象的所有权，不能拷贝和赋值。release 操作是用来将对象的所有权转移给另一个 unique\_ptr 的。

而多个 shared\_ptr 可以“共享”对象的所有权。需要共享时，可以简单拷贝和赋值。因此，并不需要 release 这样的操作来转移所有权。

**练习 12.19:** 定义你自己版本的 StrBlobPtr，更新 StrBlob 类，加入恰当的 friend 声明及 begin 和 end 成员。

【出题思路】

熟悉 weak\_ptr 的使用。

【解答】

参考本节内容实现所需程序即可。为了实现对 StrBlob 的 vector 中元素的遍历，还定义了 eq 和 neq 两个函数来比较两个 StrBlobPtr 是否指向相同 vector 的相同位置。

```
my_StrBlob.h:
#ifndef MY_STRBLOB_H
#define MY_STRBLOB_H
#include <vector>
#include <string>
#include <initializer_list>
#include <memory>
#include <stdexcept>

using namespace std;

// 提前声明，StrBlob 中的友类声明所需
class StrBlobPtr;

class StrBlob {
    friend class StrBlobPtr;
public:
    typedef vector<string>::size_type size_type;
    StrBlob();
    StrBlob(initializer_list<string> il);
    size_type size() const { return data->size(); }
    bool empty() const { return data->empty(); }
```

```

// 添加和删除元素
void push_back(const string &t) {data->push_back(t);}
void pop_back();
// 元素访问
string& front();
const string& front() const;
string& back();
const string& back() const;

// 提供给 StrBlobPtr 的接口
StrBlobPtr begin(); // 定义 StrBlobPtr 后才能定义这两个函数
StrBlobPtr end();
private:
    shared_ptr<std::vector<std::string>> data;
    // 如果 data[i] 不合法，抛出一个异常
    void check(size_type i, const std::string &msg) const;
};

inline StrBlob::StrBlob(): data(make_shared<vector<string>>()) { }
StrBlob::StrBlob(initializer_list<string> il):
    data(make_shared<vector<string>>(il)) { }

inline void StrBlob::check(size_type i, const string &msg) const
{
    if (i >= data->size())
        throw out_of_range(msg);
}

inline string& StrBlob::front()
{
    // 如果 vector 为空，check 会抛出一个异常
    check(0, "front on empty StrBlob");
    return data->front();
}

// const 版本 front
inline const string& StrBlob::front() const
{
    check(0, "front on empty StrBlob");
    return data->front();
}

inline string& StrBlob::back()
{
    check(0, "back on empty StrBlob");
    return data->back();
}

// const 版本 back
inline const string& StrBlob::back() const
{
    check(0, "back on empty StrBlob");
    return data->back();
}

```

```

}

inline void StrBlob::pop_back()
{
    check(0, "pop_back on empty StrBlob");
    data->pop_back();
}

// 当试图访问一个不存在的元素时, StrBlobPtr 抛出一个异常
class StrBlobPtr {
public:
    friend bool eq(const StrBlobPtr&, const StrBlobPtr&);

    StrBlobPtr(): curr(0) { }
    StrBlobPtr(StrBlob &a, size_t sz = 0): wptr(a.data), curr(sz) { }

    string& deref() const;
    StrBlobPtr& incr();           // 前缀递增
    StrBlobPtr& decr();           // 前缀递减

private:
    // 若检查成功, check 返回一个指向 vector 的 shared_ptr
    shared_ptr<vector<string>>
        check(size_t, const string&) const;

    // 保存一个 weak_ptr, 意味着底层 vector 可能会被销毁
    weak_ptr<vector<string>> wptr;
    size_t curr;                  // 在数组中的当前位置
};

inline
shared_ptr<vector<string>>
StrBlobPtr::check(size_t i, const string &msg) const
{
    auto ret = wptr.lock(); // vector 还存在吗?
    if (!ret)
        throw runtime_error("unbound StrBlobPtr");
    if (i >= ret->size())
        throw out_of_range(msg);
    return ret;               // 否则, 返回指向 vector 的 shared_ptr
}

inline string& StrBlobPtr::deref() const
{
    auto p = check(curr, "dereference past end");
    return (*p)[curr];         // (*p) 是对象所指向的 vector
}

// 前缀递增: 返回递增后的对象的引用
inline StrBlobPtr& StrBlobPtr::incr()
{
    // 如果 curr 已经指向容器的尾后位置, 就不能递增它
    check(curr, "increment past end of StrBlobPtr");
    ++curr;                   // 推进当前位置
    return *this;
}

```

```

}

// 前缀递减：返回递减后的对象的引用
inline StrBlobPtr& StrBlobPtr::decr()
{
    // 如果 curr 已经为 0，递减它就会产生一个非法下标
    --curr;                                // 递减当前位置
    check(-1, "decrement past begin of StrBlobPtr");
    return *this;
}

// StrBlob 的 begin 和 end 成员的定义
inline StrBlobPtr StrBlob::begin()
{
    return StrBlobPtr(*this);
}

inline StrBlobPtr StrBlob::end()
{
    auto ret = StrBlobPtr(*this, data->size());
    return ret;
}

// StrBlobPtr 的比较操作
inline
bool eq(const StrBlobPtr &lhs, const StrBlobPtr &rhs)
{
    auto l = lhs.wptr.lock(), r = rhs.wptr.lock();
    // 若底层的 vector 是同一个
    if (l == r)
        // 则两个指针都是空，或者指向相同元素时，它们相等
        return (!r || lhs.curr == rhs.curr);
    else
        return false;                      // 若指向不同 vector，则不可能相等
}

inline
bool neq(const StrBlobPtr &lhs, const StrBlobPtr &rhs)
{
    return !eq(lhs, rhs);
}
#endif

```

主程序：

```

#include <iostream>

using namespace std;

#include "my_StrBlob.h"

int main(int argc, char **argv)
{
    StrBlob b1;

```

```

{
    StrBlob b2 = { "a", "an", "the" };
    b1 = b2;
    b2.push_back("about");
    cout << b2.size() << endl;
}
cout << b1.size() << endl;
cout << b1.front() << " " << b1.back() << endl;

const StrBlob b3 = b1;
cout << b3.front() << " " << b3.back() << endl;

for (auto it = b1.begin(); neq(it, b1.end()); it.incr())
    cout << it.deref() << endl;

return 0;
}

```

**练习 12.20:** 编写程序，逐行读入一个输入文件，将内容存入一个 StrBlob 中，用一个 StrBlobPtr 打印出 StrBlob 中的每个元素。

### 【出题思路】

本题练习使用 StrBlob 和 StrBlobPtr。

### 【解答】

用 getline 逐行读取输入文件，存入 StrBlob 后，用 StrBlobPtr 从 StrBlob 的 begin 遍历到 end，逐个打印每个字符串即可。

```

#include <iostream>
#include <fstream>

using namespace std;

#include "my_StrBlob.h"

int main(int argc, char **argv)
{
    ifstream in(argv[1]);
    if (!in) {
        cout << "无法打开输入文件" << endl;
        return -1;
    }

    StrBlob b;
    string s;
    while (getline(in, s))
        b.push_back(s);

    for (auto it = b.begin(); neq(it, b.end()); it.incr())
        cout << it.deref() << endl;

    return 0;
}

```

**练习 12.21:** 也可以这样编写 StrBlobPtr 的 deref 成员：

```
std::string& deref() const
{ return (*check(curr, "dereference past end"))[curr]; }
```

你认为哪个版本更好？为什么？

### 【出题思路】

思考合法性检查不同方式的优缺点。

### 【解答】

书中的方式更好一些。将合法性检查与元素获取和返回语句分离开来，代码更清晰易读，当执行到第二条语句时，已确保 p 是存在的 vector，curr 是合法的位置，可安全地获取元素并返回。这种清晰的结构也更有利于修改不同的处理逻辑。

而本题中的版本将合法性检查和元素获取及返回合在一条语句中，不易读，也不易修改。

**练习 12.22:** 为了能让 StrBlobPtr 使用 const StrBlob，你觉得应该如何修改？定义一个名为 ConstStrBlobPtr 的类，使其能够指向 const StrBlob。

### 【出题思路】

本题练习设计 const 版本。

### 【解答】

首先，为 StrBlobPtr 定义能接受 const StrBlob &参数的构造函数：

```
StrBlobPtr(const StrBlob &a, size_t sz = 0): wptr(a.data), curr(sz)
{ }
```

其次，为 StrBlob 定义能操作 const 对象的 begin 和 end。

声明：

```
StrBlobPtr begin() const;
StrBlobPtr end() const;
```

定义：

```
inline StrBlobPtr StrBlob::begin() const
{
    return StrBlobPtr(*this);
}

inline StrBlobPtr StrBlob::end() const
{
    auto ret = StrBlobPtr(*this, data->size());
    return ret;
}
```

即可实现 StrBlobPtr 使用 const StrBlob：

```
#include <iostream>

using namespace std;

#include "my_StrBlob.h"
```

```

int main(int argc, char **argv)
{
    const StrBlob b = {"Hello", "World", "!"};

    for (auto it = b.begin(); neq(it, b.end()); it.incr())
        cout << it.deref() << endl;

    return 0;
}

```

**练习 12.23:** 编写一个程序，连接两个字符串字面常量，将结果保存在一个动态分配的 `char` 数组中。重写这个程序，连接两个标准库 `string` 对象。

#### 【出题思路】

本题练习使用动态数组。

#### 【解答】

用 `new char[xx]` 即可分配用来保存结果的 `char` 数组，其中 `xx` 应该足以保存结果字符串。由于 C 风格字符串以'\0'结尾，因此 `xx` 应不小于字符数加 1。

对字符串字面常量（即字符数组），可以使用 `strcpy` 将第一个字符串拷贝到动态数组中，然后用 `strcat` 将第二个字符串连接到动态数组中。

对两个 `string` 对象，使用`+`运算即可简单实现连接。然后用 `c_str` 获取连接结果（临时 `string` 对象）的内存地址，用 `strcpy` 拷贝到动态数组即可。

最终，不要忘记释放动态数组。

```

#include <iostream>
#include <cstring>

using namespace std;

int main(int argc, char **argv)
{
    const char *c1 = "Hello ";
    const char *c2 = "World";

    // 字符串所需空间等于字符数+1
    char *r = new char[strlen(c1) + strlen(c2) + 1];
    strcpy(r, c1);                      // 拷贝第一个字符串常量
    strcat(r, c2);                     // 连接第二个字符串常量
    cout << r << endl;

    string s1 = "hello ";
    string s2 = "world";
    strcpy(r, (s1+s2).c_str());        // 拷贝连接结果
    cout << r << endl;

    // 释放动态数组
    delete [] r;

    return 0;
}

```

**练习 12.24:** 编写一个程序，从标准输入读取一个字符串，存入一个动态分配的字符数组中。描述你的程序如何处理变长输入。测试你的程序，输入一个超出你分配的数组长度的字符串。

### 【出题思路】

本题继续练习使用动态数组。

### 【解答】

我们处理变长输入的方法是，根据动态分配的字符数组的大小确定字符串长度阈值，当读取的字符数超出阈值时，停止读取，即采取了截断的方式。还可采取其他处理方式，如抛出异常等。

另外，为了能读取空格等空白符，程序中用到了 `get` 操作来读取字符。

```
#include <iostream>
#include <cstring>

using namespace std;

int main(int argc, char **argv)
{
    char c;

    // 分配 20 个字符的动态数组
    // 因此最多存放 19 个字符
    char *r = new char[20];
    int l = 0;

    while (cin.get(c)) {
        if (isspace(c))      // 空格符、制表符等空白符
            break;
        r[l++] = c;          // 存入动态数组
        if (l == 20) {        // 已读入 19 个字符
            cout << "达到数组容量上限" << endl;
            break;
        }
    }
    r[l] = 0;
    cout << r << endl;

    // 释放动态数组
    delete [] r;

    return 0;
}
```

**练习 12.25:** 给定下面的 `new` 表达式，你应该如何释放 `pa`？

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

### 【出题思路】

理解释放动态数组的特殊方式。

**【解答】**

```
delete [] p;
```

**练习 12.26:** 用 allocator 重写第 427 页中的程序。

**【出题思路】**

本题练习使用 allocator。

**【解答】**

首先，定义一个 `allocator<string> alloc`。

然后，用 `alloc` 的 `allocate` 而不是 `new` 操作来分配内存。这样，只会分配裸内存，而不会初始化 `string` 对象。

接下来，用 `construct` 操作从读取的 `string` 对象来初始化动态数组中的 `string`。

随后动态数组的使用就和往常一样了。

使用完毕后，记住与内存分配和对象构造一样，对象析构（使用 `destroy` 操作）和内存释放（`deallocate` 操作）也是分开的。

```
#include <iostream>
#include <string>
#include <memory>

using namespace std;

int main(int argc, char **argv)
{
    allocator<string> alloc;
    // 分配 100 个未初始化的 string
    auto const p = alloc.allocate(100);
    string s;
    string *q = p; // q 指向第一个 string
    while (cin >> s && q != p + 100)
        alloc.construct(q++, s); // 用 s 初始化*q
    const size_t size = q - p; // 记住读取了多少个 string

    // 使用数组
    for (size_t i = 0; i < size; i++)
        cout << p[i] << " " << endl;

    while (q != p) // 使用完毕后释放已构造的 string
        alloc.destroy(--q);
    alloc.deallocate(p, 100); // 释放内存

    return 0;
}
```

**练习 12.27:** TextQuery 和 QueryResult 类只使用了我们已经介绍过的语言和标准库特性。不要提前看后续章节内容，只用已经学到的知识对这两个类编写你自己的版本。

#### 【出题思路】

本题综合练习已学过的知识实现文本查询程序。

#### 【解答】

12.3.2 节中已经详细介绍了两个类的实现，配套网站上也有完整程序，这里就不再重复给出设计思路和代码。但读者不要直接查看设计思路和程序，应先尝试自己实现，然后再与书中内容和代码对照。

**练习 12.28:** 编写程序实现文本查询，不要定义类来管理数据。你的程序应该接受一个文件，并与用户交互来查询单词。使用 vector、map 和 set 容器保存来自文件的数据并生成查询结果。

#### 【出题思路】

采用过程式程序设计而非面向对象的程序设计来解决这个问题，并体会两者的差异。

#### 【解答】

总体设计思路与面向对象的版本相似，但有一些差异：

1. 由于不用类来管理数据，file 和 wm 都定义为全局变量，便于在函数间共享。当然也可以定义为局部变量，通过函数参数传递。
2. 由于不必进行不同类对象间的数据共享，因此 file 和 wm 中的 set 都不必用 shared\_ptr 管理，直接定义为 vector 和 set 即可。使用它们的代码也要相应修改。
3. 由于不用类来保存查询结果，因此将 query 和 print 函数合二为一。

看起来代码较之面向对象的版本简单了不少，但读者应思考面向对象版本的诸多优势。

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
#include <vector>
#include <map>
#include <set>
// #include <cstring>

#include <cstdlib> // 要使用 EXIT_FAILURE

#include "make_plural.h"

using namespace std;

using line_no = vector<string>::size_type;
```

```

vector<string> file; // 文件每行内容
map<string, set<line_no>> wm; // 单词到行号 set 的映射

string cleanup_str(const string &word)
{
    string ret;
    for (auto it = word.begin(); it != word.end(); ++it) {
        if (!ispunct(*it))
            ret += tolower(*it);
    }
    return ret;
}

void input_text(ifstream &is)
{
    string text;
    while (getline(is, text)) { // 对文件中每一行
        file.push_back(text); // 保存此行文本
        int n = file.size() - 1; // 当前行号
        istringstream line(text); // 将行文本分解为单词
        string word;
        while (line >> word) { // 对行中每个单词
            // 将当前行号插入到其行号 set 中
            // 如果单词不在 wm 中，以之为下标在 wm 中添加一项
            wm[cleanup_str(word)].insert(n);
        }
    }
}

ostream &query_and_print(const string &sought, ostream &os)
{
    // 使用 find 而不是下标运算符来查找单词，避免将单词添加到 wm 中！
    auto loc = wm.find(sought);
    if (loc == wm.end()) { // 未找到
        os << sought << "出现了 0 次" << endl;
    } else {
        auto lines = loc->second; // 行号 set
        os << sought << "出现了" << lines.size() << "次" << endl;
        for (auto num : lines) // 打印单词出现的每一行
            os << "\t(第" << num + 1 << "行) "
                << *(file.begin() + num) << endl;
    }
    return os;
}

void runQueries(ifstream &infile)
{
    // infile 是一个 ifstream，指向我们要查询的文件
    input_text(infile); // 读入文本并建立查询 map
    // 与用户交互：提示用户输入要查询的单词，完成查询并打印结果
    while (true) {
        cout << "enter word to look for, or q to quit: ";

```

```

        string s;
        // 若遇到文件尾或用户输入了 q 时循环终止
        if (!(cin >> s) || s == "q") break;
        // 指向查询并打印结果
        query_and_print(s, cout) << endl;
    }
}

// 程序接受唯一的命令行参数，表示文本文件名
int main(int argc, char **argv)
{
    // 打开要查询的文件
    ifstream infile;
    // 打开文件失败，程序异常退出
    if (argc < 2 || !(infile.open(argv[1]), infile)) {
        cerr << "No input file!" << endl;
        return EXIT_FAILURE;
    }
    runQueries(infile);
    return 0;
}

```

**练习 12.29:** 我们曾经用 `do while` 循环来编写管理用户交互的循环（参见 5.4.4 节，第 169 页）。用 `do while` 重写本节程序，解释你倾向于哪个版本，为什么。

### 【出题思路】

采用过程式程序设计而非面向对象的程序设计来解决这个问题，并体会两者的差异。

### 【解答】

循环改成如下形式即可：

```

do {
    cout << "enter word to look for, or q to quit: ";
    string s;
    // 若遇到文件尾或用户输入了 q 时循环终止
    if (!(cin >> s) || s == "q") break;
    // 指向查询并打印结果
    query_and_print(s, cout) << endl;
} while (true);

```

显然，由于循环中的执行步骤是“输入—检查循环条件—执行查询”，检查循环条件是中间步骤，因此，`while` 和 `do while` 没有什么差异，不会比另一个更简洁。

**练习 12.30:** 定义你自己版本的 `TextQuery` 和 `QueryResult` 类，并执行 12.3.1 节（第 431 页）中的 `runQueries` 函数。

### 【出题思路】

本题综合练习已学过的知识实现文本查询程序。

**【解答】**

解答同练习 12.27。

**练习 12.31:** 如果用 `vector` 代替 `set` 保存行号, 会有什么差别? 哪种方法更好? 为什么?

**【出题思路】**

理解 `vector` 和 `set` 的差异。

**【解答】**

对这个问题而言, `vector` 更好。因为, 虽然 `vector` 不会维护元素值的序, `set` 会维护关键字的序, 但注意到, 我们是逐行读取输入文本的, 因此每个单词出现的行号是自然按升序加入到容器中的, 不必特意用关联容器来保证行号的升序。而从性能角度, `set` 是基于红黑树实现的, 插入操作时间复杂性为  $O(\log n)$  ( $n$  为容器中元素数目), 而 `vector` 的 `push_back` 可达到常量时间。

另外, 一个单词在同一行中可能出现多次。`set` 自然可保证关键字不重复, 但对 `vector` 这也不成为障碍——每次添加行号前与最后一个行号比较一下即可。总体性能仍然是 `vector` 更优。

**练习 12.32:** 重写 `TextQuery` 和 `QueryResult` 类, 用 `StrBlob` 代替 `vector<string>` 保存输入文件。

**【出题思路】**

本题练习在较大程序中配合使用 `StrBlob` 和 `StrBlobPtr` 来代替用 `shared_ptr` 管理的 `vector<string>` 及迭代器。

**【解答】**

对 `my_QueryResult.h`、`my_TextQuery.h` 和 `my_StrBlob.h` 相对于书中的原程序进行如下修改:

1. 在 `my_QueryResult.h` 中包含头文件 `my_StrBlob.h`。
2. `QueryResult` 类的 `file` 成员改为 `StrBlob` 类型, 相应的, 构造函数的第三个参数和成员函数 `get_file` 的返回类型也都改为 `StrBlob` 类型。
3. 类似的 `TextQuery` 类的成员 `file` 也改为 `StrBlob` 类型。
4. 由于 `file` 不再是 `shared_ptr` 而是 `StrBlob`, `TextQuery` 构造函数 (`my_TextQuery.cpp`) 中的 “`file->`” 均改为 “`file.`”。
5. 在原来的代码中, `TextQuery` 构造函数动态分配了一个 `vector<string>`, 用其指针初始化 `file` 成员 (`shared_ptr`)。但 `StrBlob` 类并未定义接受 `vector<string> *` 的构造函数, 因此我们在 `my_StrBlob.h` 文件中为其添加了这个构造函数, 用指针参数直接初始化 `data` 成员 (`shared_ptr`)。
6. 在函数 `print` (`my_TextQuery.cpp`) 中, 用 `file->begin()` 获得了

vector 的首位置迭代器，对其进行加法操作获得了指向第 num 个 string 的迭代器，最后通过解引用获得了这个 string，将其打印出来。但 StrBlobPtr 只定义递增和递减操作，并未定义加法运算。因此，我们为其增加了 my\_StrBlob.h 接受一个整型参数 off 的 deref 操作，能解引用出距当前位置 curr 偏移量为 off 的元素（但并不会修改 curr 的值）。

至此，所需要的修改进行完毕。

可以看到，我们对使用文本查询类的主程序未进行任何修改！读者可好好体会面向对象程序设计将接口和实现分离的优点。

```
my_QueryResult.h:
#ifndef QUERYRESULT_H
#define QUERYRESULT_H
#include <memory>
#include <string>
#include <vector>
#include <set>
#include <iostream>
#include "my_StrBlob.h"

class QueryResult {
friend std::ostream& print(std::ostream&, const QueryResult&);

public:
    typedef std::vector<std::string>::size_type line_no;
    typedef std::set<line_no>::const_iterator line_it;
    QueryResult(std::string s,
                std::shared_ptr<std::set<line_no>> p,
                StrBlob f):
        sought(s), lines(p), file(f) { }
    std::set<line_no>::size_type size() const { return lines->size(); }
    line_it begin() const { return lines->cbegin(); }
    line_it end() const { return lines->cend(); }
    StrBlob get_file() { return file; }
private:
    std::string sought;           // 要查询的单词
    std::shared_ptr<std::set<line_no>> lines; // 单词出现的行号集合
    StrBlob file;                // 输入文件
};

std::ostream &print(std::ostream&, const QueryResult&);
#endif

my_TextQuery.h:

#ifndef TEXTQUERY_H
#define TEXTQUERY_H
#include <memory>
#include <string>
#include <vector>
#include <map>
#include <set>
#include <fstream>
```

```

#include "my_QueryResult.h"

/* this version of the query classes includes two
 * members not covered in the book:
 *   cleanup_str: which removes punctuation and
 *   converst all text to lowercase
 *   display_map: a debugging routine that will print the contents
 *                 of the lookup mape
 */

class QueryResult; // 这个声明是必须的，查询函数中需返回 QueryResult 类型
class TextQuery {
public:
    using line_no = std::vector<std::string>::size_type;
    TextQuery(std::ifstream&);
    QueryResult query(const std::string&) const;
    void display_map(); // 调试辅助函数：打印映射表
private:
    StrBlob file; // 输入文件
    // 将每个单词映射到它所出现的行号的集合
    std::map<std::string,
        std::shared_ptr<std::set<line_no>>> wm;

    // 规范文本：删除标点，并转换为小写
    static std::string cleanup_str(const std::string&);
};

#endif

my_TextQuery.cpp:
#include "my_TextQuery.h"
#include "make_plural.h"

#include <cstddef>
#include <memory>
#include <iostream>
#include <string>
#include <vector>
#include <map>
#include <set>
#include <iostream>
#include <fstream>
#include <cctype>
#include <cstring>
#include <utility>

using std::size_t;
using std::shared_ptr;
using std::istringstream;
using std::string;
using std::getline;
using std::vector;
using std::map;
using std::set;
using std::cerr;

```

```

using std::cout;
using std::cin;
using std::ostream;
using std::endl;
using std::ifstream;
using std::ispunct;
using std::tolower;
using std::strlen;
using std::pair;

// 读取输入文件，建立映射
TextQuery::TextQuery(ifstream &is): file(new vector<string>)
{
    string text;
    while (getline(is, text)) {          // 读取文件的每一行
        file.push_back(text);           // 保存读入的文本行
        int n = file.size() - 1;         // 当前行号
        istringstream line(text);        // 从行中分离出单词
        string word;
        while (line >> word) {         // 对行中的每个单词
            word = cleanup_str(word);
            // 如果单词还未在wm中，使用下标操作将其添加进去
            auto &lines = wm[word]; // lines是一个shared_ptr
            if (!lines) // 第一次遇到一个单词时，此指针为空
                lines.reset(new set<line_no>); // 分配一个新的set
            lines->insert(n);             // 插入当前行号
        }
    }
}

// cleanup_str 删除标点，并将所有文本转换为小写形式，从而查询是大小写不敏感的
string TextQuery::cleanup_str(const string &word)
{
    string ret;
    for (auto it = word.begin(); it != word.end(); ++it) {
        if (!ispunct(*it))
            ret += tolower(*it);
    }
    return ret;
}

QueryResult
TextQuery::query(const string &sought) const
{
    // 如果未找到 sought，将返回一个指向下面这个set的指针
    static shared_ptr<set<line_no>> nodata(new set<line_no>);

    // 使用find而不是下标操作的原因是避免将不在wm中的单词添加进去！
    auto loc = wm.find(cleanup_str(sought));

    if (loc == wm.end())
        return QueryResult(sought, nodata, file); // 未找到
}

```

```

    else
        return QueryResult(sought, loc->second, file);
}

ostream &print(ostream & os, const QueryResult &qr)
{
    // 如果找到了单词，打印出现次数及所有出现的行号
    os << qr.sought << " occurs " << qr.lines->size() << " "
    << make_plural(qr.lines->size(), "time", "s") << endl;

    // 打印单词出现的每一行
    for (auto num : *qr.lines)                                // 对 set 中每个元素
        // 不让用户对从 0 开始的文本行号困惑
        os << "\t(line " << num + 1 << ") "
        << qr.file.begin().deref(num) << endl;

    return os;
}

// 调试函数
void TextQuery::display_map()
{
auto iter = wm.cbegin(), iter_end = wm.cend();

// 对 map 中的每个单词
for ( ; iter != iter_end; ++iter) {
cout << "word: " << iter->first << " {";

// 以常量引用方式获取位置向量，避免拷贝
auto text_locs = iter->second;
auto loc_iter = text_locs->cbegin(),
loc_iter_end = text_locs->cend();

// 打印此单词出现的所有行号
while (loc_iter != loc_iter_end)
{
cout << *loc_iter;

if (++loc_iter != loc_iter_end)
cout << ", ";

}

cout << "}\n";      // 此单词的输出列表结束
}
cout << endl;       // 结束整个 map 的输出
}

my_StrBlob.h:
#ifndef MY_STRBLOB_H
#define MY_STRBLOB_H
#include <vector>
#include <string>

```

```

#include <initializer_list>
#include <memory>
#include <stdexcept>

using namespace std;

// 提前声明，StrBlob 中的友类声明所需
class StrBlobPtr;

class StrBlob {
    friend class StrBlobPtr;
public:
    typedef vector<string>::size_type size_type;
    StrBlob();
    StrBlob(initializer_list<string> il);
    StrBlob(vector<string> *p);
    size_type size() const { return data->size(); }
    bool empty() const { return data->empty(); }
    // 添加和删除元素
    void push_back(const string &t) {data->push_back(t);}
    void pop_back();
    // 元素访问
    string& front();
    const string& front() const;
    string& back();
    const string& back() const;

    // 提供给 StrBlobPtr 的接口
    StrBlobPtr begin(); // 定义 StrBlobPtr 后才能定义这两个函数
    StrBlobPtr end();
    // const 版本
    StrBlobPtr begin() const;
    StrBlobPtr end() const;
private:
    shared_ptr<std::vector<std::string>> data;
    // 如果 data[i] 不合法，抛出一个异常
    void check(size_type i, const std::string &msg) const;
};

inline StrBlob::StrBlob(): data(make_shared<vector<string>>()) { }
inline StrBlob::StrBlob(initializer_list<string> il):
    data(make_shared<vector<string>>(il)) { }
inline StrBlob::StrBlob(vector<string> *p): data(p) { }

inline void StrBlob::check(size_type i, const string &msg) const
{
    if (i >= data->size())
        throw out_of_range(msg);
}

inline string& StrBlob::front()
{
    // 如果 vector 为空，check 会抛出一个异常
}

```

```

    check(0, "front on empty StrBlob");
    return data->front();
}

// const 版本 front
inline const string& StrBlob::front() const
{
    check(0, "front on empty StrBlob");
    return data->front();
}

inline string& StrBlob::back()
{
    check(0, "back on empty StrBlob");
    return data->back();
}

// const 版本 back
inline const string& StrBlob::back() const
{
    check(0, "back on empty StrBlob");
    return data->back();
}

inline void StrBlob::pop_back()
{
    check(0, "pop_back on empty StrBlob");
    data->pop_back();
}

// 当试图访问一个不存在的元素时, StrBlobPtr 抛出一个异常
class StrBlobPtr {
    friend bool eq(const StrBlobPtr&, const StrBlobPtr&);

public:
    StrBlobPtr(): curr(0) { }
    StrBlobPtr(StrBlob &a, size_t sz = 0): wptr(a.data), curr(sz) { }
    StrBlobPtr(const StrBlob &a, size_t sz = 0): wptr(a.data), curr(sz)
    { }

    string& deref() const;
    string& deref(int off) const;
    StrBlobPtr& incr();           // 前缀递增
    StrBlobPtr& decr();           // 前缀递减

private:
    // 若检查成功, check 返回一个指向 vector 的 shared_ptr
    shared_ptr<vector<string>>
    check(size_t, const string&) const;

    // 保存一个 weak_ptr, 意味着底层 vector 可能会被销毁
    weak_ptr<vector<string>> wptr;
    size_t curr;                  // 在数组中的当前位置
};

```

```

inline
shared_ptr<vector<string>>
StrBlobPtr::check(size_t i, const string &msg) const
{
    auto ret = wptr.lock();           // vector 还存在吗?
    if (!ret)
        throw runtime_error("unbound StrBlobPtr");
    if (i >= ret->size())
        throw out_of_range(msg);
    return ret;                      // 否则, 返回指向 vector 的 shared_ptr
}

inline string& StrBlobPtr::deref() const
{
    auto p = check(curr, "dereference past end");
    return (*p)[curr];              // (*p) 是对象所指向的 vector
}

inline string& StrBlobPtr::deref(int off) const
{
    auto p = check(curr + off, "dereference past end");
    return (*p)[curr + off];        // (*p) 是对象所指向的 vector
}

// 前缀递增: 返回递增后的对象的引用
inline StrBlobPtr& StrBlobPtr::incr()
{
    // 如果 curr 已经指向容器的尾后位置, 就不能递增它
    check(curr, "increment past end of StrBlobPtr");
    ++curr;                         // 推进当前位置
    return *this;
}

// 前缀递减: 返回递减后的对象的引用
inline StrBlobPtr& StrBlobPtr::decr()
{
    // 如果 curr 已经为 0, 递减它就会产生一个非法下标
    --curr;                          // 递减当前位置
    check(-1, "decrement past begin of StrBlobPtr");
    return *this;
}

// StrBlob 的 begin 和 end 成员的定义
inline StrBlobPtr StrBlob::begin()
{
    return StrBlobPtr(*this);
}

inline StrBlobPtr StrBlob::end()
{
    auto ret = StrBlobPtr(*this, data->size());
    return ret;
}

```

```

// const 版本
inline StrBlobPtr StrBlob::begin() const
{
    return StrBlobPtr(*this);
}

inline StrBlobPtr StrBlob::end() const
{
    auto ret = StrBlobPtr(*this, data->size());
    return ret;
}

// StrBlobPtr 的比较操作
inline
bool eq(const StrBlobPtr &lhs, const StrBlobPtr &rhs)
{
    auto l = lhs.wptr.lock(), r = rhs.wptr.lock();
    // 若底层的 vector 是同一个
    if (l == r)
        // 则两个指针都是空, 或者指向相同元素时, 它们相等
        return (!r || lhs.curr == rhs.curr);
    else
        return false; // 若指向不同 vector, 则不可能相等
}

inline
bool neq(const StrBlobPtr &lhs, const StrBlobPtr &rhs)
{
    return !eq(lhs, rhs);
}
#endif

主程序:
#include <string>
using std::string;

#include <fstream>
using std::ifstream;

#include <iostream>
using std::cin; using std::cout; using std::cerr;
using std::endl;

#include <cstdlib> // 包含 EXIT_FAILURE 的定义

#include "my_TextQuery.h"
#include "make_plural.h"

void runQueries(ifstream &infile)
{
    // infile 是一个 ifstream, 是我们要查询的文件

```

```

TextQuery tq(infile);           // 保存文件并创建映射表
// 程序主循环：提示用户输入一个单词，查询此单词并打印结果
while (true) {
    cout << "enter word to look for, or q to quit: ";
    string s;
    // 若遇到文件尾或用户输入了'q'，则退出循环
    if (!(cin >> s) || s == "q") break;
    // 执行查询并打印结果
    print(cout, tq.query(s)) << endl;
}
}

// 程序接受单一参数，指出要查询的文件
int main(int argc, char **argv)
{
    // 打开文件，将在其中查询用户指定的单词
    ifstream infile;
    // open 本身返回 void，所以我们用逗号运算符（返回第二个运算对象的值）
    // 检查 infile 的状态
    if (argc < 2 || !(infile.open(argv[1]), infile)) {
        cerr << "No input file!" << endl;
        return EXIT_FAILURE;
    }
    runQueries(infile);
    return 0;
}

```

**练习 12.33：**在第 15 章中我们将扩展查询系统，在 `QueryResult` 类中将会需要一些额外的成员。添加名为 `begin` 和 `end` 的成员，返回一个迭代器，指向一个给定查询返回的行号的 `set` 中的位置。再添加一个名为 `get_file` 的成员，返回一个 `shared_ptr`，指向 `QueryResult` 对象中的文件。

### 【出题思路】

本题练习在较大的类中添加迭代器等功能。

### 【解答】

这些功能的实现非常简单。

对于 `begin` 和 `end` 成员，希望返回行号 `set` 中的位置，因此直接调用 `lines` 的 `cbegin` 和 `cend` 即可。

对于 `get_file`，直接返回 `file` 成员即可。

配套网站中的代码已经是实现了这些功能的版本，这里不再重复列出代码。

# 第 13 章

# 拷贝控制

## 导读

本章介绍了拷贝控制成员和移动控制成员，包括：

- 拷贝控制成员的概念、作用。
- 如何用拷贝控制成员管理资源，来设计自己管理动态内存的类。
- 移动控制成员的概念、作用。

本章的练习着重帮助读者理解拷贝构造函数、析构函数、拷贝赋值运算符的基本概念，何时触发它们，它们起什么作用；理解合成拷贝控制成员的功能；练习定义拷贝控制成员来进行资源管理（特别是实现类值行为和类指针行为的类）或簿记工作；理解移动构造函数和移动赋值运算符及合成版本的概念与拷贝控制成员的区别；理解左值引用和右值引用；练习定义移动控制成员来实现资源的直接转移而非拷贝。

**练习 13.1：**拷贝构造函数是什么？什么时候使用它？

**【出题思路】**

理解拷贝构造函数的基本概念。

**【解答】**

如果构造函数的第一个参数是自身类类型的引用，且所有其他参数（如果有的话）都有默认值，则此构造函数是拷贝构造函数。拷贝构造函数在以下几种情况下会被使用：

- 拷贝初始化（用=定义变量）。
- 将一个对象作为实参传递给非引用类型的形参。
- 一个返回类型为非引用类型的函数返回一个对象。

- 用花括号列表初始化一个数组中的元素或一个聚合类中的成员。
- 初始化标准库容器或调用其 `insert/push` 操作时，容器会对其元素进行拷贝初始化。

**练习 13.2：**解释为什么下面的声明是非法的：

```
Sales_data::Sales_data(Sales_data rhs);
```

**【出题思路】**

理解拷贝构造函数的参数为什么必须是引用类型。

**【解答】**

这一声明是非法的。因为对于上一题所述的情况，我们需要调用拷贝构造函数，但调用永远也不会成功。因为其自身的参数也是非引用类型，为了调用它，必须拷贝其实参，而为了拷贝实参，又需要调用拷贝构造函数，也就是其自身，从而造成死循环。

**练习 13.3：**当我们拷贝一个 `StrBlob` 时，会发生什么？拷贝一个 `StrBlobPtr` 呢？

**【出题思路】**

理解合成的拷贝构造函数是如何工作的。

**【解答】**

这两个类都为定义拷贝构造函数，因此编译器为它们定义了合成的拷贝构造函数。合成的拷贝构造函数逐个拷贝非 `const` 成员，对内置类型的成员，直接进行内存拷贝，对类类型的成员，调用其拷贝构造函数进行拷贝。

因此，拷贝一个 `StrBlob` 时，拷贝其唯一的成员 `data`，使用 `shared_ptr` 的拷贝构造函数来进行拷贝，因此其引用计数增加 1。

拷贝一个 `StrBlobPtr` 时，拷贝成员 `wptr`，用 `weak_ptr` 的拷贝构造函数进行拷贝，引用计数不变，然后拷贝 `curr`，直接进行内存复制。

**练习 13.4：**假定 `Point` 是一个类类型，它有一个 `public` 的拷贝构造函数，指出下面程序片段中哪些地方使用了拷贝构造函数：

```
Point global;
Point foo_bar(Point arg)
{
    Point local = arg, *heap = new Point(global);
    *heap = local;
    Point pa[ 4 ] = { local, *heap };
    return *heap;
}
```

**【出题思路】**

理解何时使用拷贝构造函数。

**【解答】**

如下几个地方使用了拷贝构造函数：

1. local = arg 将 arg 拷贝给 local。
2. \*heap = local; 将 local 拷贝到 heap 指定的地址中。
3. Point pa[ 4 ] = { local, \*heap }; 将 local 和 \*heap 拷贝给数组的前两个元素。
4. return \*heap;。

**练习 13.5：**给定下面的类框架，编写一个拷贝构造函数，拷贝所有成员。你的构造函数应该动态分配一个新的 string（参见 12.1.2 节，第 407 页），并将对象拷贝到 ps 指向的位置，而不是拷贝 ps 本身。

```
class HasPtr {
public:
    HasPtr(const std::string &s = std::string()):
        ps(new std::string(s)), i(0) { }
private:
    std::string *ps;
    int i;
};
```

**【出题思路】**

本题练习定义拷贝构造函数。

**【解答】**

```
HasPtr::HasPtr(const HasPtr &hp) {
    ps = new string(*hp.ps); // 拷贝 ps 指向的对象，而不是拷贝指针本身
    i = hp.i;
}
```

**练习 13.6：**拷贝赋值运算符是什么？什么时候使用它？合成拷贝赋值运算符完成什么工作？什么时候会生成合成拷贝赋值运算符？

**【出题思路】**

理解拷贝赋值运算符的基本概念与合成的拷贝赋值运算符。

**【解答】**

拷贝赋值运算符本身是一个重载的赋值运算符，定义为类的成员函数，左侧运算对象绑定到隐含的 this 参数，而右侧运算对象是所属类类型的，作为函数的参数，函数返回指向其左侧运算对象的引用。

当对类对象进行赋值时，会使用拷贝赋值运算符。

通常情况下，合成的拷贝赋值运算符会将右侧对象的非 static 成员逐个赋予左侧对象的对应成员，这些赋值操作是由成员类型的拷贝赋值运算符来完成的。

若一个类未定义自己的拷贝赋值运算符，编译器就会为其合成拷贝赋值运算符，完成赋值操作，但对于某些类，还会起到禁止该类型对象赋值的效果。

**练习 13.7:** 当我们将一个 StrBlob 赋值给另一个 StrBlob 时，会发生什么？赋值 StrBlobPtr 呢？

#### 【出题思路】

理解合成的拷贝赋值运算符。

#### 【解答】

由于两个类都未定义拷贝赋值运算符，因此编译器为它们定义了合成的拷贝赋值运算符。

与拷贝构造函数的行为类似，赋值一个 StrBlob 时，赋值其唯一的成员 data，使用 shared\_ptr 的拷贝赋值运算符来完成，因此其引用计数增加 1。

赋值一个 StrBlobPtr 时，赋值成员 wptr，用 weak\_ptr 的拷贝赋值运算符进行赋值，引用计数不变，然后赋值 curr，直接进行内存复制。

**练习 13.8:** 为 13.1.1 节（第 443 页）练习 13.5 中 HasPtr 类编写赋值运算符。类似拷贝构造函数，你的赋值运算符应该将对象拷贝到 ps 指向的位置。

#### 【出题思路】

本题练习拷贝赋值运算符。

#### 【解答】

```
HasPtr&
HasPtr::operator=(const HasPtr &rhs)
{
    auto newps = new string(*rhs.ps); // 拷贝指针指向的对象
    delete ps;                      // 销毁原 string
    ps = newps;                     // 指向新 string
    i = rhs.i;                      // 使用内置的 int 赋值
    return *this;                   // 返回一个此对象的引用
}
```

**练习 13.9:** 析构函数是什么？合成析构函数完成什么工作？什么时候会生成合成析构函数？

#### 【出题思路】

理解析构函数的基本概念与合成析构函数。

#### 【解答】

析构函数完成与构造函数相反的工作：释放对象使用的资源，销毁非静态数据成员。从语法上看，它是类的一个成员函数，名字是波浪号接类名，没有返回值，也不接受参数。

当一个类没有定义析构函数时，编译器会为它合成析构函数。

合成的析构函数体为空，但这并不意味着它什么也不做，当空函数体执行完后，非静态数据成员会被逐个销毁。也就是说，成员是在析构函数体之后隐含的析构阶

段中进行销毁的。

**练习 13.10:** 当一个 StrBlob 对象销毁时会发生什么？一个 StrBlobPtr 对象销毁时呢？

**【出题思路】**

理解合成的析构函数。

**【解答】**

这两个类都没有定义析构函数，因此编译器会为它们合成析构函数。

对 StrBlob，合成析构函数的空函数体执行完毕后，会进行隐含的析构阶段，销毁非静态数据成员 data。这会调用 shared\_ptr 的析构函数，将引用计数减 1，引用计数变为 0，会销毁共享的 vector 对象。

对 StrBlobPtr，合成析构函数在隐含的析构阶段会销毁数据成员 wptr 和 curr，销毁 wptr 会调用 weak\_ptr 的析构函数，引用计数不变，而 curr 是内置类型，销毁它不会有特殊动作。

**练习 13.11:** 为前面练习中的 HasPtr 类添加一个析构函数。

**【出题思路】**

练习设计析构函数。

**【解答】**

只需释放 string 对象占用的空间即可。

```
~HasPtr() { delete ps; }
```

**练习 13.12:** 在下面的代码片段中会发生几次析构函数调用？

```
bool fcn(const Sales_data *trans, Sales_data accum)
{
    Sales_data item1(*trans), item2(accum);
    return item1.isbn() != item2.isbn();
}
```

**【出题思路】**

理解析构函数何时执行。

**【解答】**

这段代码中会发生三次析构函数调用：

1. 函数结束时，局部变量 item1 的生命期结束，被销毁，Sales\_data 的析构函数被调用。
2. 类似的，item2 在函数结束时被销毁，Sales\_data 的析构函数被调用。
3. 函数结束时，参数 accum 的生命期结束，被销毁，Sales\_data 的析构函数被调用。

在函数结束时，trans 的生命期也结束了，但它是 Sales\_data 的指针，并不

是它指向的 `Sales_data` 对象的生命期结束（只有 `delete` 指针时，指向的动态对象的生命期才结束），所以不会引起析构函数的调用。

**练习 13.13：**理解拷贝控制成员和构造函数的一个好方法是定义一个简单的类，为该类定义这些成员，每个成员都打印出自己的名字：

```
struct X {
    X() { std::cout << "X()" << std::endl; }
    X(const X&) { std::cout << "X(const X&)" << std::endl; }
};
```

给 `X` 添加拷贝赋值运算符和析构函数，并编写一个程序以不同方式使用 `X` 的对象：将它们作为非引用和引用参数传递；动态分配它们；将它们存放于容器中；诸如此类。观察程序的输出，直到你确认理解了什么时候会使用拷贝控制成员，以及为什么会使用它们。当你观察程序输出时，记住编译器可以略过对拷贝构造函数的调用。

### 【出题思路】

理解拷贝构造函数、拷贝赋值运算符以及析构函数何时执行。

### 【解答】

程序如下所示。请编译运行程序，观察输出结果，仔细分析每次调用对应哪个对象。例如，程序结束时有三次析构函数的调用，分别对应 `x`、`y` 和 `vx` 的第一个元素。

```
#include <iostream>
#include <vector>

using namespace std;

struct X {
    X() { cout << "构造函数 X()" << endl; }
    X(const X&) { cout << "拷贝构造函数 X(const X&)" << endl; }
    X& operator=(const X &rhs) { cout << "拷贝赋值运算符 =(const X&)" << endl; return *this; }
    ~X() { cout << "析构函数 ~X()" << endl; }
};

void f1(X x)
{
}

void f2(X &x)
{
}

int main(int argc, char **argv)
{
    cout << "局部变量：" << endl;
```

```

X x;
cout << endl;

cout << "非引用参数传递: " << endl;
f1(x);
cout << endl;

cout << "引用参数传递: " << endl;
f2(x);
cout << endl;

cout << "动态分配: " << endl;
X *px = new X;
cout << endl;

cout << "添加到容器中: " << endl;
vector<X> vx;
vx.push_back(x);
cout << endl;

cout << "释放动态分配对象: " << endl;
delete px;
cout << endl;

cout << "间接初始化和赋值: " << endl;
X y = x;
y = x;
cout << endl;

cout << "程序结束: " << endl;
return 0;
}

```

**练习 13.14:** 假定 numbered 是一个类，它有一个默认构造函数，能为每个对象生成一个唯一的序号，保存在名为 mysn 的数据成员中。假定 numbered 使用合成的拷贝控制成员，并给定如下函数：

```
void f (numbered s) { cout << s.mysn << endl; }
```

则下面代码输出什么内容？

```
numbered a, b = a, c = b;
f(a); f(b); f(c);
```

### 【出题思路】

理解拷贝控制成员的应用场合。

### 【解答】

这是一个典型的应该定义拷贝控制成员的场合。如果不定义拷贝构造函数和拷贝赋值运算符，依赖合成的版本，则在拷贝构造和赋值时，会简单复制数据成员。对本问题来说，就是将序号简单复制给新对象。

因此，代码中对 a、b、c 三个对象调用函数 f，会输出三个相同的序号——合

成拷贝构造函数被调用时简单复制序号，使得三个对象具有相同的序号。

**练习 13.15：**假定 `numbered` 定义了一个拷贝构造函数，能生成一个新的序号。这会改变上一题中调用的输出结果吗？如果会改变，为什么？新的输出结果是什么？

#### 【出题思路】

理解拷贝构造函数的使用。

#### 【解答】

在此程序中，都是拷贝构造函数在起作用，因此定义能生成新的序号的拷贝构造函数会改变输出结果。

但注意，新的输出结果不是 0、1、2，而是 3、4、5。

因为在定义变量 `a` 时，默认构造函数起作用，将其序号设定为 0。当定义 `b`、`c` 时，拷贝构造函数起作用，将它们的序号分别设定为 1、2。

但是，在每次调用函数 `f` 时，由于参数是 `numbered` 类型，又会触发拷贝构造函数，使得每一次都将形参 `s` 的序号设定为新值，从而导致三次的输出结果是 3、4、5。

**练习 13.16：**如果 `f` 中的参数是 `const numbered&`，将会怎样？这会改变输出结果吗？如果会改变，为什么？新的输出结果是什么？

#### 【出题思路】

理解参数类型与拷贝构造函数的关系。

#### 【解答】

会改变输出结果，新结果是 0、1、2。

原因是，将参数改为 `const numbered &`。由于形参类型由类类型变为引用类型，传递的不是类对象而是类对象的引用。这意味着调用 `f` 时不再触发拷贝构造函数将实参拷贝给形参，而是传递实参的引用。因此，对每次调用，`s` 都是指向实参的引用，序号自然就是实参的序号。而不是创建一个新的对象，获得一个新序号。

**练习 13.17：**分别编写前三题中所描述的 `numbered` 和 `f`，验证你是否正确预测了输出结果。

#### 【出题思路】

验证你对拷贝构造函数的理解。

#### 【解答】

程序如下所示，注释部分是三题不同的部分。

```
#include <iostream>

using namespace std;
```

```

class numbered {
private:
    static int seq;
public:
    numbered () { mysn = seq++; }
    // 13.15
    // numbered (numbered &n) { mysn = seq++; }
    int mysn;
};

int numbered::seq = 0;

// 13.16
// void f(const numbered &s)
void f(numbered s)
{
    cout << s.mysn << endl;
}

int main(int argc, char **argv)
{
    numbered a, b = a, c = b;
    f(a); f(b); f(c);

    return 0;
}

```

**练习 13.18:** 定义一个 Employee 类，它包含雇员的姓名和唯一的雇员证号。为这个类定义默认构造函数，以及接受一个表示雇员姓名的 string 的构造函数。每个构造函数应该通过递增一个 static 数据成员来生成一个唯一的证号。

### 【出题思路】

练习定义拷贝构造成员。

### 【解答】

程序如下所示。

```

#include <iostream>
#include <string>

using namespace std;

class Employee {
private:
    static int sn;

public:
    Employee () { mysn = sn++; }
    Employee (const string &s) { name = s; mysn = sn++; }
    // 为 13.19 题定义的拷贝构造函数和拷贝赋值运算符
    // Employee (Employee &e) { name = e.name; mysn = sn++; }
    // Employee& operator=(Employee &rhs) { name = rhs.name; return *this; }

```

```

const string &get_name() { return name; }
int get_mysn() { return mysn; }

private:
    string name;
    int mysn;
};

int Employee::sn = 0;

void f(Employee &s)
{
    cout << s.get_name() << ":" << s.get_mysn() << endl;
}

int main(int argc, char **argv)
{
    Employee a("赵"), b = a, c;
    c = b;
    f(a); f(b); f(c);

    return 0;
}

```

**练习 13.19:** 你的 Employee 类需要定义它自己的拷贝控制成员吗？如果需要，为什么？如果不需 要，为什么？实现你认为 Employee 需要的拷贝控制成员。

#### 【出题思路】

练习定义拷贝构造成员。

#### 【解答】

如上题程序，当用 a 初始化 b 时，会调用拷贝构造函数。如果不定义拷贝构造函数，则合成的拷贝构造函数简单复制 mysn，会使两者的序号相同。

当用 b 为 c 赋值时，会调用拷贝赋值运算符。如果不定义自己的版本，则编译器定义的合成版本会简单复制 mysn，会使两者的序号相同。

定义的两个拷贝控制成员见上一题程序中的注释。

**练习 13.20:** 解释当我们拷贝、赋值或销毁 TextQuery 和 QueryResult 类（参见 12.3 节，第 430 页）对象时会发生什么。

#### 【出题思路】

理解拷贝构造成员。

#### 【解答】

两个类都未定义拷贝控制成员，因此都是编译器为它们定义合成版本。

当 TextQuery 销毁时，合成版本会销毁其 file 和 wms 成员。对 file 成员，会将 shared\_ptr 的引用计数减 1，若变为 0，则销毁所管理的动态 vector 对象。

(会调用 `vector` 和 `string` 的析构函数)。对 `wm`, 调用 `map` 的析构函数(从而调用 `string`、`shared_ptr` 和 `set` 的析构函数), 会正确释放资源。

当 `QueryResult` 销毁时, 合成版本会销毁其 `sought`、`lines` 和 `file` 成员。类似 `TextQuery`、`string`、`shared_ptr`、`set`、`vector` 的析构函数可能被调用, 因为这些类都有设计良好的拷贝控制成员, 会正确释放资源。

当拷贝一个 `TextQuery` 时, 合成版本会拷贝 `file` 和 `wm` 成员。对 `file`, `shared_ptr` 的引用计数会加 1。对 `wm`, 会调用 `map` 的拷贝构造函数(继而调用 `string`、`shared_ptr` 和 `set` 的拷贝构造函数), 因此会正确进行拷贝操作。赋值操作类似, 只不过会将原来的资源释放掉, 例如, 原有的 `file` 的引用计数会减 1。

`QueryResult` 的拷贝和赋值类似。

**练习 13.21:** 你认为 `TextQuery` 和 `QueryResult` 类需要定义它们自己版本的拷贝控制成员吗? 如果需要, 为什么? 如果不需要, 为什么? 实现你认为这两个类需要的拷贝控制操作。

#### 【出题思路】

理解拷贝构造成员。

#### 【解答】

两个类虽然都未定义拷贝控制成员, 但它们用智能指针管理共享的动态对象(输入文件内容, 查询结果的行号集合), 用标准库容器保存大量容器。而这些标准库机制都有设计良好的拷贝控制成员, 用合成的拷贝控制成员简单地拷贝、赋值、销毁它们, 即可保证正确的资源管理。因此, 这两个类并不需要定义自己的拷贝控制成员。实际上, 这两个类的类对象之间就存在资源共享, 目前的设计已能很好地实现这种共享, 同类对象之间的共享也自然能够解决。

**练习 13.22:** 假定我们希望 `HasPtr` 的行为像一个值。即, 对于对象所指向的 `string` 成员, 每个对象都有一份自己的拷贝。我们将在下一节介绍拷贝控制成员的定义。但是, 你已经学习了定义这些成员所需的所有知识。在继续学习下一节之前, 为 `HasPtr` 编写拷贝构造函数和拷贝赋值运算符。

#### 【出题思路】

本题练习如何让一个类“行为像值”。

#### 【解答】

在之前的习题中, 我们已经为 `HasPtr` 定义了拷贝构造函数和拷贝赋值运算符, 两者相结合, 再加上析构函数(`delete ps` 即可), 已可达到目的。

```
#include <iostream>
#include <string>

using namespace std;
```

```

class HasPtr {
public:
    HasPtr(const string &s = string()):
        ps(new string(s)), i(0) {}
    HasPtr(const HasPtr &p):
        ps(new string(*p.ps)), i(p.i) {} // 拷贝构造函数
    HasPtr& operator=(const HasPtr&); // 拷贝赋值运算符
    HasPtr& operator=(const string&); // 赋予新string
    string& operator*(); // 解引用
    ~HasPtr();
private:
    string *ps;
    int i;
};

HasPtr::~HasPtr()
{
    delete ps; // 释放 string 内存
}

inline
HasPtr& HasPtr::operator=(const HasPtr &rhs)
{
    auto newps = new string(*rhs.ps); // 拷贝指针指向的对象
    delete ps; // 销毁原 string
    ps = newps; // 指向新 string
    i = rhs.i; // 使用内置的 int 赋值
    return *this; // 返回一个此对象的引用
}

HasPtr& HasPtr::operator=(const string &rhs)
{
    *ps = rhs;
    return *this;
}

string& HasPtr::operator*()
{
    return *ps;
}

int main(int argc, char **argv)
{
    HasPtr h("hi mom!");
    HasPtr h2(h); // 行为类值, h2、h3 和 h 指向不同 string
    HasPtr h3 = h;
    h2 = "hi dad!";
    h3 = "hi son!";
    cout << "h: " << *h << endl;
    cout << "h2: " << *h2 << endl;
    cout << "h3: " << *h3 << endl;
    return 0;
}

```

**练习 13.23:** 比较上一节练习中你编写的拷贝控制成员和这一节中的代码。确定你理解了你的代码和我们的代码之间的差异（如果有的话）。

**【出题思路】**

理解拷贝控制成员的规范写法。

**【解答】**

请仔细体会拷贝赋值运算符的规范写法，它是如何保证自赋值安全的。

**练习 13.24:** 如果本节中的 `HasPtr` 版本未定义析构函数，将会发生什么？如果未定义拷贝构造函数，将会发生什么？

**【出题思路】**

理解拷贝控制成员的作用。

**【解答】**

如果未定义析构函数，在销毁 `HasPtr` 对象时合成的析构函数不会释放指针 `ps` 指向的内存，造成内存泄漏。

如果未定义拷贝构造函数，在拷贝 `HasPtr` 对象时，合成的拷贝构造函数会简单复制 `ps` 成员，使得两个 `HasPtr` 指向相同的 `string`。当其中一个 `HasPtr` 修改 `string` 内容时，另一个 `HasPtr` 也被改变，这并不符合我们的设想。如果同时定义了析构函数，情况会更为糟糕，当销毁其中一个 `HasPtr` 时，`ps` 指向的 `string` 被销毁，另一个 `HasPtr` 的 `ps` 成为空悬指针。

**练习 13.25:** 假定希望定义 `StrBlob` 的类值版本，而且我们希望继续使用 `shared_ptr`，这样我们的 `StrBlobPtr` 类就仍能使用指向 `vector` 的 `weak_ptr` 了。你修改后的类将需要一个拷贝构造函数和一个拷贝赋值运算符，但不需要析构函数。解释拷贝构造函数和拷贝赋值运算符必须要做什么。解释为什么不需要析构函数。

**【出题思路】**

本题综合练习拷贝控制成员的使用。

**【解答】**

由于希望 `StrBlob` 的行为像值一样，因此在拷贝构造函数和拷贝赋值运算符中，我们应该将其数据——`string` 的 `vector` 拷贝一份，使得两个 `StrBlob` 对象指向各自的数据，而不是简单拷贝 `shared_ptr` 使得两个 `StrBlob` 指向同一个 `vector`。

`StrBlob` 不需要析构函数的原因是，它管理的全部资源就是 `string` 的 `vector`，而这是由 `shared_ptr` 负责管理的。当一个 `StrBlob` 对象销毁时，会调用 `shared_ptr` 的析构函数，它会正确调整引用计数，当需要时（引用计数变为 0）释放 `vector`。即，`shared_ptr` 保证了资源分配、释放的正确性，`StrBlob` 就不必进行相应的处理了。

**练习 13.26:** 对上一题中描述的 StrBlob 类，编写你自己的版本。

**【出题思路】**

本题综合练习使用拷贝控制成员实现类值行为。

**【解答】**

程序如下所示。可以看到，虽然主程序与练习 12.19 一样，但由于我们定义了拷贝构造函数和拷贝赋值运算符，使得 StrBlob 的行为像值一样，因此 b2 和 b1、b3 和 b1 不再共享 vector，而是都指向自己的拷贝。当向其中之一添加元素时，另一个的内容不会发生改变。读者可以注释掉拷贝构造函数与（或）拷贝赋值运算符，观察有无拷贝控制成员程序输出结果的不同。

另外一个值得注意的是拷贝赋值运算符的写法，由于 StrBlob 是用 shared\_ptr 而非内置指针类型来管理动态对象，因此直接将新创建的 shared\_ptr 赋予了 data，这不会导致自赋值错误。data 指向新的动态对象，引用计数为 1；而 shared\_ptr 的赋值运算符会将 data 原来指向的对象的引用计数减 1。当进行自赋值时，这显然不会导致非法指针问题，语义也是合理的一—data 脱离原共享对象，指向与原对象内容相同的新对象。

```
myStrBlob.h:
#ifndef MY_STRBLOB_H
#define MY_STRBLOB_H
#include <vector>
#include <string>
#include <initializer_list>
#include <memory>
#include <stdexcept>

using namespace std;

// 提前声明，StrBlob 中的友类声明所需
class StrBlobPtr;

class StrBlob {
    friend class StrBlobPtr;
public:
    typedef vector<string>::size_type size_type;
    StrBlob();
    StrBlob(initializer_list<string> il);
    StrBlob(vector<string> *p);
    StrBlob(StrBlob &s);
    StrBlob& operator= (StrBlob &rhs);
    size_type size() const { return data->size(); }
    bool empty() const { return data->empty(); }
    // 添加和删除元素
    void push_back(const string &t) {data->push_back(t);}
    void pop_back();
    // 元素访问
    string& front();
    const string& front() const;
    string& back();
```

```

const string& back() const ;

// 提供给 StrBlobPtr 的接口
StrBlobPtr begin(); // 定义 StrBlobPtr 后才能定义这两个函数
StrBlobPtr end();
// const 版本
StrBlobPtr begin() const;
StrBlobPtr end() const;
private:
    shared_ptr<std::vector<std::string>> data;
    // 如果 data[i] 不合法，抛出一个异常
    void check(size_type i, const std::string &msg) const;
};

inline StrBlob::StrBlob(): data(make_shared<vector<string>>()) { }
inline StrBlob::StrBlob(initializer_list<string> il):
    data(make_shared<vector<string>>(il)) { }
inline StrBlob::StrBlob(vector<string> *p): data(p) { }
inline StrBlob::StrBlob(StrBlob &s): data(make_shared<vector<string>>(*s.data)) { }

inline StrBlob& StrBlob::operator= (StrBlob &rhs)
{
    data = make_shared<vector<string>>(*rhs.data);
    return *this;
}

inline void StrBlob::check(size_type i, const string &msg) const
{
    if (i >= data->size())
        throw out_of_range(msg);
}

inline string& StrBlob::front()
{
    // 如果 vector 为空，check 会抛出一个异常
    check(0, "front on empty StrBlob");
    return data->front();
}

// const 版本 front
inline const string& StrBlob::front() const
{
    check(0, "front on empty StrBlob");
    return data->front();
}

inline string& StrBlob::back()
{
    check(0, "back on empty StrBlob");
    return data->back();
}

// const 版本 back

```

```

inline const string& StrBlob::back() const
{
    check(0, "back on empty StrBlob");
    return data->back();
}

inline void StrBlob::pop_back()
{
    check(0, "pop_back on empty StrBlob");
    data->pop_back();
}

// 当试图访问一个不存在的元素时，StrBlobPtr 抛出一个异常
class StrBlobPtr {
    friend bool eq(const StrBlobPtr&, const StrBlobPtr&);

public:
    StrBlobPtr(): curr(0) { }
    StrBlobPtr(StrBlob &a, size_t sz = 0): wptr(a.data), curr(sz) { }
    StrBlobPtr(const StrBlob &a, size_t sz = 0): wptr(a.data), curr(sz)
    { }

    string& deref() const;
    string& deref(int off) const;
    StrBlobPtr& incr();           // 前缀递增
    StrBlobPtr& decr();           // 前缀递减

private:
    // 若检查成功，check 返回一个指向 vector 的 shared_ptr
    shared_ptr<vector<string>>
    check(size_t, const string&) const;

    // 保存一个 weak_ptr，意味着底层 vector 可能会被销毁
    weak_ptr<vector<string>> wptr;
    size_t curr;                  // 在数组中的当前位置
};

inline
shared_ptr<vector<string>>
StrBlobPtr::check(size_t i, const string &msg) const
{
    auto ret = wptr.lock(); // vector 还存在吗？
    if (!ret)
        throw runtime_error("unbound StrBlobPtr");
    if (i >= ret->size())
        throw out_of_range(msg);
    return ret;                // 否则，返回指向 vector 的 shared_ptr
}

inline string& StrBlobPtr::deref() const
{
    auto p = check(curr, "dereference past end");
    return (*p)[curr];          // (*p) 是对象所指向的 vector
}

```

```

inline string& StrBlobPtr::deref(int off) const
{
    auto p = check(curr + off, "dereference past end");
    return (*p)[curr + off];      // (*p)是对象所指向的vector
}

// 前缀递增：返回递增后的对象的引用
inline StrBlobPtr& StrBlobPtr::incr()
{
    // 如果 curr 已经指向容器的尾后位置，就不能递增它
    check(curr, "increment past end of StrBlobPtr");
    ++curr;                      // 推进当前位置
    return *this;
}

// 前缀递减：返回递减后的对象的引用
inline StrBlobPtr& StrBlobPtr::decr()
{
    // 如果 curr 已经为 0，递减它就会产生一个非法下标
    --curr;                      // 递减当前位置
    check(-1, "decrement past begin of StrBlobPtr");
    return *this;
}

// StrBlob 的 begin 和 end 成员的定义
inline StrBlobPtr StrBlob::begin()
{
    return StrBlobPtr(*this);
}

inline StrBlobPtr StrBlob::end()
{
    auto ret = StrBlobPtr(*this, data->size());
    return ret;
}

// const 版本
inline StrBlobPtr StrBlob::begin() const
{
    return StrBlobPtr(*this);
}

inline StrBlobPtr StrBlob::end() const
{
    auto ret = StrBlobPtr(*this, data->size());
    return ret;
}

// StrBlobPtr 的比较操作
inline
bool eq(const StrBlobPtr &lhs, const StrBlobPtr &rhs)
{
    auto l = lhs.wptr.lock(), r = rhs.wptr.lock();

```

```

// 若底层的 vector 是同一个
if (l == r)
    // 则两个指针都是空，或者指向相同元素时，它们相等
    return (!r || lhs.curr == rhs.curr);
else
    return false; // 若指向不同 vector，则不可能相等
}

inline
bool neq(const StrBlobPtr &lhs, const StrBlobPtr &rhs)
{
    return !eq(lhs, rhs);
}
#endif

```

主程序：

```

#include <iostream>

using namespace std;

#include "my_StrBlob.h"

int main(int argc, char **argv)
{
    StrBlob b1;
    {
        StrBlob b2 = { "a", "an", "the" };
        b1 = b2;
        b2.push_back("about");
        cout << "b2 大小为" << b2.size() << endl;
        cout << "b2 首尾元素为" << b2.front() << " " << b2.back() << endl;
    }
    cout << "b1 大小为" << b1.size() << endl;
    cout << "b1 首尾元素为" << b1.front() << " " << b1.back() << endl;

    StrBlob b3 = b1;
    b3.push_back("next");
    cout << "b3 大小为" << b3.size() << endl;
    cout << "b3 首尾元素为" << b3.front() << " " << b3.back() << endl;

    cout << "b1 全部元素：" << endl;
    for (auto it = b1.begin(); neq(it, b1.end()); it.incr())
        cout << it.deref() << endl;

    return 0;
}

```

**练习 13.27：** 定义你自己的使用引用计数版本的 HasPtr。

**【出题思路】**

本题练习实现类指针行为。

**【解答】**

参考本节内容，即可实现如下程序。请编译运行它，观察输出结果。

```
#include <iostream>
#include <string>

using namespace std;

class HasPtr {
public:
    // 构造函数分配新的 string 和新的计数器，将计数器置为 1
    HasPtr(const string &s = string()):
        ps(new string(s)), i(0), use(new size_t(1)) {}
    // 拷贝构造函数拷贝所有三个数据成员，并递增计数器
    HasPtr(const HasPtr &p):
        ps(p.ps), i(p.i), use(p.use) { ++*use; } // 拷贝构造函数
    HasPtr& operator=(const HasPtr&); // 拷贝赋值运算符
    HasPtr& operator=(const string&); // 赋予新 string
    string& operator*(); // 解引用
    ~HasPtr();
private:
    string *ps;
    int i;
    size_t *use; // 用来记录有多少个对象共享*ps 的成员
};

HasPtr::~HasPtr()
{
    if (--*use == 0) { // 如果引用计数变为 0
        delete ps; // 释放 string 内存
        delete use; // 释放计数器内存
    }
}

HasPtr& HasPtr::operator=(const HasPtr &rhs)
{
    ++rhs.use; // 递增右侧运算对象的引用计数
    if (--*use == 0) { // 然后递减本对象的引用计数
        delete ps; // 如果没有其他用户
        delete use; // 释放本对象分配的成员
    }
    ps = rhs.ps; // 将数据从 rhs 拷贝到本对象
    i = rhs.i;
    use = rhs.use;
    return *this; // 返回本对象
}

HasPtr& HasPtr::operator=(const string &rhs)
{
    *ps = rhs;
    return *this;
}
```

```

string& HasPtr::operator*()
{
    return *ps;
}
int main(int argc, char **argv)
{
    HasPtr h("hi mom!");
    HasPtr h2 = h; // 未分配新 string, h2 和 h 指向相同的 string
    h = "hi dad!";
    cout << "h: " << *h << endl;
    cout << "h2: " << *h2 << endl;

    return 0;
}

```

**练习 13.28:** 给定下面的类, 为其实现一个默认构造函数和必要的拷贝控制成员。

|                      |                        |
|----------------------|------------------------|
| (a) class TreeNode { | (b) class BinStrTree { |
| private:             | private:               |
| std::string value;   | TreeNode *root;        |
| int count;           | };                     |
| TreeNode *left;      |                        |
| TreeNode *right;     |                        |
| }                    |                        |

### 【出题思路】

本题练习根据问题的实际需求设计拷贝控制成员。

### 【解答】

这是一个二叉树数据结构, 若实现类指针行为, 且 count 用作引用计数, 默认构造函数如下:

```

TreeNode::TreeNode()
: value(""), count(1), left(nullptr), right(nullptr) {}
BinStrTree::BinStrTree() : root(nullptr) {}

```

还可定义其他构造函数, 如:

```

TreeNode::TreeNode (const string &s = string(),
    TreeNode *lchild = nullptr, TreeNode *rchild = nullptr)
: value(s), count(1), left(lchild), right(rchild) {}
BinStrTree::BinStrTree(TreeNode *t = nullptr) : root(t) {}

```

当然, 为了创建出二叉树, 还会有创建节点、设置左右孩子节点等函数, 但不是本题的重点, 因此不再讨论。

由于希望它的行为类指针, 因此需要定义拷贝构造函数和拷贝赋值运算符来拷贝树节点指针而非树节点, 并调整引用计数。还需定义析构函数, 减少引用次数, 当引用计数变为 0 时释放内存。需要注意的是, 二叉树结构并非单一的节点, 而是多层结构, 需要递归遍历左右子树中的所有节点, 进行相应的操作。

### 拷贝构造函数:

```

BinStrTree::BinStrTree(const BinStrTree &bst) :
    root(bst.root) // 拷贝整棵树
{
}

```

```

root->CopyTree();           // 应拷贝整棵树，而非仅仅根节点
}

void TreeNode::CopyTree(void) // 拷贝以此节点为根的子树——增加引用计数
{
    if (left)                 // 左子树不空，拷贝左子树
        left->CopyTree();
    if (right)                // 右子树不空，拷贝右子树
        right->CopyTree();
    count++;
}

// 从某个节点开始拷贝子树
TreeNode::TreeNode(const TreeNode &tn) :
value(tn->value), count(1), left(tn->left), right(tn->right)
{
    if (left)                 // 左子树不空，拷贝左子树
        left->CopyTree();
    if (right)                // 右子树不空，拷贝右子树
        right->CopyTree();
}

```

析构函数：

```

BinStrTree::~BinStrTree()      // 释放整棵树
{
    if (!root->ReleaseTree()) { // 释放整棵树，而非仅仅根节点
        delete root;           // 引用计数为 0，释放节点空间
    }
}

int TreeNode::ReleaseTree(void) // 释放以此节点为根的子树
{
    if (left) {                // 左子树不空，释放左子树
        if (!left->CopyTree())
            delete left;         // 左孩子引用计数为 0，释放其空间
    }
    if (right) {               // 右子树不空，释放右子树
        if (!right->CopyTree())
            delete right;        // 右孩子引用计数为 0，释放其空间
    }
    count--;
    return count;
}

TreeNode::~TreeNode()
{
    // count 为 0 表示资源已被释放，是 delete 触发的析构函数，什么也不做即可
    if (count)
        ReleaseTree();
}

```

**练习 13.29:** 解释 swap(HasPtr&, HasPtr&) 中对 swap 的调用不会导致递归循环。

**【出题思路】**

理解 swap 调用的确定。

**【解答】**

在此 swap 函数中又调用了 swap 来交换 HasPtr 成员 ps 和 i。但这两个成员的类型分别是指针和整型，都是内置类型，因此函数中的 swap 调用被解析为 std::swap，而不是 HasPtr 的特定版本 swap（也就是自身），所以不会导致递归循环。

**练习 13.30:** 为你的类值版本的 HasPtr 编写 swap 函数，并测试它。为你的 swap 函数添加一个打印语句，指出函数什么时候执行。

**【出题思路】**

练习定义 swap。

**【解答】**

参考书中本节内容即可完成 swap 函数的编写。

swap 函数和主函数如下所示：

```
inline
void swap(HasPtr &lhs, HasPtr &rhs)
{
    cout << "交换 " << *lhs.ps << "和" << *rhs.ps << endl;
    swap(lhs.ps, rhs.ps);      // 交换指针，而不是 string 数据
    swap(lhs.i, rhs.i);        // 交换 int 成员
}

int main(int argc, char **argv)
{
    HasPtr h("hi mom!");
    HasPtr h2(h);           // 行为类值，h2、h3 和 h 指向不同 string
    HasPtr h3 = h;
    h2 = "hi dad!";
    h3 = "hi son!";
    swap(h2, h3);
    cout << "h: " << *h << endl;
    cout << "h2: " << *h2 << endl;
    cout << "h3: " << *h3 << endl;

    return 0;
}
```

**练习 13.31:** 为你的 HasPtr 类定义一个<运算符，并定义一个 HasPtr 的 vector。为这个 vector 添加一些元素，并对它执行 sort。注意何时会调用 swap。

**【出题思路】**

理解 swap 的应用。

### 【解答】

<运算符直接返回两个 HasPtr 的 ps 指向的 string 的比较结果即可，但需要注意的是，它应被声明为 const 的。

值得注意的是，在 tdm-gcc 4.8.1 的 STL 实现中，当元素数小于等于 16 时，sort 使用的是插入排序算法，且未使用 swap 交换元素，而是内存区域的整片移动。因此，当我们将命令行参数设定为不超过 16 时（本程序接受唯一的命令行参数表示 HasPtr 对象数目），会发现 swap 并未被调用（没有相应输出）。而当命令行参数大于等于 17 时，就会发现 sort 调用了 swap。因为此时 sort 采用的是快速排序算法，使用了 swap 交换元素。但你会发现，交换元素的次数可能比你预期的快速排序算法的交换元素次数少得多。原因是，在快速排序算法递归排序的过程中，如果子序列长度小于等于 16，又会转到插入排序算法，而不会继续递归直至子序列长度变为 1。

读者可编译运行程序，设置命令行参数，观察 sort 过程中是如何交换元素的。

```
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>

using std::string;
using std::cout;
using std::endl;
using std::vector;
using std::to_string;

class HasPtr {
    friend void swap(HasPtr&, HasPtr&);
public:
    HasPtr(const string &s = string()):  
        ps(new string(s)), i(0) {}  
    HasPtr(const HasPtr &p):  
        ps(new string(*p.ps)), i(p.i) {}           // 拷贝构造函数  
    HasPtr& operator=(const HasPtr&);           // 拷贝赋值运算符  
    HasPtr& operator=(const string&);            // 赋予新 string  
    string& operator*();                          // 解引用  
    bool operator<(const HasPtr&) const;         // 比较运算  
    ~HasPtr();  
private:  
    string *ps;  
    int i;  
};  
  
HasPtr::~HasPtr()  
{  
    delete ps;                                     // 释放 string 内存  
}  
  
inline  
HasPtr& HasPtr::operator=(const HasPtr &rhs)
```

```

{
    auto newps = new string(*rhs.ps); // 拷贝指针指向的对象
    delete ps;                      // 销毁原 string
    ps = newps;                     // 指向新 string
    i = rhs.i;                      // 使用内置的 int 赋值
    return *this;                   // 返回一个此对象的引用
}

HasPtr& HasPtr::operator=(const string &rhs)
{
    *ps = rhs;
    return *this;
}

string& HasPtr::operator*()
{
    return *ps;
}

inline
void swap(HasPtr &lhs, HasPtr &rhs)
{
    using std::swap;
    cout << "交换 " << *lhs.ps << "和" << *rhs.ps << endl;
    swap(lhs.ps, rhs.ps);           // 交换指针，而不是 string 数据
    swap(lhs.i, rhs.i);            // 交换 int 成员
}

bool HasPtr::operator<(const HasPtr &rhs) const
{
    return *ps < *rhs.ps;
}

int main(int argc, char **argv)
{
    vector<HasPtr> vh;
    int n = atoi(argv[1]);
    for (int i = 0; i < n; i++)
        vh.push_back(to_string(n-i));
    for (auto p : vh)
        cout << *p << " ";
    cout << endl;
    sort(vh.begin(), vh.end());
    for (auto p : vh)
        cout << *p << " ";
    cout << endl;

    return 0;
}

```

**练习 13.32:** 类指针的 HasPtr 版本会从 swap 函数受益吗？如果会，得到了什么益处？如果不是，为什么？

**【出题思路】**

深入理解 swap 和类指针的类。

**【解答】**

默认 swap 版本简单交换两个对象的非静态成员，对 HasPtr 而言，就是交换 string 指针 ps、引用计数指针 use 和整型值 i。可以看出，这种语义是符合期望的——两个 HasPtr 指向了原来对方的 string，而两者互换 string 后，各自的引用计数本应该是不变的（都是减 1 再加 1）。因此，默认 swap 版本已经能正确处理类指针 HasPtr 的交换，专用 swap 版本不会带来更多收益。

**练习 13.33:** 为什么 Message 的成员 save 和 remove 的参数是一个 Folder&？为什么我们不将参数定义为 Folder 或是 const Folder&？

**【出题思路】**

理解两个成员函数的语义，以及这种语义导致对参数类型的要求。

**【解答】**

首先，我们需要将给定 Folder 的指针添加到当前 Message 的 folders 集合中。这样，参数类型就不能是 Folder，必须是引用类型。否则，调用 save 时会将实参拷贝给形参，folders.insert 添加的就是形参（与局部变量一样在栈中，save 执行完毕就被销毁）的指针，而非原始 Folder 的指针。而参数为引用类型，就可以令形参与实参指向相同的对象，对形参 f 取地址，得到的就是原始 Folder（实参）的指针。

其次，我们需要调用 addMsg 将当前 Message 的指针添加到 Folder 的 messages 集合中，这意味着我们修改了 Folder 的内容，因此参数不能是 const 的。

**练习 13.34:** 编写本节所描述的 Message。

**【出题思路】**

练习使用拷贝控制成员实现簿记工作。

**【解答】**

参考书中本节内容完成 Message 类的编写，并与配套网站代码进行对比。

**练习 13.35:** 如果 Message 使用合成的拷贝控制成员，将会发生什么？

**【出题思路】**

理解类似本问题的簿记工作需求为何需要拷贝控制成员。

**【解答】**

Message 类包含两个数据成员：content 为 string 类型，folders 为 set。这两个标准库类都有完备的拷贝控制成员，因此 Message 使用合成的拷贝控制成员的话，简单拷贝这两个成员也能实现正确拷贝。

但是，本问题的需求不仅如此。

当拷贝 Message 时，不仅要拷贝这个 Message 在哪些 Folder 中，还要将 Message 加到每个 Folder 中——调用 addMsg。

类似的，当销毁 Message 时，需要将它从所有 Folder 中删除——调用 remMsg。

因此，不能依赖合成的拷贝控制成员，必须设计自己的版本来完成这些簿记工作。

**练习 13.36：**设计并实现对应的 Folder 类。此类应该保存一个指向 Folder 中包含的 Message 的 set。

#### 【出题思路】

本题练习利用拷贝控制成员实现正确的簿记操作。

#### 【解答】

首先，Folder 类有唯一的数据成员，保存文件夹中所有消息的指针：

```
set<Message*> msgs;
```

其次，应该实现 Message 类所用到的两个函数 addMsg 和 remMsg，将消息添加到和删除出文件夹，直接调用 msgs 的操作即可：

```
void addMsg(Message *m) { msgs.insert(m); }
void remMsg(Message *m) { msgs.erase(m); }
```

类似 Message 类将自身添加到和删除出所有 Folder 的成员函数，Folder 也应有将自身添加到和删除出所有 Message 的 folders 集合的成员函数，方便拷贝控制成员调用：

```
void Folder::add_to_Messages(const Folder &f)
{
    for (auto msg : f.msgs)
        msg->addFltr(this); // 将这个 Folder 添加到所有 Message 中
}

void Folder::remove_from_Msgs()
{
    while (!msgs.empty()) // 将这个 Folder 从它所有 Message 中删除
        (*msgs.begin())->remove(*this);
}
```

用到两个 Message 的辅助成员函数：

```
void addFltr(Folder *f) { folders.insert(f); } // 添加给定 Folder

void Message::remove(Folder &f) // 删除给定 Folder
{
    folders.erase(&f); // 将 Folder 从此 Message 中删除
    f.remMsg(this); // 反向：将 Message 也从 Folder 中删除
}
```

接下来就可以定义拷贝控制成员了，首先是拷贝构造函数，它先拷贝 `msgs` 集合，然后调用 `add_to_Messages` 添加到它所有 `Message` 的 `folders` 集合中：

```
Folder::Folder(const Folder &f) : msgs(f.msgs)
{
    add_to_Messages(f); // 将 Folder 添加到它所有 Message 的 folders 中
}
```

析构函数调用 `remove_from_Msgs` 从所有 `Message` 中删除本 `Folder`：

```
Folder::~Folder()
{
    remove_from_Msgs();
}
```

拷贝赋值运算符首先将 `Folder` 从每个旧 `Message` 中删除，然后从右侧 `Folder` 拷贝 `Message` 集合，最后将自身添加到每个新 `Message` 中：

```
Folder& Folder::operator=(const Folder &f)
{
    remove_from_Msgs(); // 从每个 Message 中删除此 Folder
    msgs = f.msgs; // 从右侧运算对象拷贝 Message 集合
    add_to_Messages(f); // 将此 Folder 添加到每个新 Message 中
    return *this;
}
```

**练习 13.37：**为 `Message` 类添加成员，实现向 `folders` 添加或删除一个给定的 `Folder*`。这两个成员类似 `Folder` 类的 `addMsg` 和 `remMsg` 操作。

#### 【出题思路】

本题练习复杂类结构中一些辅助函数的定义。

#### 【解答】

上一题已经定义了 `addFldr`，可类似定义 `remFldr`，如下所示：

```
void remFldr(Folder *f) { folders.erase(f); }
```

**练习 13.38：**我们并未使用拷贝并交换方式来设计 `Message` 的赋值运算符。你认为其原因是什么？

#### 【出题思路】

深入理解这类簿记操作问题中拷贝控制成员的作用。

#### 【解答】

如果采用拷贝并交换方式，执行过程是这样的：

1. 由于赋值运算符的参数是 `Message` 类型，因此会将实参拷贝给形参 `rhs`，这会触发拷贝构造函数，将实参的 `contents` 和 `folders` 拷贝给 `rhs`，并调用 `add_to_Folders` 将 `rhs` 添加到 `folders` 的所有文件夹中。

2. 随后赋值运算符调用 `swap` 交换 `*this` 和 `rhs`，首先遍历两者的 `folders`，将它们从自己的文件夹中删除；然后调用 `string` 和 `set` 的 `swap` 交换它们的 `contents` 和 `folders`；最后，再遍历两者新的 `folders`，将它们分别添加到自己

的新文件夹中。

3. 最后，赋值运算符结束，rhs 被销毁，析构函数调用 `remove_from_Folders` 将 rhs 从自己的所有文件夹中删除。

显然，语义是正确的，达到了预期目的。但效率低下，rhs 创建、销毁并两次添加、删除是无意义的。而采用拷贝赋值运算符的标准编写方式，形参 rhs 为引用类型，就能避免这些冗余操作，具有更好的性能。

**练习 13.39：**编写你自己版本的 `StrVec`，包括自己版本的 `reserve`、`capacity`（参见 9.4 节，第 318 页）和 `resize`（参见 9.3.5 节，第 314 页）。

### 【出题思路】

本题练习设计自己管理内存的类。

### 【解答】

实现 `capacity` 很简单，用 `cap` 指针减去 `elements` 指针即可：

```
size_t capacity() const { return cap - elements; }
```

`reserve` 预留一部分空间，需要一个辅助函数：

```
inline
void StrVec::reallocate(size_t newcapacity)
{
    // 分配新内存
    auto newdata = alloc.allocate(newcapacity);

    // 将数据从旧空间移动到新空间
    auto dest = newdata;           // dest 指向新空间中第一个空闲位置
    auto elem = elements;         // 指向旧空间中下一个元素
    for (size_t i = 0; i != size(); ++i)
        alloc.construct(dest++, std::move(*elem++));

    free();                      // 数据移动完毕，释放旧空间
}
```

`reserve` 判断当前空间是否满足需求，若不满足，调用此辅助函数：

```
void reserve(size_t n) { if (n > capacity()) reallocate(n); }
```

`resize` 有两个版本，区别是不带/带初值：

```
// 添加或销毁对象
inline
void StrVec::resize(size_t n)
{
    if (n > size()) {                     // 添加空字符串
        while (size() < n)
            push_back("");
```

```

    } else if (n < size()) {      // 销毁字符串
        while (size() > n)
            alloc.destroy(--first_free);
    }
}

// 添加对象
inline
void StrVec::resize(size_t n, const std::string &s)
{
    if (n > size()) {
        while (size() < n)
            push_back(s);
    }
}

```

结合书中本节内容，即可实现完整的 StrVec。

**练习 13.40：**为你的 StrVec 类添加一个构造函数，它接受一个 initializer\_list<string>参数。

#### 【出题思路】

本题练习设计列表初始化。

#### 【解答】

通过 begin 和 end 获得列表的整个范围，利用辅助函数 alloc\_n\_copy 分配足够多的空间，并将范围中的元素拷贝过去即可：

```

inline
StrVec::StrVec(std::initializer_list<std::string> il)
{
    // 调用 alloc_n_copy 分配与列表 il 中元素数目一样多的空间
    auto newdata = alloc_n_copy(il.begin(), il.end());
    elements = newdata.first;
    first_free = cap = newdata.second;
}

```

**练习 13.41：**在 push\_back 中，我们为什么在 construct 调用中使用后置递增运算？如果使用前置递增运算的话，会发生什么？

#### 【出题思路】

理解几个指针的含义。

#### 【解答】

因为 first\_free 指向第一个空闲位置，也就是最后一个 string 的尾后位置。当添加新 string 时，应该保存在 first\_free 指向的位置，然后将 first\_free 推进一个位置，因此后置递增运算恰好符合要求。

如果使用前置递增运算，则是先将 first\_free 推进一个位置，然后将新 string 保存在新位置上。显然，这种方法意味着 first\_free 指向最后一个

`string`, 而非尾后位置, 与 `first_free` 的设定不吻合。

**练习 13.42:** 在你的 `TextQuery` 和 `QueryResult` 类(参见 12.3 节, 第 431 页)中用你的 `StrVec` 类代替 `vector<string>`, 以此来测试你的 `StrVec` 类。

### 【出题思路】

练习 `StrVec` 的使用。

### 【解答】

对 `TextQuery.h`、`QueryResult.h` 和 `TextQuery.cpp` 做如下修改:

1. `TextQuery.h` 中 `file` 成员的定义改为:

```
std::shared_ptr<StrVec> file; // 输入字体
```

2. `QueryResult.h` 中 `file` 成员的定义改为:

```
std::shared_ptr<StrVec> file;
```

构造函数的第三个参数改为:

```
std::shared_ptr<StrVec> f
```

成员函数 `get_file` 的定义改为:

```
std::shared_ptr<StrVec> get_file() { return file; }
```

3. `TextQuery.cpp` 中构造函数初始化 `file` 成员的操作改为:

```
TextQuery::TextQuery(ifstream &is): file(new StrVec)
```

与第 12 章的练习类似, 由于类的封装特性, 主程序不用进行任何修改。

**练习 13.43:** 重写 `free` 成员, 用 `for_each` 和 `lambda`(参见 10.3.2 节, 第 346 页) 来代替 `for` 循环 `destroy` 元素。你更倾向于哪种实现, 为什么?

### 【出题思路】

复习 `lambda` 的定义和使用。

### 【解答】

将 `for` 循环改成如下形式即可。注意, `elements` 和 `first_free` 是 `string*` 类型, 因此它们指出的范围中的元素是 `string` 类型。因此, `lambda` 的参数 `s` 应该是 `string&` 类型, 在 `lambda` 的函数体中应该取 `s` 的地址, 用来调用 `destroy`。

```
for_each(elements, first_free,
[] (std::string &s){ alloc.destroy(&s); });
```

我更倾向于 `for_each` 和 `lambda` 版本, 因为这个版本只需指出范围及对范围内元素执行什么操作即可, 而 `for` 版本则需程序员小心控制指针的增减。

**练习 13.44:** 编写标准库 `string` 类的简化版本, 命名为 `String`。你的类应该至少有一个默认构造函数和一个接受 C 风格字符串指针参数的构造函数。使用 `allocator` 为你的 `String` 类分配所需内存。

### 【出题思路】

本题练习定义自己管理内存的类。

**【解答】**

总体上需要注意的地方与 StrVec 的定义类似，只不过 StrVec 管理的元素为 string，而 String 管理的元素为 char。配套网站已有完整程序，读者可以尝试自己定义 String，然后与配套网站中的代码进行对照。

**练习 13.45：**解释右值引用和左值引用的区别。**【出题思路】**

理解左值引用和右值引用。

**【解答】**

所谓右值引用就是必须绑定到右值的引用，通过`&&`获得。右值引用只能绑定到一个将要销毁的对象上，因此可以自由地移动其资源。

左值引用，也就是“常规引用”，不能绑定到要转换的表达式、字面常量或返回右值的表达式。而右值引用恰好相反，可以绑定到这类表达式，但不能绑定到一个左值上。

返回左值的表达式包括返回左值引用的函数及赋值、下标、解引用和前置递增/递减运算符，返回右值的包括返回非引用类型的函数及算术、关系、位和后置递增/递减运算符。可以看到，左值的特点是有持久的状态，而右值则是短暂的。

**练习 13.46：**什么类型的引用可以绑定到下面的初始化器上？

```
int f();
vector<int> vi(100);
int? r1 = f();
int? r2 = vi[0];
int? r3 = r1;
int? r4 = vi[0] * f();
```

**【出题思路】**

深入理解左值引用和右值引用。

**【解答】**

1. r1 必须是右值引用，因为 f 是返回非引用类型的函数，返回值是一个右值。
2. r2 必须是左值引用，因为下标运算返回的是左值。
3. r3 只能是左值引用，因为 r1 是一个变量，而变量是左值。
4. r4 只能是右值引用，因为 vi[0] \* f() 是一个算术运算表达式，返回右值。

**练习 13.47：**对你在练习 13.44（13.5 节，第 470 页）中定义的 String 类，为它的拷贝构造函数和拷贝赋值运算符添加一条语句，在每次函数执行时打印一条信息。**【出题思路】**

理解拷贝何时发生。

**【解答】**

在 String.h 和 String.cpp 中添加，编写一个简单的主程序使用 String，编译运行它，观察其行为即可。

主程序：

```
#include <iostream>
#include "String.h"
#include <vector>

using std::cout;
using std::endl;
using std::vector;

int main()
{
    String s1("One"), s2("Two");
    cout << s1 << " " << s2 << endl << endl;;
    String s3(s2);
    cout << s1 << " " << s2 << " " << s3 << endl << endl;
    s3 = s1;
    cout << s1 << " " << s2 << " " << s3 << endl << endl;
    s3 = String("Three");
    cout << s1 << " " << s2 << " " << s3 << endl << endl;

    vector<String> vs;
    // vs.reserve(4);
    vs.push_back(s1);
    vs.push_back(std::move(s2));
    vs.push_back(String("Three"));
    vs.push_back("Four");
    for_each(vs.begin(), vs.end(), [] (const String &s) { cout << s << " ";
    });
    cout << endl;

    return 0;
}
```

**练习 13.48：** 定义一个 `vector<String>` 并在其上多次调用 `push_back`。运行你的程序，并观察 `String` 被拷贝了多少次。

**【出题思路】**

理解拷贝何时发生。

**【解答】**

如上题程序。观察输出结果可以看出，由于 `String` 定义了接受 C 风格字符串的构造函数，因此只有前两个 `push_back` 触发了拷贝构造函数。

**练习 13.49：** 为你的 `StrVec`、`String` 和 `Message` 类添加一个移动构造函数和一个移动赋值运算符。

**【出题思路】**

本题练习定义移动控制成员。

**【解答】**

书中已有 StrVec 和 Message 类的移动构造函数和移动赋值运算符的详细设计，配套网站上也给出了这三个类的移动构造函数和移动赋值运算符的完整代码。读者可先尝试定义这些移动控制成员，然后与配套网站上的代码进行对比。

String 的移动构造函数设计如下：

```
String(String &&s) noexcept : sz(s.size()), p(s.p)
{ s.p = 0; s.sz = 0; }
```

与 StrVec 和 Message 的设计思路类似，首先直接拷贝参数的指针成员 p 而非拷贝 p 指向的内容，然后将参数的指针和大小清空，使得它处于一个析构安全的状态。

String 的移动赋值运算符设计如下：

```
String & String::operator=(String &&rhs) noexcept
{
    // 显式检查自赋值
    if (this != &rhs) {
        if (p)
            a.deallocate(p, sz);      // 类似析构函数的工作
        p = rhs.p;                  // 接管旧内存
        sz = rhs.sz;
        rhs.p = 0;                 // 令 rhs 析构安全
        rhs.sz = 0;
    }
    return *this;
}
```

仍然是与书中例子类似的编写方式，首先显式检查是否自赋值。若不是，首先将赋值号左侧对象的资源（指针 p）释放掉，然后将右侧对象的指针 p 拷贝给左侧对象，最后将右侧对象置于析构安全的状态。

**练习 13.50：**在你的 String 类的移动操作中添加打印语句，并重新运行 13.6.1 节（第 473 页）的练习 13.48 中的程序，它使用了一个 vector<String>，观察什么时候会避免拷贝。

**【出题思路】**

进一步理解何时使用拷贝控制成员，何时移动控制成员。

**【解答】**

在拷贝/移动构造函数和拷贝/移动赋值运算符中添加打印语句，运行练习 13.47 中的程序，观察输出结果即可。可以看到，vector 操作部分输出了以下内容：

```
Copy Constructor One
Move Constructor Two
Move Constructor One
Move Constructor Three
Move Constructor One
```

```

Move Constructor Two
Move Constructor Four
One Two Three Four

```

容易看出，

`vs.push_back(s1)` 触发一次拷贝构造，对应第一行输出。

`vs.push_back(std::move(s2))` 触发一次移动构造，对应第二行输出。

`vs.push_back(String("Three"))` 触发一次移动构造，对应第四行输出。

`vs.push_back("Four")` 触发一次移动构造，对应第七行输出。

那么，其他几次（移动）构造函数是如何触发的呢？

回忆一下，默认初始化的 `vector` 不分配内存空间。当 `push_back` 发现 `vector` 空间不足以容纳新元素时，分配新的空间（通常是加倍），将数据移动到新的空间中（由于 `String` 定义了移动构造函数，这里确实是“移动”而非“拷贝”），然后释放旧空间。

因此，当插入 `s2` 时，空间由 1 扩为 2，并将原有元素（`One`）移动到新空间，对应第三行输出。

当插入 `Three` 时，空间由 2 扩为 4，将 `One`、`Two` 移动到新空间，产生两次移动构造，对应第五、六两行输出。

尝试在创建 `vector` 后为它预留足够空间：`vs.reserve(4)`，则输出为：

```

Copy Constructor One
Move Constructor Two
Move Constructor Three
Move Constructor Four
One Two Three Four

```

因空间扩展引起的移动构造就不存在了。

**练习 13.51：** 虽然 `unique_ptr` 不能拷贝，但我们在 12.1.5 节（第 418 页）中编写了一个 `clone` 函数，它以值方式返回一个 `unique_ptr`。解释为什么函数是合法的，以及为什么它能正确工作。

### 【出题思路】

理解不可拷贝类型的例外及移动控制成员的触发条件。

### 【解答】

`unique_ptr` 不能拷贝，但有一个例外——将要被销毁的 `unique_ptr` 是可以拷贝或销毁的。因此，在 418 页的 `clone` 函数中返回局部 `unique_ptr` 对象 `ret` 是可以的，因为 `ret` 马上就要被销毁了。而此时的“拷贝”其实是触发移动构造函数进行了移动。

**练习 13.52：** 详细解释第 478 页中的 `HasPtr` 对象的赋值发生了什么？特别是，一步一步描述 `hp`、`hp2` 以及 `HasPtr` 的赋值运算符中的参数 `rhs` 的值发生了什么变化。

**【出题思路】**

理解移动控制成员的执行过程。

**【解答】**

对 `hp = hp2`, 因为 `hp2` 是一个变量, 是一个左值, 因此它传递给赋值运算符参数 `rhs` 的过程是拷贝构造过程, `rhs` 获得 `hp2` 的一个副本, `rhs.ps` 与 `hp2.ps` 指向不同的 `string`, 但两个 `string` 包含相同的内容。在赋值运算符中, 交换 `hp` 和 `rhs`, `rhs` 指向 `hp` 原来的 `string`, 在赋值结束后被销毁。最终结果, `hp` 和 `hp2` 指向两个独立的 `string`, 但内容相同。

对 `hp = std::move(hp2)`, `hp2` 传递给 `rhs` 的过程是移动构造过程, `rhs.ps` 指向 `hp2.ps` 原来的 `string`, `hp2` 的 `ps` 被设置为空指针。然后赋值运算符交换 `hp` 和 `rhs`, `rhs` 指向 `hp` 原来的 `string`, 在赋值结束后被销毁。最终结果 `hp` 指向 `hp2` 原来的 `string`, 而 `hp2` 则变为空。

**练习 13.53:** 从底层效率的角度看, `HasPtr` 的赋值运算符并不理想, 解释为什么。为 `HasPtr` 实现一个拷贝赋值运算符和一个移动赋值运算符, 并比较你的新的移动赋值运算符中执行的操作和拷贝并交换版本中执行的操作。

**【出题思路】**

从性能角度考虑移动控制成员的定义。

**【解答】**

在进行拷贝赋值时, 先通过拷贝构造创建了 `hp2` 的拷贝 `rhs`, 然后再交换 `hp` 和 `rhs`, `rhs` 作为一个中间媒介, 只是起到将值从 `hp2` 传递给 `hp` 的作用, 是一个冗余的操作。

类似的, 在进行移动赋值时, 先从 `hp2` 转移到 `rhs`, 再交换到 `hp`, 也是冗余的。

也就是说, 这种实现方式唯一的用处是统一了拷贝和移动赋值运算, 但在性能角度, 多了一次从 `rhs` 的间接传递, 性能不好。

练习 13.8 已经定义了拷贝赋值运算符, 移动赋值运算符可定义如下:

```
inline
HasPtr& HasPtr::operator=(HasPtr &&rhs) noexcept
{
    cout << "Move Assignment" << endl;
    if (this != &rhs) {
        delete ps;           // 释放旧 string
        ps = rhs.ps;        // 从 rhs 接管 string
        rhs.ps = nullptr;   // 将 rhs 置于析构安全状态
        rhs.i = 0;
    }
    return *this;          // 返回一个此对象的引用
}
```

**练习 13.54:** 如果我们为 `HasPtr` 定义了移动赋值运算符，但未改变拷贝并交换运算符，会发生什么？编写代码验证你的答案。

**【出题思路】**

理解两种赋值运算符的关系。

**【解答】**

会产生编译错误。

因为对于 `hp = std::move(hp2)` 这样的赋值语句来说，两个运算符匹配得一样好，从而产生了二义性。

**练习 13.55:** 为你的 `StrBlob` 添加一个右值引用版本的 `push_back`。

**【出题思路】**

练习定义右值引用版本成员函数。

**【解答】**

定义如下，与左值引用版本的差别除了参数类型外，就是将参数用 `move` 处理后使用。

```
void push_back(string &&t) {data->push_back(std::move(t));}
```

**练习 13.56:** 如果 `sorted` 定义如下，会发生什么：

```
Foo Foo::sorted() const & {
    Foo ret(*this);
    return ret.sorted();
}
```

**【出题思路】**

理解左值引用和右值引用版本的成员函数。

**【解答】**

首先，局部变量 `ret` 拷贝了被调用对象的一个副本。然后，对 `ret` 调用 `sorted`，由于并非是函数返回语句或函数结束（虽然写成一条语句，但执行过程是先调用 `sorted`，然后将结果返回），因此编译器认为它是左值，仍然调用左值引用版本，产生递归循环。

利用右值引用版本来完成排序的期望不能实现。

**练习 13.57:** 如果 `sorted` 定义如下，会发生什么：

```
FOO Foo::sorted() const & { return Foo(*this).sorted(); }
```

**【出题思路】**

理解左值引用和右值引用版本的成员函数。

**【解答】**

与上一题不同，本题的写法可以正确利用右值引用版本来完成排序。原因在于，编译器认为 `Foo(*this)` 是一个“无主”的右值，对它调用 `sorted` 会匹配右值引用版本。

**练习 13.58:** 编写新版本的 Foo 类, 其 sorted 函数中有打印语句, 测试这个类, 来验证你对前两题所给出的答案是否正确。

### 【出题思路】

理解左值引用和右值引用版本的成员函数。

### 【解答】

程序如下, 练习 13.56 的写法会一直输出“左值引用版本”, 直至栈溢出, 程序退出。而练习 13.57 的写法会输出一个“左值引用版本”和一个“右值引用版本”后正确结束。

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

class Foo {
public:
    Foo sorted() &&; // 用于可改变的右值
    Foo sorted() const &; // 可用于任何类型的 Foo
    // Foo 的其他成员的定义
private:
    vector<int> data;
};

// 本对象为右值, 因此可以原址排序
Foo Foo::sorted() &&
{
    cout << "右值引用版本" << endl;
    sort(data.begin(), data.end());
    return *this;
}

// 本对象是 const 或是一个左值, 哪种情况我们都不能对其进行原址排序
Foo Foo::sorted() const & {
    cout << "左值引用版本" << endl;
    Foo ret(*this); // 拷贝一个副本
    return ret.sorted();
    // return Foo(*this).sorted();
}

int main(int argc, char **argv)
{
    Foo f;
    f.sorted();
    return 0;
}
```

# 第 14 章

# 重载运算与类型转换

## 导读

本章介绍了运算符重载的基本概念，介绍了各种运算符的重载方式，并介绍了类型转换与运算符重载的关系。

本章的练习着重让读者掌握各种运算符的重载，包括输入/输出运算符、各种算术运算符、下标运算符以及函数调用运算符等。特别是，结合自定义的 `Sales_data`、`StrBlob` 等较为复杂的类进行运算符重载的练习，让读者体会运算符重载是如何帮助改进代码质量的。此外，还通过一些练习帮助读者弄清类型转换与运算符重载的关系。特别是，当有多个重载版本时，在进行函数匹配时涉及类型转换应如何处理。

**练习 14.1：**在什么情况下重载的运算符与内置运算符有所区别？在什么情况下重载的运算符又与内置运算符一样？

### 【出题思路】

理解重载运算符与内置运算符的区别。

### 【解答】

不同点：

重载操作符必须具有至少一个 `class` 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，例如对 `&&` 和 `||` 的重载版本不再具有“短路求值”的特性，两个操作数都要进行求值，而且不规定操作数的求值顺序。

相同点：

对于优先级和结合性及操作数的数目都不变。

**练习 14.2:** 为 Sales\_data 编写重载的输入、输出、加法和复合赋值运算符的声明。

**【出题思路】**

本题练习重载运算符的声明。

**【解答】**

几个运算符的声明如下所示：

```
class Sales_data
{
    friend std::istream& operator>>(std::istream&, Sales_data & );
    friend std::ostream& operator<<(std::ostream&, const Sales_data & );
    ...
public:
    Sales_data& operator+=(const Sales_data&);
};

Sales_data operator+(const Sales_data&, const Sales_data&);
```

**练习 14.3:** string 和 vector 都定义了重载的==以比较各自的对象，假设 svec1 和 svec2 是存放 string 的 vector，确定在下面的表达式中分别使用了哪个版本的==？

- |                         |                          |
|-------------------------|--------------------------|
| (a) "cobble" == "stone" | (b) svec1[0] == svec2[0] |
| (c) svec1 == svec2      | (d) svec1[0] == "stone"  |

**【出题思路】**

本题旨在理解编译器如何选择重载运算符的不同版本。

**【解答】**

- (a) " cobble " == " store " 应用了 C++ 语言内置版本的==，比较两个指针。
- (b) svec1[0] == svec2[0] 应用了 string 版本的重载==。
- (c) svec1 == svec2 应用了 vector 版本的重载==。
- (d) svec1[0] == "stone" 应用了 string 版本的重载==，字符串字面常量被转换为 string。

**练习 14.4:** 如何确定下列运算符是否应该是类的成员？

- |       |        |        |        |        |        |        |        |
|-------|--------|--------|--------|--------|--------|--------|--------|
| (a) % | (b) %= | (c) ++ | (d) -> | (e) << | (f) && | (g) == | (h) () |
|-------|--------|--------|--------|--------|--------|--------|--------|

**【出题思路】**

理解编译器如何选择重载运算符的不同版本。

**【解答】**

- (a) % 通常定义为非成员。
- (b) %= 通常定义为类成员，因为它会改变对象的状态。
- (c) ++ 通常定义为类成员，因为它会改变对象的状态。
- (d) -> 必须定义为类成员，否则编译会报错。
- (e) << 通常定义为非成员。

- (f) &&通常定义为非成员。
- (g) ==通常定义为非成员。
- (h) ()必须定义为类成员，否则编译会报错。

**练习 14.5：**在 7.5.1 节的练习 7.40（第 261 页）中，编写了下列类中某一个的框架，请问在这个类中应该定义重载的运算符吗？如果是，请写出来。

- |             |            |              |
|-------------|------------|--------------|
| (a) Book    | (b) Date   | (c) Employee |
| (d) Vehicle | (e) Object | (f) Tree     |

#### 【出题思路】

学会判断是否需要为类定义重载运算符。

#### 【解答】

我们以(b) Date 为例，为其定义重载的输出运算符，输入运算符可参照实现。显然，为 Date 定义输入输出运算符，可以让我们像输入输出内置类型对象那样输入输出 Date，在易用性和代码的可读性上有明显的好处。因此，定义这两个重载运算符是合理的。

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

class Date
{
public:
    Date() {}
    Date(int y, int m, int d) {year = y; month = m; day = d;}
    friend ostream& operator<<(ostream &os, const Date &dt);

private:
    int year, month, day;
};

ostream& operator<<(ostream &os, const Date & d)
{
    const char sep = '\t';
    os << "year:" << d.year << sep << "month:" << d.month << sep << "day:"
        << d.day << endl;
    return os;
}
```

**练习 14.6：**为你的 Sales\_data 类定义输出运算符。

#### 【出题思路】

本题练习实现输出运算符。

#### 【解答】

输出运算符将 Sales\_data 的数据成员依次写到指定输出流中即可。

```
class Sales_data
{
```

```

friend ostream& operator<<(ostream &os, const Sales_data &item);
    // 其他成员
};

ostream& operator<<(ostream &os, const Sales_data &item)
{
    const char *sep = ' ';
    os << item.isbn() << sep << item.units_sold << sep <<
        item.revenue << sep << item.avg_price();
    return os;
}

```

输入运算符的实现相反。

复合赋值运算符将给定 Sales\_data 的销售量和销售金额加到当前对象上，加法运算符则可使用复合赋值运算符来实现。

具体实现可参见配套网站上的相关代码。

**练习 14.7：**你在 13.5 节的练习（第 470 页）中曾经编写了一个 String 类，为它定义一个输出运算符。

### 【出题思路】

本题练习实现输出运算符。

### 【解答】

将字符指针成员写入输入流即可，默认的 char \* 的输出运算符会打印保存其中的 C 风格字符串。

```

class String
{
public:
    String();
    String(const char *str);
    friend ostream& operator<<(ostream &os, const String &str);

private:
    char *str;
};

ostream& operator<<(ostream &os, const String &str)
{
    cout << str;
    return os;
}

```

**练习 14.8：**你在 7.5.1 节的练习 7.40（第 261 页）中曾经选择并编写了一个类，为它定义一个输出运算符。

### 【解答】

同练习 14.5。

**练习 14.9:** 为你的 Sales\_data 类定义输入运算符。

**【出题思路】**

本题练习实现输入运算符。

**【解答】**

输入运算符从给定输入流读取对应类型的对象，存入 Sales\_data 的数据成员中。与输出不同，输入通常要进行一些正确性的判定，并进行相应处理。

```
class Sales_data
{
    friend stream& operator>>(istream &is, Sales_data &item);
    // 其他成员
};

istream& operator>>(istream &is, Sales_data &item)
{
    double price;
    is >> item.bookNo >> item.units_sold >> price;
    if (is)
    {
        item.revenue = item.units_sold * price;
    }
    else
    {
        item = Sales_data();
    }

    return is;
}
```

**练习 14.10:** 对于 Sales\_data 的输入运算符来说，如果给定了下面的输入将发生什么情况？

- (a) 0-201-99999-9 10 24.95      (b) 10 24.95 0-210-99999-9

**【出题思路】**

理解重载运算符的工作过程。

**【解答】**

(a) 参数中传入的 Sales\_data 对象将会得到输入的值，其中 bookNo、units\_sold、price 的值分别是：0-201-99999-9、10、24.95，同时 revenue 的值是 249.5。

(b) 输入错误，参数中传入的 Sales\_data 对象将会得到默认值。

**练习 14.11:** 下面的 Sales\_data 输入运算符存在错误吗？如果有，请指出来。对于这个输入运算符如果仍然给定上一个练习的输入将发生什么情况？

```
istream& operator>>(istream& in, Sales_data& s)
{
    double price;
```

```

    in >> s.bookNo >> s.units_sold >> price;
    s.revenue = s.units_sold * price;
    return in;
}

```

**【出题思路】**

理解输入运算符通常要判断输入数据的正确性。

**【解答】**

这个实现没有判断输入数据的正确性，是错误的。

(a) 如果输入的是 0-201-99999-9 10 24.95，程序将会正常执行，`Sales_data` 对象得到正确的值。

(b) 如果输入的是 10 24.95 0-210-99999-9，`bookNo`、`units_sold`、`price` 将会得到错误的值，分别是：10, 24, 0.95，而 `revenue` 的值是： $24 * 0.95 = 22.8$ 。这显然跟我们的预期结果是不一样的。

**练习 14.12：**你在 7.5.1 节的练习 7.40（第 261 页）中曾经选择并编写了一个类，为它定义一个输入运算符并确保该运算符可以处理输入错误。

**【出题思路】**

本题练习判断输入数据的正确性。

**【解答】**

```

#include<iostream>
using namespace std;

class Date
{
public:
    Date() {}
    Date(int y, int m, int d) {year = y; month = m; day = d;}
    friend istream& operator>>(istream &is, Date &dt);

private:
    int year, month, day;
};

istream& operator>>(istream &is, Date &dt)
{
    is >> dt.year >> dt.month >> dt.day;
    if (!is)
    {
        dt = Date(0, 0, 0);
    }

    return is;
}

```

**练习 14.13:** 你认为 Sales\_data 类还应该支持哪些其他算术运算符（参见表 4.1，第 124 页）？如果有的话，请给出它们的定义。

#### 【出题思路】

本题练习重载运算符的实现。

#### 【解答】

对于 Sales\_data 类，其实我们并不需要再为它添加其他算术运算符。但是这里我们可以考虑为它实现一个减法运算符。

```
class Sales_data
{
    friend Sales_data operator-(const Sales_data &lhs, const
Sales_data &rhs);
public:
    Sales_data& operator-=(const Sales_data &rhs);
    // 其他成员.
};

Sales_data operator-(const Sales_data &lhs, const Sales_data &rhs)
{
    Sales_data sub = lhs;
    sub -= rhs;
    return sub;
}

Sales_data& Sales_data::operator-=(const Sales_data &rhs)
{
    units_sold -= rhs.units_sold;
    revenue -= rhs.revenue;
    return *this;
}
```

**练习 14.14:** 你觉得为什么调用 operator+= 来定义 operator+ 比其他方法更有效？

#### 【出题思路】

理解重载运算符的不同实现方式。

#### 【解答】

显然，从头实现 operator+ 的方式与借助 operator+= 实现的方式相比，在性能上没有优势，而可读性上后者显然更好。因此，在此例中代码复用是最好的方式。

**练习 14.15:** 你在 7.5.1 节的练习 7.40（第 261 页）中曾经选择并编写了一个类，你认为它应该含有其他算术运算符吗？如果是，请实现它们；如果不是，解释原因。

#### 【出题思路】

本题练习实现重载运算符。

#### 【解答】

在练习 7.40 中，我们编写了类 Date。算术运算对 Date 并没有太大意义，因此不需要为 Date 重载算术运算符。

**练习 14.16：**为你的 StrBlob 类（参见 12.1.1 节，第 405 页）、StrBlobPtr 类（参见 12.1.6 节，第 421 页）、StrVec 类（参见 13.5 节，第 465 页）和 String 类（参见 13.5 节，第 470 页）分别定义相等运算符和不相等运算符。

### 【出题思路】

本题练习实现相等运算符。

### 【解答】

```
// 为 StrBlob 定义==和!=
class StrBlob
{
    friend bool operator==(const StrBlob &lhs, const StrBlob &rhs);
    friend bool operator!=(const StrBlob &lhs, const StrBlob &rhs);
    // 其他成员
};

bool operator==(const StrBlob &lhs, const StrBlob &rhs)
{
    return lhs.data == rhs.data;    // 所指向的 vector 相等
}

bool operator!=(const StrBlob &lhs, const StrBlob &rhs)
{
    return !(lhs == rhs);
}

// 为 StrBlobPtr 定义==和!=
class StrBlobPtr
{
    friend bool operator==(const StrBlobPtr &lhs, const StrBlobPtr &rhs);
    friend bool operator!=(const StrBlobPtr &lhs, const StrBlobPtr &rhs);
    // 其他成员
};

bool operator==(const StrBlobPtr &lhs, const StrBlobPtr &rhs)
{
    auto l = lhs.wptr.lock(), r = rhs.wptr.lock();
    if (l == r)
        // 两个指针都为空，或指向相同的 vector 且 curr 指向相同元素时，相等，否则不等
        return (!r || lhs.curr == rhs.curr);
    else
        return false; // 指向不同 vector 时，不等
}

bool operator!=(const StrBlobPtr &lhs, const StrBlobPtr &rhs)
{
    return !(lhs == rhs);
```

```

}

// 为 StrVec 定义==和!=
class StrVec
{
    friend bool operator==(const StrVec &lhs, const StrVec &rhs);
    friend bool operator!=(const StrVec &lhs, const StrVec &rhs);
    // 其他成员
};

bool operator==(const StrVec &lhs, const StrVec &rhs)
{
    if (lhs.size() != rhs.size())
    {
        return false;
    }

    for (auto itr1 = lhs.begin(), itr2 = rhs.begin(); itr1 != lhs.end() && itr2 != rhs.end(); itr1++, itr2++)
    {
        if (*itr1 != *itr2)
        {
            return false;
        }
    }

    return true;
}

bool operator!=(const StrVec &lhs, const StrVec &rhs)
{
    return !(lhs == rhs);
}

// 为 String 定义==和!=
class String
{
    friend bool operator==(const String &lhs, const String &rhs);
    friend bool operator!=(const String &lhs, const String &rhs);
    // 其他成员

    private:
        const char *str;
};

bool operator==(const String &lhs, const String &rhs)
{
    return strcmp(lhs.str, rhs.str);
}

bool operator!=(const String &lhs, const String &rhs)
{
    return !(lhs == rhs);
}

```

**练习 14.17:** 你在 7.5.1 节的练习 7.40 ( 第 261 页 ) 中曾经选择并编写了一个类, 你认为它应该含有相等运算符吗? 如果是, 请实现它; 如果不是, 解释原因。

### 【出题思路】

本题练习判断类是否需要相等运算符及实现。

### 【解答】

在练习 7.40 中, 我们实现了 Date 类。因为我们可以比较两个日期是否相等, 因此需要实现相等运算符。

```
class Date
{
    friend bool operator==(const Date &d1, const Date &d2);
    friend bool operator!=(const Date &d1, const Date &d2);
    // 其他成员
}

bool operator==(const Date &d1, const Date &d2)
{
    return d1.year == d2.year && d1.month == d2.month && d1.day == d2.day;
}

bool operator!=(const Date &d1, const Date &d2)
{
    return !(d1 == d2);
}
```

**练习 14.18:** 为你的 StrBlob 类、StrBlobPtr 类、StrVec 类和 String 类定义关系运算符。

### 【出题思路】

本题练习实现关系运算符。

### 【解答】

本题的关键是明确关系运算符的语义。

String 类的关系运算符就是比较两个字符串字典序的先后。

```
class String
{
    friend bool operator<(const String &s1, const String &s2);
    friend bool operator<=(const String &s1, const String &s2);
    friend bool operator>(const String &s1, const String &s2);
    friend bool operator>=(const String &s1, const String &s2);
    // 其他成员
};

bool operator<(const String &s1, const String &s2)
{
    return strcmp(s1.str, s2.str) < 0;
}

bool operator<=(const String &s1, const String &s2)
{
```

```

        return strcmp(s1.str, s2.str) < 0;
    }

    bool operator>(const String &s1, const String &s2)
    {
        return strcmp(s1.str, s2.str) > 0;
    }

    bool operator>=(const String &s1, const String &s2)
    {
        return strcmp(s1.str, s2.str) >= 0;
    }
}

```

两个 StrBlob 的比较就是比较两个字符串 vector。

```

class StrBlob
{
    friend bool operator<(const StrBlob &s1, const StrBlob &s2);
    friend bool operator<=(const StrBlob &s1, const StrBlob &s2);
    friend bool operator>(const StrBlob &s1, const StrBlob &s2);
    friend bool operator>=(const StrBlob &s1, const StrBlob &s2);
    // 其他成员
};

bool operator<(const StrBlob &s1, const StrBlob &s2)
{
    return *s1.data < *s2..data;
}

bool operator<=(const StrBlob &s1, const StrBlob &s2)
{
    return *s1.data <= *s2..data;
}

bool operator>(const StrBlob &s1, const StrBlob &s2)
{
    return *s1.data > *s2..data;
}

bool operator>=(const StrBlob &s1, const StrBlob &s2)
{
    return *s1.data >= *s2..data;
}

```

两个 StrBlobPtr 的比较，本质上是比较两个指向 vector 内元素的指针（迭代器），因此，首先要求两个 StrBlobPtr 指向相同的 vecotr，否则没有可比性，然后比较指向的位置即可。

```

class StrBlobPtr
{
    friend bool operator<(const StrBlobPtr &s1, const StrBlobPtr &s2);
    friend bool operator<=(const StrBlobPtr &s1, const StrBlobPtr &s2);
    friend bool operator>(const StrBlobPtr &s1, const StrBlobPtr &s2);
    friend bool operator>=(const StrBlobPtr &s1, const StrBlobPtr &s2);
    // 其他成员
};

```

```

bool operator<(const StrBlobPtr &s1, const StrBlobPtr &s2)
{
    auto l = s1.wptr.lock(), r = s2.wptr.lock();
    if (l == r)
        if (!r)
            return false; // 两个指针都为空, 认为是相等
        else
            return (lhs.curr < rhs.curr); // 指向相同 vector, 比较指针位置
    else
        return false; // 指向不同 vector 时, 不能比较
}

bool operator<=(const StrBlobPtr &s1, const StrBlobPtr &s2)
{
    auto l = s1.wptr.lock(), r = s2.wptr.lock();
    if (l == r)
        // 都为空, 或指向相同 vector 且前者位置更靠前
        return (!r || lhs.curr <= rhs.curr);
    else
        return false; // 指向不同 vector 时, 不能比较
}

bool operator>(const StrBlobPtr &s1, const StrBlobPtr &s2)
{
    auto l = s1.wptr.lock(), r = s2.wptr.lock();
    if (l == r)
        if (!r)
            return false; // 两个指针都为空, 认为是相等
        else
            return (lhs.curr > rhs.curr); // 指向相同 vector, 比较指针位置
    else
        return false; // 指向不同 vector 时, 不能比较
}

bool operator<=(const StrBlobPtr &s1, const StrBlobPtr &s2)
{
    auto l = s1.wptr.lock(), r = s2.wptr.lock();
    if (l == r)
        // 都为空, 或指向相同 vector 且前者位置更靠后
        return (!r || lhs.curr >= rhs.curr);
    else
        return false; // 指向不同 vector 时, 不能比较
}

```

两个 StrVec 的比较, 需要手工编写代码逐个比较 string。

```

class StrVec
{
    friend bool operator<(const StrVec &s1, const StrVec &s2);
    friend bool operator<=(const StrVec &s1, const StrVec &s2);
    friend bool operator>(const StrVec &s1, const StrVec &s2);
    friend bool operator>=(const StrVec &s1, const StrVec &s2);
    // 其他成员
};

```

```

bool operator<(const StrVec &s1, const StrVec &s2)
{
    for (auto p1=s1.begin(), p2=s2.begin(); p1 != s1.end() && p2 != s2.end(); p1++, p2++)
        if (*p1 < *p2)          // 之前的 string 都相等, 当前 string 更小
            return true;
        else if (*p1 > *p2) // 之前的 string 都相等, 当前 string 更大
            return false;
    // s1 中的所有 string 都与 s2 中的 string 相等, 且更短
    if (p1 == s1.end() && p2 != s2.end())
        return true;
    return false;
}

bool operator<=(const StrVec &s1, const StrVec &s2)
{
    for (auto p1=s1.begin(), p2=s2.begin(); p1 != s1.end() && p2 != s2.end(); p1++, p2++)
        if (*p1 < *p2)          // 之前的 string 都相等, 当前 string 更小
            return true;
        else if (*p1 > *p2) // 之前的 string 都相等, 当前 string 更大
            return false;
    // s1 的所有 string 都与 s2 中的 string 相等, 且更短或一样长
    if (p1 == s1.end())
        return true;
    return false;
}

bool operator>(const StrVec &s1, const StrVec &s2)
{
    for (auto p1=s1.begin(), p2=s2.begin(); p1 != s1.end() && p2 != s2.end(); p1++, p2++)
        if (*p1 < *p2)          // 之前的 string 都相等, 当前 string 更小
            return false;
        else if (*p1 > *p2) // 之前的 string 都相等, 当前 string 更大
            return true;
    // s2 中的所有 string 都与 s1 中的 string 相等, 且更短
    if (p1 != s1.end() && p2 == s2.end())
        return true;
    return false;
}

bool operator>=(const StrVec &s1, const StrVec &s2)
{
    for (auto p1=s1.begin(), p2=s2.begin(); p1 != s1.end() && p2 != s2.end(); p1++, p2++)
        if (*p1 < *p2)          // 之前的 string 都相等, 当前 string 更小
            return false;
        else if (*p1 > *p2) // 之前的 string 都相等, 当前 string 更大
            return true;
    // s2 中的所有 string 都与 s1 中的 string 相等, 且更短或一样长
    if (p2 == s2.end())
        return true;
}

```

```

        return false;
    }
}
```

**练习 14.19:** 你在 7.5.1 节的练习 7.40 ( 第 261 页 ) 中曾经选择并编写了一个类，你认为它应该含有关系运算符吗？如果是，请实现它；如果不是，解释原因。

### 【出题思路】

本题练习实现关系运算符。

### 【解答】

在练习 7.40 中，我们编写了类 Date。对于日期，可以比较其大小，因此需要为它重载关系运算符。

```

class Date
{
    friend bool operator<(const Date &d1, const Date &d2);
    friend bool operator<=(const Date &d1, const Date &d2);
    friend bool operator>(const Date &d1, const Date &d2);
    friend bool operator>=(const Date &d1, const Date &d2);
    // 其他成员
};

bool operator<(const Date &d1, const Date &d2)
{
    return (d1.year < d2.year) || (d1.year == d2.year && d1.month
        < d2.month) || (d1.year == d2.year && d1.month == d2.month
        && d1.day < d2.day);
}

bool operator<=(const Date &d1, const Date &d2)
{
    return (d1 < d2) || (d1 == d2);
}

bool operator>(const Date &d1, const Date &d2)
{
    return !(d1 <= d2);
}

bool operator>=(const Date &d1, const Date &d2)
{
    return (d1 > d2) || (d1 == d2);
}
```

**练习 14.20:** 为你的 Sales\_data 类定义加法和复合赋值运算符。

### 【出题思路】

本题练习实现重载运算符。

### 【解答】

```

class Sales_data
{
```

```

        friend Sales_data operator+(const Sales_data &lhs, const
            Sales_data &rhs);
    public:
        Sales_data& operator+=(const Sales_data &rhs);
        // 其他成员
    };

Sales_data operator+(const Sales_data &lhs, const Sales_data &rhs)
{
    Sales_data sum = lhs;
    sum += rhs;
    return sum;
}

Sales_data& Sales_data::operator+=(const Sales_data &rhs)
{
    units_sold += rhs.units_sold;
    revenue += rhs.revenue;
    return *this;
}

```

**练习 14.21:** 编写 Sales\_data 类的+和+=运算符，使得+执行实际的加法操作，而+=调用+。相比于 14.3 节（第 497 页）和 14.4 节（第 500 页）对这两个运算符的定义，本题的定义有何缺点？试讨论之。

### 【出题思路】

理解重载运算符的不同版本。

### 【解答】

如练习 14.14 解答所述，本题的方式在性能上没有优势，可读性上也不好。

```

class Sales_data
{
    friend Sales_data operator+(const Sales_data &lhs, const
        Sales_data &rhs);
    public:
        Sales_data& operator+=(const Sales_data &rhs);
        // 其他成员
    };

Sales_data operator+(const Sales_data &lhs, const Sales_data &rhs)
{
    Sales_data sum = lhs;
    sum.units_sold += rhs.units_sold;
    sum.revenue += rhs.revenue;
    return sum;
}

Sales_data& Sales_data::operator+=(const Sales_data &rhs)
{
    *this = (*this) + rhs;
}

```

**练习 14.22:** 定义赋值运算符的一个新版本，使得我们能把一个表示 ISBN 的 string 赋给一个 Sales\_data 对象。

#### 【出题思路】

本题练习实现赋值运算符。

#### 【解答】

```
class Sales_data
{
public:
    Sales_data& operator=(const string &isbn);
    // 其他成员
};

Sales_data& Sales_data::operator=(const string &isbn)
{
    bookNo = isbn;
    return *this;
}
```

**练习 14.23:** 为你的 StrVec 类定义一个 initializer\_list 赋值运算符。

#### 【出题思路】

本题练习实现赋值运算符。

#### 【解答】

```
class StrVec
{
public:
    StrVec& operator=(std::initializer_list<std::string> il);
    // 其他成员
};

StrVec& StrVec::operator=(std::initializer_list<std::string> il)
{
    auto data = alloc_n_copy(il.begin(), il.end());
    free();
    elements = data.first;
    first_free = cap = data.second;
    return *this;
}
```

**练习 14.24:** 你在 7.5.1 节的练习 7.40 ( 第 261 页 ) 中曾经选择并编写了一个类，你认为它应该含有拷贝赋值和移动赋值运算符吗？如果是，请实现它们。

#### 【出题思路】

理解拷贝赋值和移动赋值运算符的作用。

#### 【解答】

在练习 7.40 中，我们编写了类 Date，它只有 3 个 int 类型的数据成员，浅拷贝就能满足需求，因此不需要另外定义拷贝赋值和移动赋值运算符。

**练习 14.25:** 上题的这个类还需要定义其他赋值运算符吗？如果是，请实现它们；同时说明运算对象应该是什么类型并解释原因。

### 【出题思路】

理解赋值运算符的作用。

### 【解答】

这完全取决于实际的需求，例如，如果你希望能用 `string` 形式的日期来初始化 `Date`，就需要定义一个接受 `string` 的赋值运算符。

```
class Date
{
public:
    Date& operator=(const string &date);
    // 其他成员
};

Date& Sales_data::operator=(const string &date)
{   // 接受“1970-1-1”形式的日期字符串
    istringstream in(date);
    char ch1, ch2;
    in >> year >> ch1 >> month >> ch2 >> day;
    if (!in || ch1 != '-' || ch2 != '-')
        throw std::invalid_argument("Bad date");
    if (month < 1 || month > 12 || day < 1 || day > 31)
        throw std::invalid_argument("Bad date");
    return *this;
}
```

显然，日期合法性的检查并不完美，读者可以尝试改进。

**练习 14.26:** 为你的 `StrBlob` 类、`StrBlobPtr` 类、`StrVec` 类和 `String` 类定义下标运算符。

### 【出题思路】

本题练习实现下标运算符。

### 【解答】

```
class StrBlob
{
public:
    std::string& operator[](std::size_t n) { return data[n]; }
    const std::string& operator[](std::size_t n) const
    { return data[n]; }
    // 其他成员
};

class StrBlobPtr
{
public:
    std::string& operator[](std::size_t n) { return
(*wptr.lock())[n]; }
    const std::string& operator[](std::size_t n) const { return
(*wptr.lock())[n]; }
}
```

```

        (*wptr.lock())[n]; }

// 其他成员
};

class StrVec
{
public:
    std::string& operator[](std::size_t n) { return elements[n]; }
    const std::string& operator[](std::size_t n) const
    { return elements[n]; }

// 其他成员
};

class String
{
public:
    char& operator[](std::size_t n) { return (char) str[n]; }
    const char& operator[](std::size_t n) const { return (char) str[n]; }

// 其他成员

private:
    char *str;
};

```

**练习 14.27:** 为你的 StrBlobPtr 类添加递增和递减运算符。

#### 【出题思路】

本题练习实现递增和递减运算符。

#### 【解答】

```

class StrBlobPtr
{
public:
    // 前缀
    StrBlobPtr& operator++();
    StrBlobPtr& operator--();

    // 后缀
    StrBlobPtr operator++(int);
    StrBlobPtr operator-(int);

};

StrBlobPtr& StrBlobPtr::operator++()
{
    check(curr, "increment past end of StrBlobPtr");
    ++curr;
    return *this;
}

StrBlobPtr& StrBlobPtr::operator--()
{
    -curr;
    check(-1, "decrement past begin of StrBlobPtr");
}

```

```

        return *this;
    }

StrBlobPtr StrBlobPtr::operator++(int)
{
    StrBlobPtr ret = *this;
    ++(*this);
    return ret;
}

StrBlobPtr StrBlobPtr::operator-(int)
{
    StrBlobPtr ret = *this;
    -(*this);
    return ret;
}

```

**练习 14.28：**为你的 StrBlobPtr 类添加加法和减法运算符，使其可以实现指针的算术运算（参见 3.5.3 节，第 106 页）。

#### 【出题思路】

本题练习实现加法和减法运算符。

#### 【解答】

```

class StrBlobPtr
{
    friend StrBlobPtr operator+(int n);
    friend StrBlobPtr operator-(int n);
    // 其他成员
};

StrBlobPtr StrBlobPtr::operator+(int n)
{
    auto ret = *this;
    ret.curr += n;
    return ret;
}

StrBlobPtr StrBlobPtr::operator-(int n)
{
    auto ret = *this;
    ret.curr -= n;
    return ret;
}

```

**练习 14.29：**为什么不定义 const 版本的递增和递减运算符？

#### 【出题思路】

理解递增和递减运算符。

#### 【解答】

对于++和--运算符，无论它是前缀版本还是后缀版本，都会改变对象本身的值，

因此不能定义成 `const` 的。

**练习 14.30:** 为你的 `StrBlobPtr` 类和在 12.1.6 节练习 12.22 (第 423 页) 中定义的 `ConstStrBlobPtr` 类分别添加解引用运算符和箭头运算符。注意：因为 `ConstStrBlobPtr` 的数据成员指向 `const vector`，所以 `ConstStrBlobPtr` 中的运算符必须返回常量引用。

### 【出题思路】

本题练习实现解引用和箭头运算符。

### 【解答】

```
class StrBlobPtr
{
public:
    std::string& operator*() const
    {
        auto p = check(curr, "dereference past end");
        return (*p)[curr];
    }

    std::string* operator->() const
    {
        return &(this->operator*());
    }

    // 其他成员
};

class ConstStrBlobPtr
{
public:
    const std::string& operator*() const
    {
        auto p = check(curr, "dereference past end.");
        return (*p)[curr];
    }

    const std::string* operator->() const
    {
        return &(this->operator*());
    }

    // 其他成员
};
```

**练习 14.31:** 我们的 `StrBlobPtr` 类没有定义拷贝构造函数、赋值运算符及析构函数，为什么？

### 【出题思路】

理解拷贝控制成员。

**【解答】**

对于 StrBlobPtr 类 它的数据成员有两个，分别是 `weak_ptr<vector<string>>` 和 `size_t` 类型的，前者定义了自己的拷贝构造函数、赋值运算符和析构函数，后者是内置类型，因此默认的拷贝语义即可，无须为 `StrBlobPtr` 定义拷贝构造函数、赋值运算符和析构函数。

**练习 14.32：** 定义一个类令其含有指向 `StrBlobPtr` 对象的指针，为这个类定义重载的箭头运算符。

**【出题思路】**

本题练习定义箭头运算符访问成员对象的数据。

**【解答】**

```
class MyClass
{
public:
    std::string* operator->() const
    {
        return ptr->operator->();
    }

private:
    StrBlobPtr *ptr;
};
```

**练习 14.33：** 一个重载的函数调用运算符应该接受几个运算对象？

**【出题思路】**

理解调用运算符。

**【解答】**

0 个或多个。

**练习 14.34：** 定义一个函数对象类，令其执行 `if-then-else` 的操作：该类的调用运算符接受三个形参，它首先检查第一个形参，如果成功返回第二个形参的值，如果不成功返回第三个形参的值。

**【出题思路】**

本题练习定义调用运算符。

**【解答】**

```
class IfElseThen
{
public:
    IfElseThen() {}
    IfElseThen(int i1, int i2, int i3) : iVal1(i1), iVal2(i2), iVal3(i3) {}

    int operator()(int i1, int i2, int i3)
```

```

    {
        return i1 ? i2 : i3;
    }

private:
    int iVal1, iVal2, iVal3;
};

```

**练习 14.35:** 编写一个类似于 PrintString 的类，令其从 istream 中读取一行输入，然后返回一个表示我们所读内容的 string。如果读取失败，返回空 string。

### 【出题思路】

本题练习定义调用运算符。

### 【解答】

```

class ReadString
{
public:
    ReadString(istream &is = cin) : is(is) {}
    std::string operator()()
    {
        string line;
        if (!getline(is, line))
        {
            line = " ";
        }

        return line;
    }

private:
    istream &is;
};

```

**练习 14.36:** 使用前一个练习定义的类读取标准输入，将每一行保存为 vector 的一个元素。

### 【出题思路】

本题练习使用调用运算符。

### 【解答】

```

void testReadString()
{
    ReadString rs;
    vector<string> vec;
    while (true)
    {
        string line = rs();
        if (!line.empty())
        {

```

```
        vec.push_back(line);
    }
    else
    {
        break;
    }
}
```

**练习 14.37：**编写一个类令其检查两个值是否相等。使用该对象及标准库算法编写程序，令其替换某个序列中具有给定值的所有实例。

【出题思路】

本题练习定义和使用调用运算符。

### 【解答】

```
class IntCompare
{
public:
    IntCompare(int v) : val(v) {}
    bool operator()(int v) { return val == v; }
private:
    int val;
};

int main()
{
    vector<int> vec = {1, 2, 3, 2, 1};
    const int oldValue = 2;
    const int newValue = 200;
    IntCompare icmp(oldValue);
    std::replace_if(vec.begin(), vec.end(), icmp, newValue);

    return 0;
}
```

**练习 14.38:** 编写一个类令其检查某个给定的 string 对象的长度是否与一个阈值相等。使用该对象编写程序，统计并报告在输入的文件中长度为 1 的单词有多少个、长度为 2 的单词有多少个、……、长度为 10 的单词又有多少个。

### 【出题思路】

本题练习定义和使用调用运算符。

### 【解答】

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

class StrLenIs
```

```

{
public:
    StrLenIs(int len) : len(len) {}
    bool operator()(const string &str) { return str.length() == len; }

private:
    int len;
};

void readStr(istream &is, vector<string> &vec)
{
    string word;
    while (is >> word)
    {
        vec.push_back(word);
    }
}

int main()
{
    vector<string> vec;
    readStr(cin, vec);
    const int minLen = 1;
    const int maxLen = 10;
    for (int i = minLen; i <= maxLen; i++)
    {
        StrLenIs slenIs(i);
        cout << "len : " << i << ", cnt : " << count_if(vec.begin(),
            vec.end(), slenIs) << endl;
    }

    return 0;
}

```

**练习 14.39：**修改上一题的程序令其报告长度在 1 至 9 之间的单词有多少个、长度在 10 以上的单词又有多少个。

### 【出题思路】

本题练习定义和使用调用运算符。

### 【解答】

```

class StrLenBetween
{
public:
    StrLenBetween(int minLen, int maxLen) : minLen(minLen),
        maxLen(maxLen) {}
    bool operator()(const string &str)
    {
        return str.length() >= minLen && str.length() <=
            maxLen;
    }

private:

```

```

        int minLen;
        int maxLen;
    };

class StrNotShorterThan
{
public:
    StrNotShorterThan(int len) : minLen(len) {}
    bool operator()(const string & str) { return str.length() >= minLen; }

private:
    int minLen;
};

// 使用上一题目中的函数
extern readStr(istream & is, vector<string> & vec);

int main()
{
    vector<string> vec;
    readStr(cin, vec);

    StrLenBetween slenBetween(1, 9);
    StrNotShorterThan sNotShorterThan(10);
    cout << "len 1~9 : " << count_if(vec.begin(), vec.end(),
        slenBetween) << endl;
    cout << "len >= 10 : " << count_if(vec.begin(), vec.end(),
        sNotShorterThan) << endl;

    return 0;
}

```

**练习 14.40：**重新编写 10.3.2 节（第 349 页）的 biggies 函数，使用函数对象类替换其中的 lambda 表达式。

### 【出题思路】

本题练习定义和使用调用运算符。

### 【解答】

```

class IsShorter
{
public:
    bool operator()(const string &s1, const string &s2)
    {
        return s1.size() < s2.size();
    }
};

class NotShorterThan
{
public:
    NotShorterThan(int len) : minLen(len) {}
    bool operator()(const string &str)

```

```

    {
        return str.size() >= minLen;
    }

private:
    int minLen;
};

class PrintString
{
public:
    void operator()(const string &str)
    {
        cout << str << " ";
    }
};

void biggies(vector<string> &words, vector<string>::size_type sz)
{
    elimDups(words);

    IsShorter is;
    stable_sort(words.begin(), words.end(), is);

    NotShorterThan nst(sz);
    auto wc = find_if(words.begin(), words.end(), nst);

    auto count = words.end() - wc;
    cout << count << " " << make_plural(count, "word", "s") <<
        " of length " << sz << " or longer" << endl;

    PrintString ps;
    for_each(wc, words.end(), ps);
    cout << endl;
}

```

**练习 14.41:** 你认为 C++11 新标准为什么要增加 lambda? 对于你自己来说, 什么情况下会使用 lambda, 什么情况下会使用类?

#### 【出题思路】

本题旨在理解 lambda。

#### 【解答】

在 C++11 中, lambda 是通过匿名的函数对象来实现的, 因此我们可以把 lambda 看作是对函数对象在使用方式上进行的简化。

当代码需要一个简单的函数, 并且这个函数并不会在其他地方被使用时, 就可以使用 lambda 来实现, 此时它所起的作用类似于匿名函数。

但如果这个函数需要多次使用, 并且它需要保存某些状态的话, 使用函数对象则更合适一些。

**练习 14.42：**使用标准库函数对象及适配器定义一条表达式，令其

- (a) 统计大于 1024 的值有多少个。
- (b) 找到第一个不等于 pooh 的字符串。
- (c) 将所有的值乘以 2。

### 【出题思路】

本题练习使用函数对象。

### 【解答】

```
(a) count_if(vec.begin(), vec.end(), bind2nd(greater<int>(), 1024));
(b) find_if(vec.begin(), vec.end(), bind2nd(not_equal_to<string>(),
    "pooh"));
(c) transform(vec.begin(), vec.end(), vec.begin(), bind2nd(multiplies
    <int>(), 2));
```

**练习 14.43：**使用标准库函数对象判断一个给定的 int 值是否能被 int 容器中的所有元素整除。

### 【出题思路】

本题练习使用函数对象。

### 【解答】

```
bool dividedByAll(vector<int> &ivec, int dividend)
{
    return count_if(ivec.begin(), ivec.end(), bind1st(modulus<int>(),
        dividend)) == 0;
}
```

**练习 14.44：**编写一个简单的桌面计算器使其能处理二元运算。

### 【出题思路】

本题练习使用函数对象。

### 【解答】

```
#include <iostream>
#include <map>
#include <algorithm>
using namespace std;

map<string, function<int (int, int)>> binOps = {
    { "+" , plus<int>() },
    { "-" , minus<int>() },
    { "*" , multiplies<int>() },
    { "/" , divides<int>() },
    { "%" , modulus<int>() }
};

int main()
{
    int left, right;
```

```

    string op;
    cin >> a >> op >> b;
    cout << binOps[op](a, b) << endl;

    return 0;
}

```

**练习 14.45:** 编写类型转换运算符将一个 Sales\_data 对象分别转换成 string 和 double，你认为这些运算符的返回值应该是什么？

**【出题思路】**

理解类型转换运算符。

**【解答】**

如果要转换成 string，那么返回值应该是 bookNo。

如果要转换成 double，那么返回值应该是 revenue。

**练习 14.46:** 你认为应该为 Sales\_data 类定义上面两种类型转换运算符吗？应该把它们声明成 explicit 的吗？为什么？

**【出题思路】**

理解类型转换运算符。

**【解答】**

Sales\_data 类不应该定义这两种类型转换运算符，因为对于该类来讲，它包含三个数据成员：bookNo、units\_sold 和 revenue，只有三者在一起才是有效的数据。

但是如果确实想要定义这两个类型转换运算符的话，应该把它们声明成 explicit 的，这样可以防止 Sales\_data 在某些情况下被默认转换成 string 或 double 类型，这有可能导致我们意料之外的运算结果。

**练习 14.47:** 说明下面这两个类型转换运算符的区别。

```

struct Integral {
    operator const int();
    operator int() const;
};

```

**【出题思路】**

理解类型转换运算符。

**【解答】**

前者将对象转换成 const int，在接受 const int 值的地方才能够使用。

后者则将对象转换成 int 值，相对来说更加通用一些。

**练习 14.48:** 你在 7.5.1 节的练习 7.40(第 261 页) 中曾经选择并编写了一个类，你认为它应该含有向 `bool` 的类型转换运算符吗？如果是，解释原因并说明该运算符是否应该是 `explicit` 的；如果不是，也请解释原因。

### 【出题思路】

理解类型转换运算符。

### 【解答】

在练习 7.40 中，我们编写了类 `Date`，它含有 3 个数据成员：`year`、`month` 和 `day`。

我们可以为 `Date` 类提供一个 `bool` 类型的转换运算符，用来检查 3 个数据成员是否都是有效值(比如 `month` 是否介于 1~12 之间，`day` 是否超出了当月的天数)。

`bool` 类型转换运算符应该声明为 `explicit` 的，因为我们是有意要在条件表达式中使用它的。

**练习 14.49:** 为上一题提到的类定义一个转换目标是 `bool` 的类型转换运算符，先不用在意这么做是否应该。

### 【出题思路】

本题练习类型转换运算符。

### 【解答】

```
class Date
{
public:
    explicit operator bool()
    {
        vector<vector<int>> days_per_month = {
            {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
            {31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}
        };

        return 1 <= month && month <= 12 && 1 <= day && day <=
            days_per_month[isLeapYear() ? 1 : 0][month - 1];
    }

    bool isLeapYear()
    {
        return (year % 4 == 0 && year % 100 != 0) || (year %
            400 == 0);
    }

    // 其他成员
};
```

**练习 14.50:** 在初始化 `ex1` 和 `ex2` 的过程中，可能用到哪些类类型的转换序列呢？说明初始化是否正确并解释原因。

```
struct LongDouble {
    LongDouble(double = 0.0);
```

```

        operator double();
        operator float();
    };
LongDouble ldObj;
int ex1 = ldObj;
float ex2 = ldObj;

```

**【出题思路】**

理解类型转换运算符。

**【解答】**

对于 `int ex1 = ldObj;`，它需要把 `LongDouble` 类型转换成 `int` 类型，但是 `LongDouble` 并没有定义对应的类型转换运算符，因此它会尝试使用其他的来转换，其中 `operator double()` 和 `operator float()` 都满足需求。但编译器无法确定哪一个更合适，因此会产生二义性错误：`conversion from 'LongDouble' to 'int' is ambiguous.`

对于 `float ex2 = ldObj;`，它需要把 `LongDouble` 转换成 `float` 类型，而我们恰好定义了对应的类型转换运算符，因此只需要直接调用 `operator float()` 即可。

**练习 14.51：**在调用 `calc` 的过程中，可能用到哪些类型转换序列呢？说明最佳可行函数是如何被选出来的。

```

void calc(int);
void calc(LongDouble);
double dval;
calc(dval);           // 哪个 calc?

```

**【出题思路】**

理解类型转换运算符。

**【解答】**

这里会优先调用 `void calc(int)` 函数。因为 `double` 转换为 `int` 是标准类型转换，而转换为 `LongDouble` 则是转换为用户自定义类型，实际上是调用了转换构造函数，因此前者优先。

**练习 14.52：**在下面的加法表达式中分别选用了哪个 `operator+?` 列出候选函数、可行函数及为每个可行函数的实参执行的类型转换：

```

struct LongDouble {
    // 用于演示的成员 operator+; 在通常情况下+是个非成员
    LongDouble operator+(const SmallInt&);
    // 其他成员与 14.9.2 节(第 521 页)一致
};

LongDouble operator+(LongDouble&, double);
SmallInt si;
LongDouble ld;
ld = si + ld;
ld = ld + si;

```

**【出题思路】**

理解类型转换运算符。

**【解答】**

对于 `ld=si+ld`, 由于 `LongDouble` 不能转换为 `SmallInt`, 因此 `SmallInt` 的成员 `operator+` 和 `friend operator+` 都不可行。

由于 `SmallInt` 不能转换为 `LongDouble`, `LongDouble` 的成员 `operator+` 和非成员 `opeartor+` 也都不可行。

由于 `SmallInt` 可以转换为 `int`, `LongDouble` 可以转换为 `float` 和 `double`, 所以内置的 `opeartor+(int, float)` 和 `operator+(int, double)` 都可行, 会产生二义性。

对于 `ld=ld+si`, 类似上一个加法表达式, 由于 `SmallInt` 不能转换为 `double`, `LongDouble` 也不能转换为 `SmallInt`, 因此 `SmallInt` 的成员 `operator+` 和两个非成员 `operator+` 都不匹配。

`LongDouble` 的成员 `operator+` 可行, 且为精确匹配。

`SmallInt` 可以转换为 `int`, `LongDouble` 可以转换为 `float` 和 `double`, 因此内置的 `opeartor+(float, int)` 和 `operator+(double, int)` 都可行。但它们都需要类型转换, 因此 `LongDouble` 的成员 `operator+` 优先匹配。

**练习 14.53:** 假设我们已经定义了如第 522 页所示的 `SmallInt`, 判断下面的加法表达式是否合法。如果合法, 使用了哪个加法运算符? 如果不合法, 应该怎样修改代码才能使其合法?

```
SmallInt s1;
double d = s1 + 3.14;
```

**【出题思路】**

理解类型转换运算符。

**【解答】**

如上一题所述, 内置的 `operator+(int, double)` 是可行的, 而 `3.14` 可以转换为 `int`, 然后再转换为 `SmallInt`, 所以 `SmallInt` 的成员 `operator+` 也是可行的。两者都需要进行类型转换, 所以会产生二义性。改为: `double d = s1 + SmallInt(3.14);` 即可。

# 第 15 章

## 面向对象程序设计

### 导读

本章介绍了面向对象程序设计的两个重要概念：继承和动态绑定，包括：

- 继承、基类、派生类的基本概念。
- 虚函数和虚基类。
- 继承中的访问控制、类作用域、构造函数和拷贝控制等问题。

本章练习的最重要目的是让读者理解这些基本概念。掌握根据实际问题特点设计合理的类层次的能力。特别是对比较大的例子——文本查询程序进行修改，练习类层次的设计，来更好地掌握这些基本能力。

### 练习 15.1：什么是虚成员？

#### 【出题思路】

熟悉理解虚函数、虚成员的定义。

#### 【解答】

在类中被声明为 `virtual` 的成员，基类希望这种成员在派生类中重定义。除了构造函数外，任意非 `static` 成员都可以为虚成员。

### 练习 15.2：`protected` 访问说明符与 `private` 有何区别？

#### 【出题思路】

区分 `protected` 和 `private` 的访问权限控制的不同之处。

#### 【解答】

`protected` 为受保护的访问标号，`protected` 成员可以被该类的成员、友元

和派生类成员（非友元）访问，而不可以被该类型的普通用户访问。而 `private` 成员只能被基类的成员和友元访问，派生类不能访问。

### 练习 15.3：定义你自己的 `Quote` 类和 `print_total` 函数。

#### 【出题思路】

书中示例，作为基类用于后续练习题。

#### 【解答】

```
class Quote
{
public:
    Quote()=default;
    Quote(const std::string &book,double sales_price):
        bookNo(book),price(sales_price){}
    std::string isbn() const {return bookNo;}
    // 返回给定数量的书籍的销售总额，派生类改写并使用不同的折扣计算方法
    virtual double net_price(std::size_t n) const {return n*price;}
    virtual ~Quote()=default;           // 对析构函数进行动态绑定
private:
    std::string bookNo;                // 书籍的 ISBN 编号
protected:
    double price=0.0;                  // 代表普通状态下不打折的价格
};

double print_total(ostream &os,const Quote &item, size_t n)
{
    // 根据传入 item 形参的对象类型调用 Quote::net_price
    // 或者 Bulk_quote::net_price
    double ret = item.net_price(n);
    os << "ISBN: " << item.isbn() << "# sold: " << n << " total due: "
    << ret << endl;
    return ret;
}
```

### 练习 15.4：下面哪条声明语句是不正确的？请解释原因。

- (a) class Derived : public Derived { ... };
- (b) class Derived : private Base { ... };
- (c) class Derived : public Base;

#### 【出题思路】

熟悉派生类的定义、声明要求。

#### 【解答】

- (a) 错误，不能用类本身作为类的基类。
- (c) 声明类时，不可以包含派生列表。

**练习 15.5:** 定义你自己的 Bulk\_quote 类。

**【出题思路】**

继承方式练习，Quote 作为基类。

**【解答】**

```
class Bulk_quote : public Quote
{
public:
    double net_price( size_t cnt ) const override
    {
        if ( cnt >=min_qty )
            return cnt * ( 1- discount ) * price;
        else
            return cnt * price;
    }
private:
    size_t min_qty;
    double discount;
};
```

**练习 15.6:** 将 Quote 和 Bulk\_quote 的对象传给 15.2.1 节（第 529 页）练习中的 print\_total 函数，检查该函数是否正确。

**【出题思路】**

练习基类和派生类的使用。

**【解答】**

编写简单的主函数，声明 Quote 和 Bulk\_quote 对象，调用 print\_total 即可。

**练习 15.7:** 定义一个类使其实现一种数量受限的折扣策略，具体策略是：当购买书籍的数量不超过一个给定的限量时享受折扣，如果购买量一旦超过了限量，则超出的部分将以原价销售。

**【出题思路】**

本题练习特定策略的类定义。

**【解答】**

```
class Limited_quote : public Quote
{
public:
    double net_price( size_t cnt ) const override
    {
        if ( cnt <=min_qty )
            return cnt * ( 1- discount ) * price;
        else
            return min_qty*(1-discount)*price+(cnt-min_qty) * price;
    }
};
```

```

private:
    size_t min_qty;
    double discount;
};

```

**练习 15.8：**给出静态类型和动态类型的定义。

**【出题思路】**

区分静态类型和动态类型，熟悉其定义内容。

**【解答】**

静态类型在编译时就已经确定了，它是变量声明时的类型或表达式生成的类型；而动态类型则是变量或表达式表示的内存中的对象的类型，动态类型直到运行时才能知道。如：`Quote *pQuote = new Bulk_quote;;`，指针 `pQuote` 的静态类型是 `Quote`，在编译时就已经确定了。但是它的动态类型是 `Bulk_quote`，直到运行时才能知道它指向的是基类还是派生类。如果一个变量非指针也非引用，则它的静态类型和动态类型永远一致。但基类的指针或引用的动态类型可能与其动态类型不一致。

**练习 15.9：**在什么情况下表达式的静态类型可能与动态类型不同？请给出三个静态类型与动态类型不同的例子。

**【出题思路】**

具体举例说明静态类型与动态类型的不同。

**【解答】**

```

Bulk_quote bulk;
Quote *pQuote=&bulk;
Quote &rQuote=bulk;
// 传递给 item 的如果是派生类对象，即是静态类型和动态类型不同的情况
double print_total(ostream &os, const Quote &item, size_t n);

```

**练习 15.10：**回忆我们在 8.1 节（第 279 页）进行的讨论，解释第 284 页中将 `ifstream` 传递给 `Sales_data` 的 `read` 函数的程序是如何工作的。

**【出题思路】**

理解静态类型和动态类型。

**【解答】**

在要求使用基类型对象的地方，可以使用派生类型的对象来代替，是静态类型和动态类型不同的典型例子。

**练习 15.11：**为你的 `Quote` 类体系添加一个名为 `debug` 的虚函数，令其分别显示每个类的数据成员。

**【出题思路】**

虚函数的构造练习。

**【解答】**

```

class Quote
{
public:
    Quote()=default;
    Quote(const std::string &book,double sales_price):
        bookNo(book),price(sales_price){}
    std::string isbn() const {return bookNo;}
    // 返回给定数量的书籍的销售总额，派生类改写并使用不同的折扣计算方法
    virtual double net_price(std::size_t n) const {return n*price;}
    virtual void debug()
    {
        cout<<"bookNo="<<bookNo<<" price="<<price<<endl;
    }
    virtual ~Quote()=default;           // 对析构函数进行动态绑定
private:
    std::string bookNo;               // 书籍的 ISBN 编号
protected:
    double price=0.0;                // 代表普通状态下不打折的价格
};

class Bulk_quote : public Quote
{

public:
    Bulk_quote(const string &book="",double sales_price=0,std::size_t
               qty=0,double disc_rate=0):
        Quote(book,sales_price),min_qty(qty),discount(disc_rate) { }

    double net_price( std::size_t cnt ) const
    {
        if ( cnt > min_qty )
            return cnt * ( 1- discount ) * price;
        else
            return cnt * price;
    }
    virtual void debug()
    {
        Quote::debug();    // bookNo 变量为 private，所以不能直接访问 bookNo
                           // 只能调用基类的 debug() 函数来显示
        cout<<"min_qty="<<min_qty<<" discount="<<discount<<endl;
    }
};

private:
    std::size_t min_qty;
    double discount;
};

```

**练习 15.12：**有必要将一个成员函数同时声明成 `override` 和 `final` 吗？为什么？

**【出题思路】**

熟悉 `override` 和 `final` 说明符的使用场景。

**【解答】**

有必要。

`override`: 在 C++11 新标准中我们可以使用 `override` 关键字来说明派生类中的虚函数。这么做的好处是在使得我们的意图更加清晰即明确地告诉编译器我们想要覆盖掉已存在的虚函数。如果定义了一个函数与基类中的名字相同但是形参列表不同，在不使用 `override` 关键字的时候这种函数定义是合法的，在使用了 `override` 关键字之后这种行为是非法的，编译器会提示出错。

`final`: 如果我们将某个函数定义成 `final`，则不允许后续的派生类来覆盖这个函数，否则会报错。

因此同时将一个成员函数声明成 `override` 和 `final` 能够使我们的意图更加清晰。

**练习 15.13：**给定下面的类，解释每个 `print` 函数的机理：

```
class base {
public:
    string name() { return basename; }
    virtual void print(ostream &os) { os << basename; }
private:
    string basename;
};

class derived : public base {
public:
    void print(ostream &os) { print(os); os << " " << i; }
private:
    int i;
};
```

在上述代码中存在问题吗？如果有，你该如何修改它？

**【出题思路】**

熟悉掌握派生类的应用及工作原理

**【解答】**

问题：没有声明类 `derived` 是从 `base` 派生过来的。

改正如下：

```
struct base
{
    base(string szNm):basename(szNm) {}
    string name()
    {
        return basename;
    }
```

```

virtual void print(ostream &os)
{
    os<<basename;
}
private:
    string basename;
}

struct derived : public base
{
    derived(string szName, int iVal) : base(szName), mem(iVal)()
    void print()
    {
        print(base :: ostream &os);
        os<<"--"<<mem;
    }
private:
    int mem;
}

```

**练习 15.14:** 给定上一题中的类以及下面这些对象, 说明在运行时调用哪个函数:

|                   |                    |                   |
|-------------------|--------------------|-------------------|
| base bobj;        | base *bp1 = &bobj; | base &br1 = bobj; |
| derived dobj;     | base *bp2 = &dobj; | base &br2 = dobj; |
| (a) bobj.print(); | (b) dobj.print();  | (c) bp1->name();  |
| (d) bp2->name();  | (e) br1.print();   | (f) br2.print();  |

### 【出题思路】

熟悉派生类和基类在运行过程中的调用机制。

### 【解答】

- (a) bobj.print(); 用的是基类的 print 函数。
- (b) dobj.print(); 用的是派生类的 print 函数。
- (c) bp1->name(); 用的是基类的 name 函数。
- (d) pb2->name(); 用的是基类的 name 函数。
- (b) br1.print(); 用的是基类的 print 函数。
- (f) br2.print(); 用的是派生类的 print 函数。

**练习 15.15:** 定义你自己的 Disc\_quote 和 Bulk\_quote。

### 【出题思路】

本题练习实现不同折扣策略。

### 【解答】

```

class Disc_quote:public Quote
{
public:
    Disc_quote(const string &book="",double sales_price=0.0,size_t
               qty=0,double disc= 0.0):Quote(book, sales_price),quantity
               (qty),discount(disc){}
    double net_price(size_t cnt) const=0;
}

```

```

protected:
    size_t quantity;
    double discount;
};

class Bulk_quote : public Disc_quote
{
public:
    Bulk_quote(const string &book="",double sales_price=0,size_t
               qty=0,double disc_rate=0):
        Disc_quote(book,sales_price,qty,disc_rate) { }

    double net_price( size_t cnt ) const
    {
        if ( cnt > quantity )
            return cnt * ( 1- discount ) * price;
        else
            return cnt * price;
    }
};

```

**练习 15.16:** 改写你在 15.2.2 节(第 533 页)练习中编写的数量受限的折扣策略，令其继承 Disc\_quote。

### 【出题思路】

本题练习折扣策略函数覆盖操作。

### 【解答】

```

class Limited_quote : public Disc_quote
{
public:
    Limited_quote(const string &book="",double sales_price=0.0,size_t
                  qty=0,double disc_rate=0.0):
        Disc_quote(book,sales_price,qty,disc_rate) { }

    double net_price( size_t cnt ) const override
    {
        if ( cnt <=quantity )
            return cnt * ( 1- discount ) * price;
        else
            return quantity*(1-discount)*price+(cnt-quantity) * price;
    }
};

```

**练习 15.17:** 尝试定义一个 Disc\_quote 的对象，看看编译器给出的错误信息是什么？

### 【出题思路】

不能创建抽象基类的对象。

### 【解答】

在笔者的编译器中，给出的错误信息是：Variable type 'Disc\_quote' is an abstract class。

**练习 15.18：**假设给定了第 543 页和第 544 页的类，同时已知每个对象的类型如注释所示，判断下面的哪些赋值语句是合法的。解释那些不合法的语句为什么不被允许：

```
Base *p = &d1;           // d1 的类型是 Pub_Derv
p = &d2;                 // d2 的类型是 Priv_Derv
p = &d3;                 // d3 的类型是 Prot_Derv
p = &dd1;                // dd1 的类型是 Derived_from_Public
p = &dd2;                // dd2 的类型是 Derived_from_Private
p = &dd3;                // dd3 的类型是 Derived_from_Protected
```

#### 【出题思路】

熟悉不同说明符导致基类与派生类间的不同访问控制。

#### 【解答】

只有 d1 和 dd1 才能够赋值。这是因为：只有当派生类公有地继承基类时，用户代码才能使用派生类向基类的转换；也就是说，如果派生类继承基类的方式是受保护的或者私有的，则用户代码不能使用该转换。

在题中，只有 d1 和 dd1 类是公有的继承基类，故只有它们才能完成向基类的转换。

**练习 15.19：**假设 543 页和 544 页的每个类都有如下形式的成员函数：

```
void memfcn(Base &b) { b = *this; }
```

对于每个类，分别判断上面的函数是否合法。

#### 【出题思路】

熟悉继承的各种运用情况。

#### 【解答】

Derived\_from\_Private: private Priv\_Derv 这个类的函数不合法。

原因如下：

- 无论派生类以什么方式继承基类，派生类的成员函数和友元都能使用派生类向基类的转换；派生类向其直接基类的类型转换对于派生类的成员和函数来说永远是可访问的。

- 如果派生类继承基类的方式是公有的或者受保护的，则派生类的成员和友元可以使用派生类向基类的类型转换；反之，如果派生类继承基类的方式是私有的，则不能使用。

**练习 15.20：**编写代码检验你对前面两题的回答是否正确。

### 【出题思路】

实际编程练习，判断继承运用是否掌握。

### 【解答】

```
#include <iostream>
using namespace std;
class Base
{
public:
    void pub_mem();
protected:
    int prot_mem;
private:
    char priv_mem;
};
struct Pub_Derv:public Base
{
    int f() {return prot_mem;}
    void memfcn(Base &b)
    {
        b=*this;
        cout<<"Pub_Derv"<<endl;
    }
};
struct Priv_Derv:private Base
{
    int f1() const {return prot_mem;}
    void memfcn(Base &b)
    {
        b=*this;
        cout<<"Priv_Derv"<<endl;
    }
};
struct Prot_Derv:protected Base
{
    int f2() {return prot_mem;}
    void memfcn(Base &b)
    {
        b=*this;
        cout<<"Prot_Derv"<<endl;
    }
};
struct Derived_from_Public:public Pub_Derv
{
    int use_base() {return prot_mem;}
    void memfcn(Base &b)
    {
        b=*this;
        cout<<"Derived_from_Public"<<endl;
    }
};
struct Derived_from_Protected:protected Prot_Derv
{
```

```

int use_base()
{
    return prot_mem;
}
void memfcn(Base &b)
{
    b=*this;
    cout<<"Derived_from_Protected"<<endl;
}
};

int main(int argc, const char * argv[])
{
    Pub_Derv d1;
    Priv_Derv d2;
    Prot_Derv d3;
    Derived_from_Public dd1;
    // Derived_from_Private dd2;
    Derived_from_Protected dd3;
    Base base;
    Base *p=new Base;
    p=&d1;           // d1 的类型是 Pub_Derv
    //p=&d2;         // d2 的类型是 Priv_Derv
    //p=&d3;         // d3 的类型是 Prot_Derv
    p=&dd1;          // dd1 的类型是 Derived_from_Public
    //p=&dd2;          // dd2 的类型是 Derived_from_Private
    //p=&dd3;          // dd3 的类型是 Derived_from_Protected

    d1.memfcn(base);
    d2.memfcn(base);
    d3.memfcn(base);
    dd1.memfcn(base);
    // dd2.memfcn(base);
    dd3.memfcn(base);
    return 0;
}

```

代码运行结果：

```

Pub_Derv
Priv_Derv
Prot_Derv
Derived_from_Public
Derived_from_Protected

```

**练习 15.21：**从下面这些一般性抽象概念中任选一个（或者选一个你自己的），将其对应的一组类型组织成一个继承体系：

- (a) 图像文件格式（如 gif、tiff、jpeg、bmp）
- (b) 几何图元（如矩形、圆、球形、锥形）
- (c) C++语言的类型（如类、函数、成员函数）

#### 【出题思路】

练习继承层次构造。

**【解答】**

对(b)中的几何图元组织成一个继承层次：1) 公共基类 Figure，表示几何图元；  
 2) 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。

**练习 15.22：**对于你在上一题中选择的类，为其添加合适的虚函数及公有成员和受保护的成员。

**【出题思路】**

类构造练习。

**【解答】**

```
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, doouble, double);
    virtual double cubage() = 0;
protected:
    double zSize;
}
class Rectangle : public Figure_2D{
public:
    Rectangle(double, double);
    virtual double area();
    virtual double perimeter();
}

class Circle : public Figure_2D{
public:
    Circle (double, double);
    virtual double area();
    virtual double perimeter();
}

class Sphere : public Figure_3D{
public:
    Sphere(double, double, double);
    virtual double cubage();
}
```

```

}

class Cone : public Figure_3D{
public:
    Cone(double, double, double);
    virtual double cubage();
}

```

**练习 15.23:** 假设第 550 页的 D1 类需要覆盖它继承而来的 fcn 函数，你应该如何对其进行修改？如果你修改之后 fcn 匹配了 Base 中的定义，则该节的那些调用语句将如何解析？

**【出题思路】**

虚函数与其作用域的练习。

**【解答】**

1. 将 D1 类的 fcn 函数更改为 int fcn()。
2. p2->fcn(42)，这一条调用语句将会出错。

**练习 15.24:** 哪种类需要虚析构函数？虚析构函数必须执行什么样的操作？

**【出题思路】**

熟悉虚析构函数的知识。

**【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。

虚析构函数可以为空，即不执行任何操作。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。

**练习 15.25:** 我们为什么为 Disc\_quote 定义一个默认构造函数？如果去掉该构造函数的话会对 Bulk\_quote 的行为产生什么影响？

**【出题思路】**

理解基类或派生类的合成拷贝控制的知识。

**【解答】**

因为 Disc\_quote 的默认构造函数会运行 Quote 的默认构造函数，而 Quote 默认构造函数会完成成员的初始化工作。

如果去掉该构造函数的话，Bulk\_quote 的默认构造函数而无法完成 Disc\_quote 的初始化工作。

**练习 15.26:** 定义 `Quote` 和 `Bulk_quote` 的拷贝控制成员，令其与合成的版本行为一致。为这些成员以及其他构造函数添加打印状态的语句，使得我们能够知道正在运行哪个程序。使用这些类编写程序，预测程序将创建和销毁哪些对象。重复实验，不断比较你的预测和实际输出结果是否相同，直到预测完全准确再结束。

### 【出题思路】

本题考查基类和派生类的构造函数与析构函数的调用过程。

### 【解答】

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

class Quote
{
public:
    Quote()=default;
    Quote(const std::string &book="", double sales_price=0.0):
        bookNo(book), price(sales_price)
    {
        cout<<"Quote constructor is running"=><endl;
    }
    std::string isbn() const
    {
        return bookNo;
    }
    // 返回给定数量的书籍的销售总额，派生类改写并使用不同的折扣计算方法
    virtual double net_price(std::size_t n) const
    {
        return n*price;
    }
    virtual void debug()
    {
        cout<<"bookNo="<<bookNo<<" price="<<price<<endl;
    }
    virtual ~Quote()
    {
        cout<<"Quote destructor is running"=><endl;
    }

    friend ostream &operator <<(ostream&,Quote&);

private:
    std::string bookNo;                      // 书籍的 ISBN 编号
protected:
    double price=0.0;                        // 代表普通状态下不打折的价格
};

ostream & operator <<(ostream&os,Quote "e")
{
    os<<"\tUsing operator <<(ostream &,Quote &);"<<endl;
```

```

        return os;
    };

class Bulk_quote:public Quote
{
public:
    Bulk_quote(const string &book="",double sales_price=0.0,size_t
               qty=0,double disc=0.0):
    Quote(book,sales_price),min_qty(qty),discount(disc)
    {
        cout<<"Bulk_constructor is running"=<<endl;
    }
    double net_price(size_t cnt) const
    {
        if ( cnt > min_qty )
            return cnt * ( 1- discount ) * price;
        else
            return cnt * price;
    }
    ~Bulk_quote()
    {
        cout<<"Bulk_quote destructor is running"=<<endl;
    }

private:
    size_t min_qty;
    double discount;
};

ostream &operator<<(ostream &os,Bulk_quote& bq)
{
    os<<"\tUsing operator <<(ostream&,Bulk_quote &)"=<<endl;
    return os;
}

int main(int argc, const char * argv[])
{
    Quote base("C++ Primer",128.0);
    Bulk_quote bulk("Core Python Programming",89,5,0.19);
    cout<<base<<endl;
    cout<<bulk<<endl;

    return 0;
}

```

运行结果如下：

```

Quote constructor is running
Quote constructor is running
Bulk_constructor is running
Using operator <<(ostream &,Quote &);

Using operator <<(ostream&,Bulk_quote &)

Bulk_quote destructor is running
Quote destructor is running
Quote destructor is running

```

**练习 15.27:** 重新定义你的 Bulk\_quote 类，令其继承构造函数。

**【出题思路】**

本题练习继承构造函数的定义。

**【解答】**

```
class Disc_quote:public Quote
{
public:
    Disc_quote(const string &book="",double sales_price=0.0,size_t
    qty=0,double
    disc=0.0):Quote(book,sales_price),quantity(qty),discount(disc){}
    double net_price(size_t cnt) const=0;
protected:
    size_t quantity;
    double discount;
};

class Bulk_quote : public Disc_quote
{
public:
    using Disc_quote::Disc_quote;
    double net_price( size_t cnt ) const
    {
        if ( cnt > quantity )
            return cnt * ( 1- discount ) * price;
        else
            return cnt * price;
    }
};
```

**练习 15.28:** 定义一个存放 Quote 对象的 vector，将 Bulk\_quote 对象传入其中。计算 vector 中所有元素总的 net\_price。

**【出题思路】**

本题是在容器中放置对象的使用练习。

**【解答】**

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

int main()
{
    vector<Quote> itemVec;
    for (size_t i = 0; i != 10; ++i)
    {
        Bulk_quote item("C++ Primer",6,5,0.5);
        itemVec.push_back(item);
    }

    double sum = 0;
```

```

for (vector<Quote>::iterator iter = itemVec.begin(); iter != itemVec.end(); ++iter)
{
    sum += iter -> net_price(10); // 调用 Quote:: net_price
}

cout << sum << endl; // 输出 600
}

```

**练习 15.29:** 再运行一次你的程序，这次传入 `Quote` 对象的 `shared_ptr`。如果这次计算出的总额与之前的程序不一致，解释为什么；如果一致，也请说明原因。

#### 【出题思路】

本题是在容器中放置指针的使用练习，与放置对象的过程结果存在差异。

#### 【解答】

```
vector< shared_ptr<Quote>> itemVec;
```

程序产生的结果会存在差异。因为当通过 `Quote` 类型的对象调用虚函数 `net_price` 时，不实行动态绑定，调用的是 `Quote` 类中定义的版本；而通过 `Quote` 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_quote` 类中定义的版本。

**练习 15.30:** 编写你自己的 `Basket` 类，用它计算上一个练习中交易记录的总价。

#### 【出题思路】

本题是类构造编程练习。

#### 【解答】

```
class Basket {
public:
    // Basket 使用合成的默认构造函数和拷贝控制成员
    void add_item(const std::shared_ptr<Quote> &sale)
    {
        items.insert(sale);
    }
    // 打印每本书的总价和购物篮中所有书的总价
    double total_receipt(std::ostream&) const;
private:
    // 该函数用于比较 shared_ptr, multiset 成员会用到它
    static bool compare(const std::shared_ptr<Quote> &lhs,
                        const std::shared_ptr<Quote> &rhs)
    { return lhs->isbn() < rhs->isbn(); }
    // multiset 保存多个报价，按照 compare 成员排序
    std::multiset<std::shared_ptr<Quote>, decltype(compare)*>
        items{compare};
};

double Basket::total_receipt(ostream &os) const
{
    double sum = 0.0; // 保存实时计算出的总价格
}
```

```

// iter 指向 ISBN 相同的一批元素中的第一个
// upper_bound 返回一个迭代器，该迭代器指向这批元素的最后一个的下一个位置
for (auto iter = items.cbegin();
     iter != items.cend();
     iter = items.upper_bound(*iter)) {
    // 我们知道在当前的 Basket 中至少有一个该关键字的元素
    // 打印该书籍对应的项目
    sum += print_total(os, **iter, items.count(*iter));
}
os << "Total Sale: " << sum << endl; // 打印最终的总价格
return sum;
}

```

**练习 15.31：**已知 s1、s2、s3 和 s4 都是 string，判断下面的表达式分别创建了什么样的对象：

- (a) Query(s1) | Query(s2) & ~Query(s3);
- (b) Query(s1) | (Query(s2) & ~Query(s3));
- (c) (Query(s1) & (Query(s2) | (Query(s3) & Query(s4))));

#### 【出题思路】

熟悉对象创建工作机理。

#### 【解答】

(a) 共创建 12 个对象：6 个 Query\_base 对象以及其相关联的句柄。6 个 Query\_base 对象分别是 3 个 WordQuery 对象，1 个 NotQuery 对象，1 个 AndQuery 对象，1 个 OrQuery 对象。

(b) 与(a)同。

(c) 共创建 14 个对象：7 个 Query\_base 对象及其相关联的句柄。7 个 Query\_base 对象分别是 4 个 WordQuery 对象，2 个 AndQuery 对象，1 个 OrQuery 对象。

**练习 15.32：**当一个 Query 类型的对象被拷贝、移动、赋值或销毁时，将分别发生什么？

#### 【出题思路】

理解类层次中的拷贝、移动、赋值和销毁行为。

#### 【解答】

Query 类未定义自己的拷贝/移动控制成员，当进行这些操作时，执行默认语义。而其唯一的数据成员是 Query\_base 的 shared\_ptr，因此，当拷贝、移动、赋值或销毁一个 Query 对象时，会调用 shared\_ptr 的对应控制成员，从而实现多个 Query 对象正确共享一个 query\_base。而 shared\_ptr 的控制成员调用 Query\_base 的控制成员时，由于指向的可能是 Query\_base 的派生类对象，因此可能在类层次中进行相应的拷贝/移动操作，调用 query\_base 的派生类的相应控制成员。

**练习 15.33:** 当一个 `Query_base` 类型的对象被拷贝、移动、赋值或销毁时，将分别发生什么？

**【出题思路】**

理解类层次中的拷贝、移动、赋值和销毁行为。

**【解答】**

`Query_base` 是一个虚基类，不允许直接声明其对象。

当其派生类对象进行这些操作时，会调用 `Query_base` 的相应控制成员。而 `Query_base` 没有定义自己的拷贝/移动控制成员，实际上它没有任何数据成员，无须定义这些操作，因此进行这些操作时，执行默认语义，什么也不会发生。

**练习 15.34:** 针对图 15.3（第 565 页）构建的表达式：

- (a) 列举出在处理表达式的过程中执行的所有构造函数。
- (b) 列举出 `cout << q` 所调用的 `rep`。
- (c) 列举出 `q.eval()` 所调用的 `eval`。

**【出题思路】**

本题要求熟练掌握构造函数及其具体操作内容。

**【解答】**

(a) 处理表达式 `Query("fiery") & Query("bird") | Query("wind")` 所执行的构造函数如下：

```
WordQuery(std::string&)
Query(const std::string&)
WordQuery(std::string&)
Query(const std::string&)
WordQuery(std::string&)
Query(const std::string&)
BinaryQuery(Query,Query,std::string)
AndQuery(Query,Query)
BinaryQuery(Query,Query,std::string)
Query(std::shared_ptr<Query_base> query)
BinaryQuery(Query,Query,std::string)
OrQuery(Query,Query)
Query(std::shared_ptr<Query_base> query)
```

(b) 执行 `cout << q` 各个类的 `rep` 函数的调用次序为：

```
BinaryQuery、Query、WordQuery、Query、BinaryQuery、Query、WordQuery、
Query、WordQuery、BinaryQuery、Query、WordQuery、Query、WordQuery、
BinaryQuery、Query、WordQuery、Query、BinaryQuery、Query、WordQuery、
Query、WordQuery、Query、BinaryQuery、Query、WordQuery、Query、BinaryQuery、
Query、WordQuery、Query、WordQuery
```

(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
```

**练习 15.35:** 实现 Query 类和 Query\_base 类，其中需要定义 rep 而无须定义 eval。

### 【出题思路】

本题是类构造的练习。

### 【解答】

```
class Query {
    // 这些运算符需要访问接受 shared_ptr 的构造函数，而该函数是私有的
    friend Query operator~(const Query &);
    friend Query operator|(const Query&, const Query&);
    friend Query operator&(const Query&, const Query&);

public:
    Query(const std::string&);                                // 构建一个新的 WordQuery
    // 接口函数：调用对应的 Query_base 操作
    QueryResult eval(const TextQuery &t) const
        { return q->eval(t); }
    std::string rep() const { return q->rep(); }

private:
    Query(std::shared_ptr<Query_base> query): q(query) { }
    std::shared_ptr<Query_base> q;
};

class Query_base {
    friend class Query;
protected:
    using line_no = TextQuery::line_no;      // 用于 eval 函数
    virtual ~Query_base() = default;

private:
    // eval 返回与当前 Query 匹配的 QueryResult
    virtual QueryResult eval(const TextQuery&) const = 0;
    // rep 是表示查询的一个 string
    virtual std::string rep() const = 0;
};
```

**练习 15.36:** 在构造函数和 rep 成员中添加打印语句，运行你的代码以检验你对本节第一个练习中 (a)、(b) 两小题的回答是否正确。

### 【出题思路】

检验类层次中成员函数的调用关系。

### 【解答】

在各个类的构造函数和 rep 中添加打印语句即可。注意，Query 类有一个 public 和一个 private 共两个构造函数。

**练习 15.37:** 如果在派生类中含有 shared\_ptr<Query\_base>类型的成员而非 Query 类型的成员，则你的类需要做出怎样的改变？

**【出题思路】**

练习类层次的不同实现方式。

**【解答】**

书中的实现方式是用 `Query` 类封装了 `Query_base` 指针，管理实际查询处理用到的不同 `Query` 类型对象。

如果不使用 `Query` 类，则涉及使用 `Query` 类型的地方，都要改成 `Query_base` 指针。如创建单个词查询时，就必须创建 `WordQuery` 类而不是 `Query` 对象。几个重载的布尔运算符也不能再针对 `Query` 对象，而需针对 `Query_base` 指针，从而复杂的查询请求无法写成目前的简单形式，而需逐个运算完成，将结果赋予 `Query_base` 指针，然后再进行下一步运算。资源管理方面也需要重新设计。

因此，当前的设计仍是最佳方式。

**练习 15.38：**下面的声明合法吗？如果不合法，请解释原因；如果合法，请指出该声明的含义。

```
BinaryQuery a = Query("fiery") & Query("bird");
AndQuery b = Query("fiery") & Query("bird");
OrQuery c = Query("fiery") & Query("bird");
```

**【出题思路】**

理解虚函数和类层次的概念。

**【解答】**

第一条声明不合法，因为 `BinaryQuery` 中的 `eval` 是纯虚函数。

第二条声明不合法，不能将 `Query` 转换为 `AndQuery`。

第三条声明不合法，不能将 `Query` 转换为 `OrQuery`。

**练习 15.39：**实现 `Query` 类和 `Query_base` 类，求图 15.3（第 565 页）中表达式的值并打印相关信息，验证你的程序是否正确。

**【出题思路】**

练习复杂类层次的实现。

**【解答】**

参考书中本节内容和配套网站中的代码即可。

**练习 15.40：**在 `OrQuery` 的 `eval` 函数中，如果 `rhs` 成员返回的是空集将发生什么？如果 `lhs` 是空集呢？如果 `lhs` 和 `rhs` 都是空集又将发生什么？

**【出题思路】**

理解并集的计算过程。

**【解答】**

`OrQuery` 的 `eval` 从 `lhs` 和 `rhs` 获取范围来构造 `set`（或向其插入），而 `set`

的构造和插入操作可以正确处理空范围，因此无论 `lhs` 和 `rhs` 的结果是否为空集，`eval` 都能得到正确结果。

**练习 15.41：**重新实现你的类，这次使用指向 `Query_base` 的内置指针而非 `shared_ptr`。请注意，做出上述改动后你的类将不能再使用合成的拷贝控制成员。

### 【出题思路】

练习在复杂程序中自己实现资源管理。

### 【解答】

关键是在没有了 `shared_ptr` 的帮助后，要为 `Query` 类设计拷贝控制成员，管理内存。

除了将成员 `q` 的类型改为 `Query_base*` 外，还需增加引用计数成员 `int* uc`，并增加拷贝构造函数、拷贝赋值运算符和析构函数。具体方法参考第 13 章的练习即可。当然，其他用到 `q` 的地方也需要进行修改。

```
class Query {
...
public:
    Query(const std::string&); // 构造一个新的 WordQuery
    // 拷贝构造函数
    Query(const Query& query) : q(query.q), uc(query.uc) { ++*uc; }
    Query& operator=(const Query& query); // 拷贝赋值运算符
...
    ~Query();
private:
    Query(Query_base* query): q(query), uc(new int(1)) { }
    Query_base* q;
    int* uc;
};
inline
Query::Query(const std::string &s): q(new WordQuery(s)), uc(new int(1))
{ }

inline
Query::~Query()
{
    if (--*uc == 0) {
        delete q;
        delete uc;
    }
}

inline
Query& Query::operator=(const Query& query)
{
    ++*query.uc;
    if (--*uc == 0) {
        delete q;
        delete uc;
    }
}
```

```

    }
    q = query.q;
    uc = query.uc;
    return *this;
}

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 &operand)
{
    return new NotQuery(operand);
}

```

**练习 15.42：**从下面的几种改进中选择一种，设计并实现它：

- (a) 按句子查询并打印单词，而不再是按行打印。
- (b) 引入一个历史系统，用户可以按编号查阅之前的某个查询，并可以在其中增加内容或者将其与其他查询组合。
- (c) 允许用户对结果做出限制，比如从给定范围的行中挑出匹配的进行显示。

### 【出题思路】

本题练习复杂程序的设计和实现。

### 【解答】

在配套网站中的代码基础上进行如下修改：

- (a) 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。

在 TextQuery.cpp 中将 TextQuery 类的构造函数修改如下：

```

// 读输入文件，将每个句子存储为 lines_of_text 的一个元素
TextQuery::TextQuery(ifstream &is): file(new vector<string>)
{
    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)) // 读完了最后一个句子
{
    file->push_back(sentence);
    int n = file->size() - 1;
    istringstream is(sentence);
    string word;
    while (is >> word) {
        auto &lines = wm[word];
        if (!lines)
            lines.reset(new set<line_no>);
        lines->insert(n);
    }
    sentence.assign("");
    // 将 sentence 清空，准备读下一个句子
}
}

```

此外，将 print 函数中输出提示的"line"改为"sentence"。

(b)为了记录查询历史，可声明一个 vector，每个元素是保存 3 个 string 的定长 array，保存一个查询的 3 个查询词。还需修改 get\_word(s) 和主程序的逻辑，允许用户选择历史查询。当用户输入一个合法的查询编号时，从 vector 对应元素中提取出 3 个查询词，重构 Query，完成查询并输出结果。查询添加内容及多查询组合的功能可类似添加。

```

bool get_word(string &s1)
{
    cout << "enter a word to search for, or q to quit, or h to history: ";
    cin >> s1;
    if (!cin || s1 == "q") return false;
    else return true;
}

int main(int argc, char **argv)
{
    // 读取文件建立映射表
    TextQuery file = get_file(argc, argv);
    vector<array<string, 3>> h;

    // 程序主循环：提示用户输入一个单词，在文件中查找它并打印结果
    while (true) {
        string sought1, sought2, sought3;
        if (!get_word(sought1)) break;
        if (sought1 != "h") {
            cout << "\nenter second and third words: ";
            cin >> sought2 >> sought3;
            // 对给定字符串查找所有出现位置
            Query q = Query(sought1) & Query(sought2)
                | Query(sought3);
            h.push_back({sought1, sought2, sought3});
            cout << "\nExecuting Query for: " << q << endl;
            const auto results = q.eval(file);
            // 打印匹配结果
            print(cout, results);
        } else { // 用户输入了"h"，表示要提取历史查询
    }
}

```

```

cout << "\nenter Query no.: ";
int i;
cin >> i;
if (i < 1 || i > h.size())      // 历史编号合法性检查
    cout << "\nBad Query no." << endl;
else {
    // 提取 3 个查询词，重构查询
    Query q = Query(h[i-1][0]) & Query(h[i-1][1])
        | Query(h[i-1][2]);
    cout << "\nExecuting Query for: " << q << endl;
    const auto results = q.eval(file);
    // 打印匹配结果
    print(cout, results);
}
}
return 0;
}

```

(c) 定义另一个版本的 print，接受两个参数 beg 和 end，指出要输出的行号即可。

```

ostream &print(ostream & os, const QueryResult &qr, int beg, int end)
{
    // 如果找到了单词，打印出现次数及所有出现的行号
    os << qr.sought << " occurs " << qr.lines->size() << " "
    << make_plural(qr.lines->size(), "time", "s") << endl;

    // 打印单词出现的每一行
    for (auto num : *qr.lines) // 对 set 中每个元素
        // 不让用户对从 0 开始的文本行号困惑
        if (num + 1 >= beg && num + 1 <= end)
            os << "\t(line " << num + 1 << ")"
            << *(qr.file->begin() + num) << endl;

    return os;
}

```

# 第 16 章

# 模板与泛型编程

## 导读

本章介绍了模板的相关知识，包括：

- 如何定义模板。
- 模板参数推断过程。
- 重载模板。
- 可变参数模板及模板特例化。

本章的练习着重帮助读者掌握如何利用模板这种语言特性来实现泛型编程，包括理解模板的基本概念、练习定义函数模板和类模板、理解模板参数推断过程、理解重载模板对函数匹配的影响以及理解练习可变参数模板和特例化的设计。

**练习 16.1：**给出实例化的定义。

**【出题思路】**

理解实例化的基本概念。

**【解答】**

当调用一个函数模板时，编译器会利用给定的函数实参来推断模板实参，用此实际实参代替模板参数来创建出模板的一个新的“实例”，也就是一个真正可以调用的函数，这个过程称为实例化。

**练习 16.2：**编写并测试你自己版本的 compare 函数。

**【出题思路】**

本题练习定义和使用函数模板。

**【解答】**

参考书中本节内容的实现即可，配套网站上有完整代码可供对照。

**练习 16.3：**对两个 `Sales_data` 对象调用你的 `compare` 函数，观察编译器在实例化过程中如何处理错误。

**【出题思路】**

理解函数模板对参数类型的要求。

**【解答】**

在 tdm gcc-4.8.1 中，对两个 `Sales_data` 对象调用 `compare` 函数模板，编译器会报告如下错误。原因是 `compare` 是用`<`运算符来比较两个对象的，需要类型 `T` 事先定义`<`运算符。但 `Sales_data` 类并未定义`<`运算符，因此会报告错误。

```
error: no match for 'operator<' (operand types are 'const Sales_data'
and 'const Sales_data') |
```

**练习 16.4：**编写行为类似标准库 `find` 算法的模板。函数需要两个模板类型参数，一个表示函数的迭代器参数，另一个表示值的类型。使用你的函数在一个 `vector<int>` 和一个 `list<string>` 中查找给定值。

**【出题思路】**

本题练习设计函数模板。

**【解答】**

用模板类型参数 `I` 表示迭代器类型，用 `T` 表示值的类型。`find` 算法接受两个类型为 `I` 的参数 `b, e` 表示迭代器，和一个类型为 `T` 的参数 `v` 表示要查找的值。函数遍历范围  $[b, e)$  查找 `v`，因此对 `I` 和 `T` 的要求是 `I` 必须支持`++`运算符和`!=`运算符，来实现遍历，并支持`*`运算符来获取元素值，且`*`运算的结果类型必须为 `T`。当对 `vector<int>` 调用 `find` 时，`I` 被解析为 `vector<int>::iterator`，`T` 被解析为 `int`；当对 `list<string>` 调用 `find` 时，`I` 被解析为 `list<string>::iterator`，`T` 被解析为 `string`。

```
#include <iostream>
#include <string>
#include <vector>
#include <list>

using namespace std;

template <typename I, typename T>
I find(I b, I e, const T &v)
{
    while (b != e && *b != v)
        b++;
    return b;
}
```

```

int main()
{
    vector<int> vi = { 0, 2, 4, 6, 8, 10 };
    list<string> ls = { "Hello", "World", "!" };

    auto iter = find(vi.begin(), vi.end(), 6);
    if (iter == vi.end())
        cout << "can not find 6" << endl;
    else
        cout << "find 6 at position " << iter - vi.begin() << endl;

    auto iter1 = find(ls.begin(), ls.end(), "mom");
    if (iter1 == ls.end())
        cout << "can not find mom" << endl;
    else
        cout << "found mom" << endl;

    return 0;
}

```

**练习 16.5：**为 6.2.4 节（第 195 页）中的 print 函数编写模板版本，它接受一个数组的引用，能处理任意大小、任意元素类型的数组。

### 【出题思路】

本题练习设计多模板参数的函数模板。

### 【解答】

由于希望 print 处理任意大小和任意元素类型的数组，因此需要两个模板参数：T 是类型参数，表示数组元素类型；N 是 size\_t 类型常量，表示数组大小。

```

#include <iostream>
#include <string>

using namespace std;

template <typename T, size_t N>
void print(const T (&a)[N])
{
    for (auto iter = begin(a); iter != end(a); iter++)
        cout << *iter << " ";
    cout << endl;
}

int main()
{
    int a[6] = { 0, 2, 4, 6, 8, 10 };
    string vs[3] = { "Hello", "World", "!" };

    print(a);
    print(vs);

    return 0;
}

```

**练习 16.6:** 你认为接受一个数组实参的标准库函数 begin 和 end 是如何工作的？  
定义你自己版本的 begin 和 end。

### 【出题思路】

本题练习设计 begin 和 end。

### 【解答】

begin 应返回数组首元素指针，因此是 return &a[0]。end 返回尾后指针，因此在 begin 上加上数组大小 N 即可。完成两个函数的编写后，可利用上一题的程序进行验证。

```
template <typename T, size_t N>
const T* my_begin(const T (&a) [N])
{
    return &a[0];
}

template <typename T, size_t N>
const T* my_end(const T (&a) [N])
{
    return &a[0] + N;
}
```

**练习 16.7:** 编写一个 constexpr 模板，返回给定数组的大小。

### 【出题思路】

本题练习设计 constexpr 模板。

### 【解答】

由于数组大小是数组类型的一部分，通过模板参数可以获取，因此在 constexpr 模板中直接返回它即可。

```
#include <iostream>
#include <string>

using namespace std;

template <typename T, size_t N>
constexpr int arr_size(const T (&a) [N])
{
    return N;
}

template <typename T, size_t N>
void print(const T (&a) [N])
{
    for (int i = 0; i < arr_size(a); i++)
        cout << a[i] << " ";
    cout << endl;
}

int main()
{
```

```

int a[6] = { 0, 2, 4, 6, 8, 10 };
string vs[3] = { "Hello", "World", "!" };

print(a);
print(vs);

return 0;
}

```

**练习 16.8：**在第 97 页中的“关键概念”中，我们注意到，C++程序员喜欢使用!=而不喜欢<。解释这个习惯的原因。

#### 【出题思路】

理解泛型编程的一个重点：算法对类型要求决定了算法的适用范围。

#### 【解答】

泛型编程的一个目标就是令算法是“通用的”——适用于不同类型。所有标准库容器都定义了==和!=运算符，但其中只有少数定义了<运算符。因此，尽量使用!=而不是<，可减少你的算法适用容器的限制。

**练习 16.9：**什么是函数模板？什么是类模板？

#### 【出题思路】

理解模板的基本概念。

#### 【解答】

简单来说，函数模板是可以实例化出特定函数的模板，类模板是可以实例化出特定类的模板。从形式上来说，函数模板与普通函数相似，只是要以关键字template开始，后接模板参数列表，类模板与普通类的关系类似。在使用上，编译器会根据调用来为我们推断函数模板的模板参数类型，而使用类模板实例化特定类就必须显式指定模板参数。

**练习 16.10：**当一个类模板实例化时，会发生什么？

#### 【出题思路】

理解类模板的实例化过程。

#### 【解答】

当我们使用一个类模板时，必须显式提供模板实参列表，编译器将它们绑定到模板参数，来替换类模板定义中模板参数出现的地方，这样，就实例化出一个特定的类。我们随后使用的其实是这个特定的类。

**练习 16.11：**下面 List 的定义是错误的。应如何修正它？

```
template <typename elemType> class ListItem;
```

```

template <typename elemType> class List {
public:
    List<elemType>();
    List<elemType>(const List<elemType> &);
    List<elemType>& operator=(const List<elemType> &);
    ~List();
    void insert(ListItem *ptr, elemType value);
private:
    ListItem *front, *end;
};

```

**【出题思路】**

理解类模板不是一个类型。

**【解答】**

我们应该牢记，类模板的名字不是一个类型名，只有实例化后才能形成类型，而实例化总是要提供模板实参的。因此，在上述代码中直接使用 `ListItem` 是错误的，应该使用 `ListItem<elemType>`，这才是一个类型。

这个规则有一个例外，就是在类模板作用域中，可以不提供实参，直接使用模板名，也就是说，上述代码中，类内的 `List<elemType>` 可简化为 `List`。

**练习 16.12：**编写你自己版本的 `Blob` 和 `BlobPtr` 模板，包含书中未定义的多个 `const` 成员。

**【出题思路】**

本题练习定义类模板。

**【解答】**

参考书中本节内容编写即可，`const` 版本和普通版本在实现上没有差别。完成后可参考配套网站上的代码。

**练习 16.13：**解释你为 `BlobPtr` 的相等和关系运算符选择哪种类型的友好关系？

**【出题思路】**

理解对模板如何设定友好关系。

**【解答】**

由于函数模板的实例化只处理特定类型，因此，对于相等和关系运算符，对每个 `BlobPtr` 实例与用相同类型实例化的运算符建立一对一的友好关系即可。

```

template <typename T> class BlobPtr {
    friend bool
    operator==(const BlobPtr<T> &, const BlobPtr<T> &);
    ...
};

```

**练习 16.14：**编写 Screen 类模板，用非类型参数定义 Screen 的高和宽。

**【出题思路】**

本题练习定义类模板。

**【解答】**

类模板有两个非类型参数 H 和 W，表示屏幕的高和宽。这样，成员 height 和 width 就不再需要了。注意，在类外给出成员函数的定义时，需要给出完整的模板实参列表。

头文件如下所示：

```
#include <string>
#include <iostream>

template <int H, int W>
class Screen {
public:
    Screen() : contents(H * W, ' ') {}
    Screen(char c) : contents(H * W, c) {}
    friend class Window_mgr;
    char get() const // 获取光标处的内容
        { return contents[cursor]; } // 隐含是内联的
    inline char get(int, int) const; // 显式指定内联
    Screen &clear(char = bkground);
private:
    static const char bkground = ' ';
public:
    Screen &move(int r, int c); // 随后指定内联
    Screen &set(char);
    Screen &set(int, int, char);
    // 重载 display：普通版本和 const 版本
    Screen &display(std::ostream &os)
        { do_display(os); return *this; }
    const Screen &display(std::ostream &os) const
        { do_display(os); return *this; }
private:
    // 实际完成显示的函数
    void do_display(std::ostream &os) const { os << contents; }
    int cursor = 0;
    std::string contents;
};

template <int H, int W>
Screen<H, W> &Screen<H, W>::clear(char c)
{
    contents = std::string(H*W, c);
    return *this;
}

template <int H, int W> // 可以定义时再指定内联
inline
Screen<H, W> &Screen<H, W>::move(int r, int c)
{
```

```

int row = r * W;                                // 计算行位置
cursor = row + c;                                // 将光标移动到此行指定列
return *this;                                     // 返回当前对象(左值)
}

template <int H, int W>
char Screen<H, W>::get(int r, int c) const // 在类内已声明为内联
{
    int row = r * W;                                // 计算行位置
    return contents[row + c];                      // 将光标移动到此行指定列
}

template <int H, int W>
inline Screen<H, W> &Screen<H, W>::set(char c)
{
    contents[cursor] = c;                          // 将光标处的内容设置为新值
    return *this;                                   // 返回当前对象(左值)
}

template <int H, int W>
inline Screen<H, W> &Screen<H, W>::set(int r, int col, char ch)
{
    contents[r*W + col] = ch;                     // 设置给定位置内容为新值
    return *this;                                   // 返回当前对象(左值)
}

```

主函数如下所示：

```

#include <iostream>
using std::cout; using std::endl;

#include <string>
using std::string;

#include "tScreen.h"

int main()
{
    Screen<5,3> myScreen;
    myScreen.display(cout);
    // 将光标移动到特定位置，并设置其内容
    myScreen.move(4,0).set('#');

    Screen<5,5> nextScreen('X');
    nextScreen.move(4,0).set('#').display(cout);
    cout << "\n";
    nextScreen.display(cout);
    cout << endl;

    const Screen<5,3> blank;
    myScreen.set('#').display(cout); // 调用非 const 版本
    cout << endl;
    blank.display(cout);          // 调用 const 版本
}

```

```

cout << endl;

myScreen.clear('Z').display(cout); cout << endl;
myScreen.move(4, 0);
myScreen.set('#');
myScreen.display(cout); cout << endl;
myScreen.clear('Z').display(cout); cout << endl;

// 由于 temp 类型是 Screen<5, 3>而非 Screen<5, 3>&
Screen<5, 3> temp = myScreen.move(4, 0); // 则返回值被拷贝
temp.set('#'); // 改变 temp 就不会影响 myScreen
myScreen.display(cout);
cout << endl;

return 0;
}

```

**练习 16.15：**为你的 Screen 模板实现输入和输出运算符。Screen 类需要哪些友元（如果需要的话）来令输入和输出运算符正确工作？解释每个友元声明（如果有的话）为什么是必要的。

### 【出题思路】

本题练习定义模板的友元。

### 【解答】

头文件中友元相关的代码如下：

```

template <int H, int W> class Screen;

template<int H, int W>
std::ostream & operator<<(std::ostream &, Screen<H, W>&);

template<int H, int W>
std::istream & operator>>(std::istream &, Screen<H, W>&);

template <int H, int W>
class Screen {
    friend std::ostream & operator<< <H, W>(std::ostream &, Screen<H, W>&);
    friend std::istream & operator>> <H, W>(std::istream &, Screen<H, W>&);
    ...
};

template<int H, int W>
std::ostream & operator<<(std::ostream &os, Screen<H, W>& s) {
    os << s.contents;
    return os;
}

template<int H, int W>
std::istream & operator>>(std::istream &is, Screen<H, W>& s) {

```

```

    string t;
    is >> t;
    s.contents = t.substr(0, H*W);
    return is;
}

```

在主程序中加入如下测试代码：

```

cout << endl;
cin >> myScreen;
cout << myScreen << endl << nextScreen << endl << temp << endl;

```

**练习 16.16：**将 StrVec 类(参见 13.5 节, 第 465 页)重写为模板, 命名为 Vec。

#### 【出题思路】

本题练习模板的定义和使用。

#### 【解答】

类模板有一个类型参数 T, 表示向量的元素类型。在模板定义中, 将原来的 string 用 T 替换即可。类模板编写方式与前几题类似, 没有特别需要注意的地方。读者编写完毕后可与配套网站中的代码对照。

**练习 16.17：**声明为 typename 的类型参数和声明为 class 的类型参数有什么不同(如果有的话)? 什么时候必须使用 typename?

#### 【出题思路】

理解 typename 和 class。

#### 【解答】

当用来声明模板类型参数时, typename 和 class 是完全等价的, 都表明模板参数是一个类型。在 C++最初引入模板时, 是使用 class 的。但为了避免与类(或类模板)定义中的 class 相混淆, 引入了 typename 关键字。从字面上看, typename 还暗示了模板类型参数不必是一个类类型。因此, 现在更建议使用 typename。

如本节所述, typename 还有其他用途, 当在模板类型参数上使用作用域运算符::来访问其成员时, 如 T::value\_type, 在实例化之前可能无法辨别访问的到底是静态成员还是类型成员。对此, C++默认通过::访问的是静态成员。为了指明访问的是类型成员, 需要在名字前使用 typename 关键字, 如 typename T::value\_type(), 表明 value\_type 是类型成员, 这里是创建一个 value\_type 类型的对象, 并进行值初始化。

**练习 16.18：**解释下面每个函数模板声明并指出它们是否非法。更正你发现的每个错误。

- (a) template <typename T, U, typename V> void f1(T, U, V);
- (b) template <typename T> T f2(int &T);

```

(c) inline template <typename T> T foo(T, unsigned int*);
(d) template <typename T> f4(T, T);
(e) typedef char Ctype;
template <typename Ctype> Ctype f5(Ctype a);

```

**【出题思路】**

理解模板参数的作用域等特性。

**【解答】**

- (a) 非法。必须指出 U 是类型参数（用 typename）还是非类型参数。
- (b) 非法。在作用域中，模板参数名不能重用，而这里重用 T 作为函数参数名。
- (c) 非法。在模板定义时才能指定 inline。
- (d) 非法。未指定函数模板返回类型。
- (e) 合法。在模板作用域中，类型参数 Ctype 屏蔽了之前定义的类型别名 Ctype。

**练习 16.19：**编写函数，接受一个容器的引用，打印容器中的元素。使用容器的 size\_type 和 size 成员来控制打印元素的循环。

**【出题思路】**

练习用 typename 指明类型成员。

**【解答】**

我们设定循环变量的类型为容器类型（模板参数）的 size\_type，用容器对象（函数参数）的 size 控制循环的终止条件。在循环体中用 at 来获取容器元素，进行打印。由于 size\_type 是容器的类型成员而非静态数据成员，因此在前面加上 typename 特别指出。

```

#include <iostream>
#include <string>
#include <vector>

using namespace std;

template <typename C>
void print(const C &c)
{
    for (typename C::size_type i = 0; i < c.size(); i++)
        cout << c.at(i) << " ";
    cout << endl;
}

int main()
{
    string s = "Hello!";
    print(s);

    vector<int> vi = { 0, 2, 4, 6, 8 };
    print(vi);

    return 0;
}

```

**【其他解题思路】**

显然,由于使用了 `at` 获取容器元素,本程序不适用于 `list` 和 `forward_list`。可以通过迭代器来遍历容器,但与题意隐含地要求用位置编号控制循环稍有违和。

**练习 16.20:** 重写上一题的函数,使用 `begin` 和 `end` 返回的迭代器来控制循环。

**【出题思路】**

练习定义函数模板,复习用迭代器遍历容器。

**【解答】**

比上一题更为简单,用 `begin` 获取容器首位置迭代器,将判定迭代器是否到尾后迭代器(`end`)作为循环判定条件,在循环中解引用迭代器获得元素值。显然,这种方法的适用范围比上一题的方法更宽,可用于 `list` 和 `forward_list`。

```
#include <iostream>
#include <string>
#include <vector>
#include <list>

using namespace std;

template <typename C>
void print(const C &c)
{
    for (auto i = c.begin(); i != c.end(); i++)
        cout << *i << " ";
    cout << endl;
}

int main()
{
    string s = "Hello!";
    print(s);

    vector<int> vi = { 0, 2, 4, 6, 8 };
    print(vi);

    list<string> ls = { "He1", "lo", "!" };
    print(ls);

    return 0;
}
```

**练习 16.21:** 编写你自己的 `DebugDelete` 版本。

**【出题思路】**

本题练习定义成员模板。

**【解答】**

参考书中本节内容编写即可,配套网站上有完整代码供对照。

**练习 16.22:** 修改 12.3 节(第 430 页)中你的 TextQuery 程序, 令 shared\_ptr 成员使用 DebugDelete 作为它们的删除器(参见 12.1.4 节, 第 415 页)。

### 【出题思路】

本题练习使用自定义删除器。

### 【解答】

只需对 TextQuery.cpp 中 file 的初始化进行修改, 创建一个 DebugDelete 对象作为第二个参数即可:

```
TextQuery::TextQuery(ifstream &is): file(new vector<string>,
    DebugDelete("shared_ptr"))
```

**练习 16.23:** 预测在你的查询主程序中何时会执行调用运算符。如果你的预测和实际不符, 确认你理解了原因。

### 【出题思路】

本题旨在理解删除器的工作机制。

### 【解答】

当 shared\_ptr 的引用计数变为 0, 需要释放资源时, 才会调用删除器进行资源释放。分析查询主程序, runQueries 函数结束时, TextQuery 对象 tq 生命期结束, 此时 shared\_ptr 的引用计数变为 0, 会调用删除器释放资源(string 的 vector), 此时调用运算符被执行, 释放资源, 打印一条信息。由于 runQueries 是主函数最后执行的语句, 因此运行效果是程序结束前最后打印出信息。

编译运行上一题的程序, 观察输出结果是否如此。

**练习 16.24:** 为你的 Blob 模板添加一个构造函数, 它接受两个迭代器。

### 【出题思路】

本题练习定义类模板的成员模板。

### 【解答】

参考书中本节内容编写即可。然后修改主程序, 添加相应的测试代码, 如下:

```
Blob<string> b3(b1.begin(), b1.end());
for(auto p = b3.begin(); p != b3.end(); ++p)
    cout << *p << endl;
```

**练习 16.25:** 解释下面这些声明的含义:

```
extern template class vector<string>;
template class vector<Sales_data>;
```

### 【出题思路】

理解实例化控制。

### 【解答】

第一条语句的 extern 表明不在本文件中生成实例化代码, 该实例化的定义会

在程序的其他文件中。

第二条语句用 `Sales_data` 实例化 `vector`, 在其他文件中可用 `extern` 声明此实例化, 使用此定义。

**练习 16.26:** 假设 `NoDefault` 是一个没有默认构造函数的类, 我们可以显式实例化 `vector<NoDefault>` 吗? 如果不可以, 解释为什么。

#### 【出题思路】

理解显式实例化类模板会实例化所有成员函数。

#### 【解答】

答案是否定的。

原因是, 当我们显式实例化 `vector<NoDefault>` 时, 编译器会实例化 `vector` 的所有成员函数, 包括它接受容器大小参数的构造函数。`vector` 的这个构造函数会使用元素类型的默认构造函数来对元素进行值初始化, 而 `NoDefault` 没有默认构造函数, 从而导致编译错误。

**练习 16.27:** 对下面每条带标签的语句, 解释发生了什么样的实例化(如果有的话)。如果一个模板被实例化, 解释为什么; 如果未实例化, 解释为什么没有。

```
template <typename 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)
}
```

#### 【出题思路】

理解显式实例化。

#### 【解答】

(a)、(b)、(c)和(f)分别发生了 `Stack` 对 `char`、`double`、`int` 和 `string` 的实例化, 因为这些语句都要用到这些实例化的类。

(d)、(e)未发生实例化, 因为在本文件之前的位置已经发生了所需的实例化。

**练习 16.28:** 编写你自己版本的 `shared_ptr` 和 `unique_ptr`。

#### 【出题思路】

本题练习定义复杂的类模板。

#### 【解答】

对于 `shared_ptr` (我们的版本命名为 `SP`)，关于引用计数的管理、拷贝构造函数和拷贝赋值运算符等的设计，参考 `HasPtr` 即可。

对于 `unique_ptr`（我们的版本命名为 `UP`），无须管理引用计数，也不支持拷贝构造函数和拷贝赋值运算符，只需设计 `release` 和 `reset` 等函数实现资源释放即可。

```

    }

    p = rhs.p;                                // 拷贝指针
    use = rhs.use;
    return *this;                             // 返回本对象
}

template <typename T, class... Args>
SP<T> make_SP(Args&&... args)
{
    return SP<T>(new T(std::forward<Args>(args)...));
}

template <typename T>
class UP {
public:
    UP() : p(nullptr) {}
    UP(const UP &) = delete;                  // 禁止拷贝构造函数
    explicit UP(T* pt) : p(pt) {}             // 构造函数
    UP& operator=(const UP&) = delete;       // 禁止拷贝赋值运算符
    ~UP();
    T* release();                            // 交出控制权
    void reset(T *new_p);                   // 释放对象
    T& operator*() { return *p; }           // 解引用运算符
    T& operator*() const { return *p; }      // const 版

private:
    T *p;
};

template <typename T>
UP<T>::~UP()
{
    if (p)                                    // 如果已经分配了空间
        delete p;                           // 释放对象内存
}

template <typename T>
void UP<T>::reset(T *new_p = nullptr)
{
    if (p)
        delete p;                         // 释放对象内存
    p = new_p;                            // 指向新对象
}

template <typename T>
T* UP<T>::release()
{
    T *q = p;
    p = nullptr;                          // 清空指针

    return q;                            // 返回对象指针
}
#endif

```

**练习 16.29** 修改你的 Blob 类，用你自己的 `shared_ptr` 代替标准库中的版本。

### 【出题思路】

本题练习使用类模板。

### 【解答】

对 Blob 类模板，需将 `shared_ptr` 替换为 `SP`，将 `make_shared` 替换为 `make_SP`。可以看出，我们并未完整实现 `shared_ptr` 和 `unique_ptr` 的全部功能，如，未实现`->`运算符。因此，在 Blob 类模板中，我们将使用`->`运算符的地方都改为先使用解引用运算符`*`，再使用`.`运算符。同时可以看到，我们也并没有将所有`->`改为`*`和`.`，这是由于类模板的特性，未使用的成员函数是不实例化的。因此，主函数中未用到的地方不进行修改，程序也能正确编译运行。读者可尝试实现 `shared_ptr` 和 `unique_ptr` 的完整功能，并把 Blob 中未修改的地方也修改过来。Blob 中的修改比较零散，这里不再列出，读者阅读源代码即可。

**练习 16.30:**重新运行你的一些程序，验证你的 `shared_ptr` 类和修改后的 Blob 类。(注意：实现 `weak_ptr` 类型超出了本书范围，因此你不能将 `BlobPtr` 类与你修改后的 Blob 一起使用。)

### 【出题思路】

本题练习使用类模板。

### 【解答】

主程序如下所示，其中测试了使用 `SP` 的 Blob 和 UP。由于不能与 `BlobPtr` 一起使用，因此打印 Blob 所有内容时，使用的是 `at` 获取指定下标元素的方式。

```
#include <string>
using std::string;

#include <iostream>
using std::cout; using std::endl;

#include "sp_Blob.h"

int main()
{
    Blob<string> b1;                                // 空 Blob
    cout << b1.size() << endl;
    { // 新作用域
        Blob<string> b2 = {"a", "an", "the"};
        b1 = b2;                                     // b1 和 b2 共享相同的元素
        b2.push_back("about");
        cout << b1.size() << " " << b2.size() << endl;
    } // b2 被销毁，但它指向的元素不能被销毁
    cout << b1.size() << endl;
    for(size_t i = 0; i < b1.size(); ++i)
        cout << b1.at(i) << " ";
    cout << endl << endl;
```

```

UP<int> u1(new int(42));
cout << *u1 << endl;
UP<int> u2(u1.release());
cout << *u2 << endl;

    return 0;
}

```

**练习 16.31:** 如果我们将 `DebugDelete` 与 `unique_ptr` 一起使用，解释编译器将删除器处理为内联形式的可能方式。

**【出题思路】**

理解 `shared_ptr` 和 `unique_ptr` 使用删除器的方式。

**【解答】**

`shared_ptr` 是运行时绑定删除器，而 `unique_ptr` 则是编译时绑定删除器。`unique_ptr` 有两个模板参数，一个是所管理的对象类型，另一个是删除器类型。因此，删除器类型是 `unique_ptr` 类型的一部分，在编译时就可知道，删除器可直接保存在 `unique_ptr` 对象中。通过这种方式，`unique_ptr` 避免了间接调用删除器的运行时开销，而编译时还可以将自定义的删除器，如 `DebugDelete` 编译为内联形式。

**练习 16.32:** 在模板实参推断过程中发生了什么？

**【出题思路】**

理解模板实参推断。

**【解答】**

对一个函数模板，当我们调用它时，编译器会利用调用中的函数实参来推断其模板参数，这些模板实参实例化出的版本与我们的函数调用应该是最匹配的版本，这个过程就称为模板实参推断。

**练习 16.33:** 指出在模板实参推断过程中允许对函数实参进行的两种类型转换。

**【出题思路】**

理解模板实参推断过程中的类型转换。

**【解答】**

在模板实参推断过程中，如果函数形参的类型使用了模板类型参数，则只允许进行两种类型转换。

1. `const` 转换：可以将一个非 `const` 对象的引用（或指针）传递给一个 `const` 对象（或指针）形参。
2. 数组或函数到指针的转换：如果函数形参不是引用类型，则可以对数组或函数类型的实参应用正常的指针转换。一个数组实参可以转换为一个指向其首元素的

指针。类似的，一个函数实参可以转换为一个该函数类型的指针。

**练习 16.34:** 对下面的代码解释每个调用是否合法。如果合法，T 的类型是什么？如果不合法，为什么？

```
template <class T> int compare(const T&, const T&);  
    (a) compare("hi", "world"); (b) compare("bye", "dad");
```

#### 【出题思路】

理解模板实参推断过程中的类型转换。

#### 【解答】

在模板实参推断过程中，允许数组到指针的转换。但是，如果形参是一个引用，则数组不会转换为一个指针。因此，两个调用都是非法的。

**练习 16.35:** 下面调用中哪些是错误的（如果有的话）？如果调用合法，T 的类型是什么？如果调用不合法，问题何在？

```
template <typename T> T calc(T, int);  
template <typename T> T fcn(T, T);  
double d; float f; char c;  
    (a) calc(c, 'c');      (b) calc(d, f);  
    (c) fcn(c, 'c');     (d) fcn(d, f);
```

#### 【出题思路】

理解有普通类型形参情况下的类型转换。

#### 【解答】

(a) 调用合法。因为 calc 的第二个参数是 int 类型，所以可以进行正常的类型转换，将 char 转换为 int。而 T 被推断为 char。

(b) 调用合法。同(a)，对 f 进行正常的类型转换，将 float 转换为 int，T 被推断为 double。

(c) 调用合法。c 和 'c' 都是 char 类型，T 被推断为 char。

(d) 调用非法。d 为 double 类型，f 为 float 类型，T 无法推断为适配的类型。

注意，上述“合法”的判定都是基于模板实参满足函数对类型的需求的前提下做出的。否则，调用仍然是非法的。例如，如果 calc 中对第一个参数进行了成员访问操作 a.x++，即要求 T 是类类型，且包含成员 x，x 的类型支持++运算符，此时前两个调用就是非法的。

**练习 16.36:** 进行下面的调用会发生什么：

```
template <typename T> f1(T, T);  
template <typename T1, typename T2> f2(T1, T2);  
int i = 0, j = 42, *p1 = &i, *p2 = &j;  
const int *cp1 = &i, *cp2 = &j;  
    (a) f1(p1, p2);      (b) f2(p1, p2);      (c) f1(cp1, cp2);  
    (d) f2(cp1, cp2);    (e) f1(p1, cp1);    (f) f2(p1, cp1);
```

**【出题思路】**

理解函数参数类型涉及多模板参数情况下的类型转换。

**【解答】**

- (a) 调用合法。T 被推断为 int\*。
- (b) 调用合法。T1 和 T2 都被推断为 int \*。
- (c) 调用合法。T 被推断为 const int \*。
- (d) 调用合法。T1 和 T2 都被推断为 const int \*。
- (e) 调用非法。T 被推断为 int \* 或是 const int \* 都不能匹配调用。
- (f) 调用合法。T1 被推断为 int \*, T2 被推断为 const int \*。

**练习 16.37:** 标准库 max 函数有两个参数，它返回实参中的较大者。此函数有一个模板类型参数。你能在调用 max 时传递给它一个 int 和一个 double 吗？如果可以，如何做？如果不可以，为什么？

**【出题思路】**

理解显式指定模板实参。

**【解答】**

可以用一个 int 和一个 double 调用 max，显式指定模板实参即可：

```
auto m = max<double>(1, 2.0);
```

**练习 16.38:** 当我们调用 make\_shared（参见 12.1.1 节，第 401 页）时，必须提供一个显式模板实参。解释为什么需要显式模板实参以及它是如何使用的。

**【出题思路】**

理解显式指定模板实参。

**【解答】**

在调用 make\_shared 时，有时不给出参数，表示进行值初始化。有时给出的参数与维护的动态对象的类型不直接相关，如 make\_shared<string>(10, '9') 创建值为"9999999999"的 string。因此，编译器无法从参数推断模板实参，就需要显式指定模板实参。

**练习 16.39:** 对 16.1.1 节（第 578 页）中的原始版本的 compare 函数，使用一个显式模板实参，使得可以向函数传递两个字符串字面常量。

**【出题思路】**

本题练习显式指定模板实参。

**【解答】**

```
compare<string>("Hello", "World");
```

**练习 16.40：**下面的函数是否合法？如果不合法，为什么？如果合法，对可以传递的实参类型有什么限制（如果有的话）？返回类型是什么？

```
template <typename It>
auto fcn3(It beg, It end) -> decltype(*beg + 0)
{
    // 处理序列
    return *beg; // 返回序列中一个元素的拷贝
}
```

#### 【出题思路】

熟悉尾置返回类型。

#### 【解答】

函数是合法的，但用 `decltype(*beg + 0)` 作为尾置返回类型，导致两个问题：

1. 序列元素类型必须支持+运算符。
2. `*beg + 0` 是右值，因此 `fcn3` 的返回类型被推断为元素类型的常量引用。

**练习 16.41：**编写一个新的 `sum` 版本，它的返回类型保证足够大，足以容纳加法结果。

#### 【出题思路】

本题练习定义尾置返回类型。

#### 【解答】

令 `sum` 有两个模板参数，分别表示两个加法运算对象的类型，当然它们应该是相容的类型。在设计尾置返回类型时，首先计算 `sum` 的两个函数参数的和，然后对它们应用 `decltype` 来获得足以容纳和的返回类型。

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

template <typename T1, typename T2>
auto sum(T1 a, T2 b) -> decltype(a + b)
{
    return a + b;
}

int main()
{
    auto a = sum(1, 1);
    cout << a << " " << sizeof(a) << endl;
    auto b = sum(1, 1.1);
    cout << b << " " << sizeof(b) << endl;
    auto c = sum(1, 1.1f);
    cout << c << " " << sizeof(c) << endl;

    return 0;
}
```

**练习 16.42:** 对下面每个调用，确定 T 和 val 的类型：

```
template <typename T> void g(T&& val);
int i = 0; const int ci = i;
(a) g(i); (b) g(ci); (c) g(i * ci);
```

#### 【出题思路】

理解函数模板的函数参数是右值引用时，如何进行类型推断。

#### 【解答】

(a) T 为 int &，val 的类型为 int &。原因是，当实参为一个左值时，编译器推断 T 为实参的左值引用类型，而非左值类型。而 int& && 在引用折叠规则的作用下，被折叠为 int&。

(b) T 为 const int &，val 的类型为 const int &。原因同上一题。

(c) T 为 int，val 的类型为 int &&。原因是，实参是一个右值，编译器就推断 T 为该右值的类型，因此 val 的类型就是右值类型的右值引用。

一个有意思的问题是，对此题，如何验证你的答案？类似书中本节的例子，我们可以在 g 中声明类型为 T 的局部变量 v，将 val 赋值给它。然后打印 v 和 val 的地址，即可判断 T 为 int & 还是 int。还可通过对 v 赋值来判断 T 是否为 const 的。

**练习 16.43:** 使用上一题定义的函数，如果我们调用 g(i = ci)，g 的模板参数将是什么？

#### 【出题思路】

深入理解类型推断。

#### 【解答】

注意，这里是 g(i = ci)，而不是 g(i == ci)。因此，实参不是一个右值，而是一个左值——i = ci 返回的是左值 i（你可以尝试打印 i 和 i = ci 的左值来验证）。因此，最终 T 被推断为 int &，val 经过引用折叠被确定为 int &。

**练习 16.44:** 使用与第一题中相同的三个调用，如果 g 的函数参数声明为 T (而不是 T&&)，确定 T 的类型。如果 g 的函数参数是 const T& 呢？

#### 【出题思路】

理解函数参数是左值引用时的类型推断。

#### 【解答】

当 g 的函数参数声明为 T 时，表明参数传递是传值的，三个调用情况如下：

(a) T 为 int，val 的类型为 int。

(b) T 为 int，val 的类型为 int。

(c) T 为 int，val 的类型为 int。

当 g 的函数参数声明为 const T& 时，表明可以传递给它任何类型的实参，而 T

的类型推断结果也不会是 `const` 的，因此，三个调用情况如下：

- (a) `T` 为 `int`, `val` 的类型为 `const int &`。
- (b) `T` 为 `int`, `val` 的类型为 `const int &&`。
- (c) `T` 为 `int`, `val` 的类型为 `const int & &`。

**练习 16.45:** 给定下面的模板，如果我们对一个像 `42` 这样的字面常量调用 `g`，解释会发生什么？如果我们对一个 `int` 类型的变量调用 `g` 呢？

```
template <typename T> void g(T&& val) { vector<T> v; }
```

#### 【出题思路】

深入理解模板参数推断。

#### 【解答】

(a) 对 `g(42)`, `T` 被推断为 `int`, `val` 的类型为 `int &&`, 因此 `v` 是整数的 `vector`。

(b) 对 `g(i)`, `T` 被推断为 `int &`, `val` 的类型折叠为 `int &`。因此, `v` 被声明为 `int &` 的 `vector`。回忆一下, `vector` 是如何保存它的元素的？它管理动态内存空间，在其中保存它的元素。这就需要维护指向动态内存空间的指针，此指针的类型应该是指向元素类型的指针。但注意，引用不是对象，没有实际地址，因此不能定义指向引用的指针（参见 2.3.2 节）。因此，不能定义指向 `int &` 的指针，也就不能声明 `int &` 的 `vector`，编译失败。

**练习 16.46:** 解释下面的循环，它来自 13.5 节(第 469 页)中的 `StrVec::reallocation`:

```
for (size_t i = 0; i != size(); ++i)
    alloc.construct(dest++, std::move(*elem++));
```

#### 【出题思路】

理解 `std::move` 的工作原理。

#### 【解答】

此循环将 `elem` 开始的内存中的 `string` 对象移到 `dest` 开始的内存中。

每个循环步中，调用 `construct` 在新内存空间中创建对象。若第二个参数是一个左值，则进行拷贝动作。但在上面的代码中，用 `std::move` 将一个左值转换为右值引用，这样，`construct` 会调用 `string` 的移动构造函数将数据从旧内存空间移动而不是拷贝到新的内存空间中，避免了不必要的数据拷贝操作。

**练习 16.47:** 编写你自己版本的翻转函数，通过调用接受左值引用和右值引用参数的函数来测试它。

#### 【出题思路】

本题练习转发的设计。

#### 【解答】

参考书中本节内容编写，与配套网站中的代码进行对照即可。完整程序如下所

示。读者要特别注意 `flip(g, i, 42)`，可尝试改为 `flip1` 或 `flip2`，编译程序，观察现象。

```
#include <utility>
#include <iostream>
using std::cout; using std::endl;

// 模板接受一个可调用对象和两个参数，将两个参数“翻转”后用来调用给定的可调用对象
template <typename F, typename T1, typename T2>
void flip(F f, T1 &&t1, T2 &&t2)
{
    f(std::forward<T2>(t2), std::forward<T1>(t1));
}

void f(int v1, int &v2)           // 注意，v2 是一个引用
{
    cout << v1 << " " << ++v2 << endl;
}

void g(int &&i, int& j)
{
    cout << i << " " << j << endl;
}

// flip1 实现不完整：顶层 const 和引用都丢掉了
template <typename F, typename T1, typename T2>
void flip1(F f, T1 t1, T2 t2)
{
    f(t2, t1);
}

template <typename F, typename T1, typename T2>
void flip2(F f, T1 &&t1, T2 &&t2)
{
    f(t2, t1);
}

int main()
{
    int i = 0, j = 0, k = 0, l = 0;
    cout << i << " " << j << " " << k << " " << l << endl;

    f(42, i);                  // f 改变其实参 i
    flip1(f, j, 42);           // 通过 flip1 调用 f 不会改变 j
    flip2(f, k, 42);           // 正确：k 被改变了
    g(l, i);
    flip(g, i, 42);            // 正确：第三个参数的右值属性被保留了
    cout << i << " " << j << " " << k << " " << l << endl;

    return 0;
}
```

**练习 16.48：**编写你自己版本的 debug\_rep 函数。

**【出题思路】**

理解模板重载。

**【解答】**

参考书中本节内容编写，与配套网站中的代码进行对照即可。尝试定义宏 OVERCHAR (#define OVERCHAR) 和 SPECIALIZED (#define SPECIALIZED) 来改变源码，重新编译运行程序，观察输出结果的差异（同时定义宏 DEBUG，可以观察函数（模板）被调用的次序，更好地帮助你理解结果差异的原因）。

例如，宏 OVERCHAR 只在 SPECIALIZED 未定义时起作用。程序总会定义接受 string 的非模板函数，若未定义 OVERCHAR，则还会定义接受 char \* 和 const char \* 的非模板函数。因此，当未定义 OVERCHAR 时，主程序中的

```
cout << debug_rep("hi") << endl;
```

会输出（打开了 DEBUG）：

```
const char *    const string & "hi"
```

因为此时最佳匹配是接受 const char \* 的非模板函数，它构造了一个 string，继续调用 debug\_rep，因此又会匹配接受 string 的非模板函数。

```
std::string debug_rep(const char *p)
{
#ifndef DEBUG
    std::cout << "const char *" << "\t";
#endif
    return debug_rep(std::string(p));
}
```

而当定义 OVERCHAR 时，会输出：

```
T* const T& pointer: hi h
```

因为此时最佳匹配是接受 T \* 的模板，它又调用 debug\_rep 打印指针指向的值，又会匹配最通用的模板 const T &。

```
template <typename T> std::string debug_rep(T *p)
{
#ifndef DEBUG
    std::cout << "T*" << "\t";
#endif
    std::ostringstream ret;
    ret << "pointer: " << p; // 打印指针自己的值
    if (p)
        ret << " " << debug_rep(*p); // 打印 p 指向的值
    else
        ret << " null pointer"; // p 为空指针
    return ret.str(); // 返回打印出的内容（保存在 ret 的字符串中）
}
```

而当打开 SPECIALIZED 后，输出为：

```
const string & "hi"
```

原因是，打开 SPECIALIZED 后，不再定义非模板函数，而只会定义模板的特例化版本，此时最佳匹配是接受 const string & 的特例化版本：

```
template <> std::string debug_rep(const std::string &s)
```

```

#endif
{
#ifdef DEBUG
    std::cout << "const string &" << "\t";
#endif
    return ' ' + s + ' ';
}

```

读者可自行分析主程序中其他 `debug_rep` 调用输出的异同。

**练习 16.49:** 解释下面每个调用会发生什么：

```

template <typename T> void f(T);
template <typename T> void f(const T*);
template <typename T> void g(T);
template <typename T> void g(T*);
int i = 42, *p = &i;
const int ci = 0, *p2 = &ci;
g(42); g(p); g(ci); g(p2);
f(42); f(p); f(ci); f(p2);

```

**【出题思路】**

理解模板重载。

**【解答】**

`g(42)` 匹配模板 3，`T` 被推断为 `int`。  
`g(p)` 匹配模板 4，`T` 被推断为 `int`。  
`g(ci)` 匹配模板 3，`T` 被推断为 `int`。  
`g(p2)` 匹配模板 4，`T` 被推断为 `const int`。  
`f(42)` 匹配模板 1，`T` 被推断为 `int`。  
`f(p)` 匹配模板 1，`T` 被推断为 `int*`。  
`f(ci)` 匹配模板 1，`T` 被推断为 `int`。  
`f(p2)` 匹配模板 2，`T` 被推断为 `int`。

**练习 16.50:** 定义上一个练习中的函数，令它们打印一条身份信息。运行该练习中的代码。如果函数调用的行为与你的预期不符，确定你理解了原因。

**【出题思路】**

理解模板重载。

**【解答】**

```

#include <iostream>
#include <typeinfo>
using std::cout; using std::endl;

template <typename T> void f(T a)
{
    cout << "f(" << typeid(T).name() << endl;
}

```

```

template <typename T> void f(const T *a)
{
    cout << "f(const T*), const T*是" << typeid(const T*).name() << endl;
}

template <typename T> void g(T a)
{
    cout << "g(T), T是" << typeid(T).name() << endl;
}

template <typename T> void g(T *a)
{
    cout << "g(T*), T*是" << typeid(T*).name() << endl;
}

int main()
{
    int i = 42, *p = &i;
    const int ci = 0, *p2 = &ci;
    g(42); g(p); g(ci); g(p2);
    f(42); f(p); f(ci); f(p2);

    return 0;
}

```

**练习 16.51:** 调用本节中的每个 `foo`, 确定 `sizeof...(Args)` 和 `sizeof...(rest)` 分别返回什么。

#### 【出题思路】

理解可变参数模板。

#### 【解答】

对 4 个调用, `sizeof...(Args)` 和 `sizeof...(rest)` 分别返回:

```

3 3
2 2
1 1
0 0

```

**练习 16.52:** 编写一个程序验证你对上一题的答案。

#### 【出题思路】

理解可变参数模板。

#### 【解答】

```

#include <iostream>
#include <string>
using std::cout; using std::endl; using std::string;

template <typename T, typename... Args>
void foo(const T &t, const Args& ... rest)
{

```

```

cout << sizeof...(Args) << " "; // 模板类型参数的数目
cout << sizeof...(rest) << endl; // 函数参数的数目
}

int main()
{
    int i = 0; double d = 3.14; string s = "how now brown cow";
    foo(i, s, 42, d); // 包中有三个参数
    foo(s, 42, "hi"); // 包中有两个参数
    foo(d, s); // 包中有一个参数
    foo("hi"); // 空包

    return 0;
}

```

**练习 16.53:** 编写你自己版本的 print 函数，并打印一个、两个及五个实参来测试它，要打印的每个实参都应有不同的类型。

### 【出题思路】

参考书中本节内容编写即可。

### 【解答】

```

#include <iostream>
#include <string>

using namespace std;

// 用来终止递归并打印最后一个元素的函数
// 此函数必须在可变参数版本的 print 定义之前声明
template<typename T>
ostream &print(ostream &os, const T &t)
{
    return os << t << endl; // 包中最后一个元素之后不打印分隔符
}

// 包中除了最后一个元素之外的其他元素都会调用这个版本的 print
template <typename T, typename... Args>
ostream &
print(ostream &os, const T &t, const Args&... rest) // 扩展 Args
{
    os << t << ", "; // 打印第一个实参
    return print(os, rest...); // 扩展 rest，递归打印其他参数
}

int main()
{
    int i = 0;
    string s = "Hello";

    print(cout, i);
    print(cout, i, s);
}

```

```

    print(cout, i, s, 42.1, 'A', "End");
    return 0;
}

```

**练习 16.54:** 如果我们对一个没有<<运算符的类型调用 print，会发生什么？

**【出题思路】**

理解模板对类型参数的要求。

**【解答】**

由于 print 要求函数参数类型支持<<运算符，因此会产生编译错误。

**练习 16.55:** 如果我们的可变参数版本 print 的定义之后声明非可变参数版本，解释可变参数的版本会如何执行。

**【出题思路】**

理解模板对类型参数的要求。

**【解答】**

将非可变参数版本放在可变参数版本之后，也属于“定义可变参数版本时，非可变参数版本声明不在作用域中”的情况。因此，可变参数版本将陷入无限递归。

注意，这里的无限递归并不是运行时的无限递归调用，而是发生在编译时递归的包扩展。例如，调用 print(cout, i, s, 42)，正常的包扩展过程是：

```

print(cout, s, 42)
print(cout, 42)

```

最后一步与非可变参数版本匹配。但当非可变参数版本不在作用域中时，还会继续扩展为：

```
print(cout)
```

这就无法与任何模板匹配了，从而产生编译错误。

**练习 16.56:** 编写并测试可变参数版本的 errorMsg。

**【出题思路】**

理解并练习包扩展。

**【解答】**

参考书中本节内容编写即可。需要注意的是，print 要求参数类型支持<<运算符，因此需为 Sales\_data 定义<<运算符：

```

ostream &operator<<(ostream &os, const Sales_data &item)
{
    os << item.isbn() << " " << item.units_sold << " "
    << item.revenue << " " << item.avg_price();
    return os;
}

```

其他代码限于篇幅略。

**练习 16.57:** 比较你的可变参数版本的 `errorMsg` 和 6.2.6 节（第 198 页）中的 `error_msg` 函数。两种方法的优点和缺点各是什么？

**【出题思路】**

理解可变参数模板。

**【解答】**

相对于 6.2.6 节的版本，本节的版本不要求参数具有相同类型。当不知道实参数目也不知道它们的类型时，本节的版本非常有用。而 6.2.6 节的版本只能适用于相同类型的多个参数的情形，对另一种类型，就要为其编写新的版本，编程工作量大。当然，本节的版本较为复杂，需要更多模板相关的知识来确保代码正确，例如复杂的扩展模式。

**练习 16.58:** 为你的 `StrVec` 类及你为 16.1.2 节（第 591 页）中练习编写的 `Vec` 类添加 `emplace_back` 函数。

**【出题思路】**

本题练习转发参数包。

**【解答】**

参照书中本节内容编写即可。为主程序添加如下测试代码：

```
cout << "emplace " << svec.size() << endl;
svec.emplace_back("End");
svec.emplace_back(3, '!');
print(svec);
```

**练习 16.59:** 假定 `s` 是一个 `string`，解释调用 `svec.emplace_back(s)` 会发生什么。

**【出题思路】**

本题要求理解参数包转发过程。

**【解答】**

由于 `s` 是一个左值，经过包扩展，它将以如下形式传递给 `construct`。

```
std::forward<string>(s)
```

`forward<string>` 的结果类型是 `string&`，因此，`construct` 将得到一个左值引用实参，它继续将此参数传递给 `string` 的拷贝构造函数来创建新元素。

**练习 16.60:** 解释 `make_shared`（参见 12.1.1 节，第 401 页）是如何工作的。

**【出题思路】**

本题要求理解参数包转发过程。

**【解答】**

`make_shared` 的工作过程类似 `emplace_back`。

它接受参数包，经过扩展，转发给 new，作为 vector 的初始化参数。

**练习 16.61：**定义你自己版本的 make\_shared。

**【出题思路】**

本题练习定义参数包转发。

**【解答】**

参考 emplace\_back 的设计，可以很容易地编写出 make\_shared:

```
template <typename T, class... Args>
SP<T> make_SP(Args&&... args)
{
    return SP<T>(new T(std::forward<Args>(args)...));
}
```

在练习 16.30 的主程序的基础上增加如下测试代码即可：

```
// 测试 make_SP
vector<string> vs = {"Hello", "World", "!"};
Blob<string> b3(vs.begin(), vs.end());
cout << b3.size() << endl;
for(size_t i = 0; i < b3.size(); ++i)
    cout << b3.at(i) << " ";
cout << endl << endl;

string as[3] = {"This", "is", "end"};
Blob<string> b4(as, as + 3);
cout << b4.size() << endl;
for(size_t i = 0; i < b4.size(); ++i)
    cout << b4.at(i) << " ";
cout << endl << endl;
```

**练习 16.62：**定义你自己版本的 hash<Sales\_data>，并定义一个 Sales\_data 对象的 unordered\_multiset。将多条交易记录保存到容器中，并打印其内容。

**【出题思路】**

本题练习类模板特例化。

**【解答】**

参考书中本节内容编写即可，配套网站上有完整代码供参考。测试用主程序如下：

```
#include <cstddef>
using std::size_t;

#include <string>
using std::string;

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

#include <unordered_set>
using std::unordered_multiset;
```

```

#include <functional>

#include "Sales_data.h"

using std::hash;

int main()
{
    // 使用 hash<Sales_data> 和 Sales_data 的==运算符
    unordered_multiset<Sales_data> SDset;
    Sales_data item;
    while (cin >> item) {
        SDset.insert(item);
    }
    cout << SDset.size() << endl;
    for (auto sd : SDset)
        cout << sd << endl;

    return 0;
}

```

**练习 16.63:** 定义一个函数模板，统计一个给定值在一个 vector 中出现的次数。测试你的函数，分别传递给它一个 double 的 vector，一个 int 的 vector 以及一个 string 的 vector。

### 【出题思路】

本题练习定义模板函数。

### 【解答】

```

#include <iostream>
#include <vector>
#include <cstring>

using namespace std;

template <typename T>
int occur(vector<T> &vec, const T &v)
{
    int ret = 0;

    for (auto a : vec)
        if (a == v)
            ret++;
    return ret;
}

template <>
int occur(vector<char *> &vec, char * const &v)
{
    int ret = 0;

    for (auto a : vec)

```

```

    if (!strcmp(a, v))
        ret++;
    return ret;
}

int main()
{
    vector<double> vd = { 1.1, 3.14, 2.2, 3.14, 3.3, 4.4};
    cout << occur(vd, 3.14) << endl;

    vector<int> vi = { 0, 1, 2, 3, 4, 5 };
    cout << occur(vi, 0) << endl;

    vector<string> vs = { "Hello", "World", "!" };
    cout << occur(vs, string("end")) << endl;

    vector<char *> vp;
    vp.push_back(new char[6]);
    vp.push_back(new char[6]);
    vp.push_back(new char[2]);
    strcpy(vp[0], "Hello");
    strcpy(vp[1], "World");
    strcpy(vp[2], "!");
    char *w = new char[6];
    strcpy(w, "World");
    cout << occur(vp, w) << endl;
    delete w;
    delete vp[2];
    delete vp[1];
    delete vp[0];

    return 0;
}

```

**练习 16.64:** 为上一题中的模板编写特例化版本来处理 `vector<const char*>`。编写程序使用这个特例化版本。

### 【出题思路】

本题练习定义特例化版本。

### 【解答】

见上一题。

注意，当注释掉特例化版本时，最后一个 `occur` 调用会匹配通用版本，用`==`比较 `char *`，从而无法找到相等的 C 风格字符串。

**练习 16.65:** 在 16.3 节(第 617 页)中我们定义了两个重载的 `debug_rep` 版本，一个接受 `const char*` 参数，另一个接受 `char*` 参数。将这两个函数重写为特例化版本。

### 【出题思路】

本题练习定义特例化版本。

**【解答】**

两个特例化版本如下，完整程序见练习 16.48。

```
template<> std::string debug_rep(char *p)
{ return debug_rep(std::string(p)); }
template <> std::string debug_rep(const char *cp)
{ return debug_rep(std::string(cp)); }
```

**练习 16.66：**重载 `debug_rep` 函数与特例化它相比，有何优点和缺点？

**【出题思路】**

理解特例化与重载的区别。

**【解答】**

重载是会影响函数匹配的，也就是说，编译器在函数匹配过程中会将新的重载版本作为候选之一来选择最佳匹配。这就需要小心设计，避免实际匹配不如我们所愿。

特例化则不影响函数匹配，它并不是为编译器进行函数匹配提供一个新的选择，而是为模板的一个特殊实例提供不同于原模板的特殊定义，本质上是接管了编译器在完成函数匹配后的部分实例化工作。即，当某个模板是最佳匹配时，且需要实例化为这个特殊实例时，不再从原模板进行实例化，而是直接使用这个特例化版本。因此，特例化更为简单——当某个模板不符合我们的需求时，只需设计满足需求的特例化版本即可。

**练习 16.67：**定义特例化版本会影响 `debug_rep` 的函数匹配吗？如果不影响，为什么？

**【出题思路】**

理解特例化。

**【解答】**

如上题。

# 第 17 章

## 标准库特殊设施

### 导读

---

本章介绍了 tuple、bitset、正则表达式、随机数和特殊的 IO 操作。  
本章的练习帮助读者熟悉这些标准库设施的使用。

---

**练习 17.1:** 定义一个保存三个 int 值的 tuple，并将其成员分别初始化为 10、20 和 30。

**【出题思路】**

本题练习定义 tuple。

**【解答】**

注意只能直接初始化。

```
tuple<int, int, int> ti{10, 20, 30};
```

**练习 17.2:** 定义一个 tuple，保存一个 string、一个 vector<string> 和一个 pair<string, int>。

**【出题思路】**

本题练习定义 tuple。

**【解答】**

```
tuple<string, vector<string>, pair<string,int>> t;
```

**练习 17.3:** 重写 12.3 节（第 430 页）中的 TextQuery 程序，使用 tuple 替代 QueryResult 类。你认为哪种设计更好？为什么？

**【出题思路】**

在较大的例子中练习定义和使用 tuple。

### 【解答】

首先修改 TextQuery 的定义：

1. 不再包含 QueryResult.h，而是包含 tuple 头文件。
2. 将 line\_no、line\_it 和 print 的声明从 QueryResult.h 拷贝到 t\_TextQuery.h 中。
3. 将 QueryResult 定义为一个 tuple 类型而不再是一个类，tuple 的三项分别是原 QueryResult 类的数据成员。  
.....

```
#include <tuple>

typedef std::vector<std::string>::size_type line_no;
typedef std::set<line_no>::const_iterator line_it;
typedef std::tuple<std::string, std::shared_ptr<std::set<line_no>>, std::shared_ptr<std::vector<std::string>>> QueryResult;
// 这个声明是必需的，查询函数中需返回 QueryResult 类型
class TextQuery {
public:
    TextQuery(std::ifstream&);
    QueryResult query(const std::string&) const;
    void display_map(); // 调试辅助函数：打印映射表
private:
    std::shared_ptr<std::vector<std::string>> file; // 输入文件
    // 将每个单词映射到它出现的行号的集合
    std::map<std::string, std::shared_ptr<std::set<line_no>>> wm;
    // 规范文本：删除标点，并转换为小写
    static std::string cleanup_str(const std::string&);
};

std::ostream &print(std::ostream&, const QueryResult&);
```

然后修改 TextQuery.cpp：

1. TextQuery::query 成员函数虽然返回 QueryResult，但其代码其实不必修改，因为这里构造 QueryResult 使用的是直接初始化方式，改为 tuple 类型后，初始化语句的形式和原来完全一致。
2. 主要修改 print，只有它直接访问了 QueryResult 的成员。对 QueryResult 对象 qr，原来通过 qr.sought、qr.lines 的形式访问检索词和搜索到的行号集合，现在改为 get<0>(qr)、get<1>(qr)这样的形式即可。

```
ostream &print(ostream & os, const QueryResult &qr)
{
    // 如果找到了单词，打印出现次数及所有出现的行号
    os << get<0>(qr) << " occurs " << get<1>(qr)->size() << " "
    << make_plural(get<1>(qr)->size(), "time", "s") << endl;
    // 打印单词出现的每一行
```

```

        for (auto num : *get<1>(qr)) // 对 set 中每个元素
            // 不让用户对从 0 开始的文本行号困惑
            os << "\t(line " << num + 1 << ") "
            << *(get<2>(qr)->begin() + num) << endl;

    return os;
}

```

可以看到，修改比较简单，而使用 tuple 的代码也比定义一个类更为简单。若查询结果只是临时使用，比如输出后即丢弃，则使用 tuple 是一种简单有效的方式。否则，若查询结果还要用来进行其他处理，定义 QueryResult 类是更好的方式。

### 练习 17.4：编写并测试你自己版本的 findBook 函数。

#### 【出题思路】

本题练习用 tuple 返回多个值。

#### 【解答】

参考书中本节内容编写即可，配套网站上有完整代码供对照。编写主程序测试 findBook。

```

int main(int argc, char **argv)
{
    assert(argc > 1);
    // 文件中每个元素保存一个特定书店的销售记录
    vector<vector<Sales_data>> files;
    for (int cnt = 1; cnt != argc; ++cnt)
        files.push_back(build_store(argv[cnt]));

    ifstream in("findbook.in"); // 要搜索的 ISBN 号
    reportResults(in, cout, files);
}

```

它调用 build\_store 来读取命令行参数中指出的书店销售记录文件，存到 files 中。然后通过 reportResults 调用 findBook 来查找指定书目的销售记录。

build\_store 如下：

```

vector<Sales_data> build_store(const string &s)
{
    Sales_data item;
    vector<Sales_data> ret;
    ifstream is(s);
    while (read(is, item))
        ret.push_back(item);
    sort (ret.begin(), ret.end(), lt); // equal_range 要求序列排好序
    return ret;
}

```

注意，为了正确执行 findBook，build\_store 按书目的 isbn 进行了排序，其中 lt 比较两个 Sales\_data 的 isbn。

**练习 17.5:** 重写 `findBook`, 令其返回一个 `pair`, 包含一个索引和一个迭代器 `pair`。

### 【出题思路】

本题练习 `tuple` 替代解决方案。

### 【解答】

相对于 `tuple` 版, 主要修改:

1. `matches` 的类型改为 `pair`, 其中第二个成员还是一个 `pair`, 保存书目的起止迭代器。
2. `findBook` 中构造返回值的部分, 用 `make_pair` 构造一个 `pair`, 第一个参数与 `tuple` 版本一样, 第二个参数直接用 `found`。
3. `reportResults` 从 `findBook` 返回的 `matches` 中提取数据, 进行输出的部分。`get<0>(store)`、`get<1>(store)` 和 `get<2>(store)` 改为 `store.first`、`store.second.first` 和 `store.second.second`。

```
typedef pair<vector<Sales_data>::size_type,
            pair<vector<Sales_data>::const_iterator,
                  vector<Sales_data>::const_iterator>> matches;

ret.push_back(make_pair(it - files.cbegin(), found));

os << "store " << store.first << " sales: "
     << accumulate(store.second.first, store.second.second,
                   Sales_data(s))
     << endl;
```

**练习 17.6:** 重写 `findBook`, 不使用 `tuple` 或 `pair`。

### 【出题思路】

本题练习定义类代替 `tuple` 方式。

### 【解答】

首先定义 `matches` 类, 它有一个 `size_type` 成员和两个迭代器成员, 构造函数接受一个 `size_type` 参数和一个 `pair` 参数来初始化 3 个成员, 另外定义 3 个成员函数分别获取 3 个数据成员。

```
class matches {
public:
    matches(vector<Sales_data>::size_type n,
            pair<vector<Sales_data>::const_iterator,
                  vector<Sales_data>::const_iterator> f)
        : num(n), first(f.first), last(f.second) {}
    vector<Sales_data>::size_type get_num() const { return num; }
    vector<Sales_data>::const_iterator get_first() const { return first; }
    vector<Sales_data>::const_iterator get_last() const { return last; }
private:
    vector<Sales_data>::size_type num;
    vector<Sales_data>::const_iterator first, last;
```

```
};
```

然后修改 reportResults 中访问 3 个数据的方法：

```
os << "store " << store.get_num() << " sales: "
    << accumulate(store.get_first(), store.get_last(),
                  Sales_data(s))
    << endl;
```

**练习 17.7：**解释你更倾向于哪个版本的 findBook，为什么。

**【出题思路】**

理解 pair、tuple 和类实现的差别。

**【解答】**

对于本题，只是简单使用搜索结果，pair 和 tuple 都是简单直接的实现方式。若搜索结果还需进行复杂的计算、处理，定义一个类对其进行封装更好。

**练习 17.8：**在本节最后一段代码中，如果我们将 Sales\_data() 作为第三个参数传递给 accumulate，会发生什么？

**【出题思路】**

复习 Sales\_data 的构造函数的使用。

**【解答】**

Sales\_data() 是 Sales\_data 的默认构造函数，对所有数据成员都采用值初始化，因此 isbn 被初始化为空字符串。因此，在输出结果中，将看不到书目的 isbn。

**练习 17.9：**解释下列每个 bitset 对象所包含的位模式：

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

**【出题思路】**

熟悉 bitset 的定义。

**【解答】**

(a)bitvec 为 32 位，第 5 位为 1，剩余位为 0。

(b)bv 为 32 位，0、2、4、6 这 4 位为 1，剩余位为 0。

(c)bv 为 8 位，用 bstr 来对其进行初始化。若读入的字符串不是单纯的二进制字符串，程序会抛出 invalid\_argument 异常。

**练习 17.10：**使用序列 1、2、3、5、8、13、21 初始化一个 bitset，将这些位置位。对另一个 bitset 进行默认初始化，并编写一小段程序将其恰当的位置位。

**【出题思路】**

练习 bitset 操作。

**【解答】**

构造一个无符号整数，将 1、2、3、5、8、13、21 这 7 位置位，然后用其初始化 bitset。

对于第二小问，用 set 成员函数对默认初始化的 bitset 置位即可。

```
#include <iostream>
#include <bitset>

using namespace std;

int main(int argc, char **argv)
{
    unsigned bp = 2 | (11 << 2) | (1 << 5) | (1 << 8)
        | (1 << 13) | (1 << 21);
    bitset<32> bv(bp);
    cout << bv << endl;

    bitset<32> bvl;
    bvl.set(1); bvl.set(2); bvl.set(3); bvl.set(5);
    bvl.set(8); bvl.set(13); bvl.set(21);
    cout << bvl << endl;

    return 0;
}
```

**练习 17.11：**定义一个数据结构，包含一个整型对象，记录一个包含 10 个问题的真/假测验的解答。如果测验包含 100 道题，你需要对数据结构做出什么改变（如果需要的话）？

**【出题思路】**

本题练习 bitset 的定义和使用。

**【解答】**

如果使用整数保存测验解答，那么对于 10 个问题的测验，只需一个短整型对象即可。如果改为 100 道题，则需 4 个 32 位整数或是 2 个 64 位整数。而且修改的并不仅仅是数据结构，所有对整型数进行操作来修改解答和评分的代码都要相应进行修改，工作量很大。

采用 bitset 则有很明显的劣势，当题目数改变时，我们只需改变 bitset 的规模，而操作 bitset 来完成改答案、评分的代码则只需进行很小的修改。

最佳的方式是定义一个类模板，它有一个模板参数表示题目数，有一个 bitset 成员保存解答，然后定义一些成员函数来完成改答案、评分等操作。当题目数发生变化，我们只需实例化一个新版本即可，其他代码均无须改动。

**练习 17.12：**使用前一题中的数据结构，编写一个函数，它接受一个问题编号和一个表示真/假解答的值，函数根据这两个参数更新测验的解答。

**【出题思路】**

本题练习 `bitset` 的定义和使用。

**【解答】**

此函数直接调用 `bitset` 的 `set` 操作即可，程序见下题。

**练习 17.13：**编写一个整型对象，包含真/假测验的正确答案。使用它来为前两题中的数据结构生成测验成绩。

**【出题思路】**

本题练习 `bitset` 的定义和使用。

**【解答】**

使用一个循环，比较两个 `bitset` 的相同位的数目，即可得到测验成绩。

```
#include <iostream>
#include <bitset>

using namespace std;

template <size_t N>
class exam {
public:
    exam() : s() {}
    size_t get_size() { return N; }
    void set_solution(size_t n, bool b) { s.set(n, b); }
    bitset<N> get_solution() const { return s; }
    size_t score(const bitset<N> &a);
private:
    bitset<N> s;
};

template <size_t N>
size_t exam<N>::score(const bitset<N> &a)
{
    size_t ret = 0;

    for (size_t i = 0; i < N; i++)
        if (s[i] == a[i])
            ret++;

    return ret;
}

int main(int argc, char **argv)
{
    exam<80> e;
    e.set_solution(0, 1);
    e.set_solution(79, 1);
    cout << e.get_solution() << endl;

    bitset<80> a;
    cout << e.get_size() << "题对了"
```

```

    << e.score(a) << "题" << endl;
}

return 0;
}

```

**练习 17.14：**编写几个正则表达式，分别触发不同错误。运行你的程序，观察编译器对每个错误的输出。

#### 【出题思路】

熟悉正则表达式可能出现的错误。

#### 【解答】

首先练习书中本节的例子，然后尝试做其他修改，如：

```
regex r("[[:alnum:]]+\\".(cpp|cxx|cc)$", regex::icase)
```

运行程序后，在 Visual C++ 上会报告如下错误（由于 tdm-gcc 4.8.1 对正则表达式支持不完整，本节程序在 VC 中编译运行）：

```
regex_error(error_ctype): The expression contained an invalid
character class name.
code: 1
```

你会发现不太容易将一个正则表达式改成错误的形式，很多时候，只会改为另一个合法的正则表达式。这也意味着，当设计正则表达式时要特别小心，要保证它符合我们的要求。

**练习 17.15：**编写程序，使用模式查找违反“i 在 e 之前，除非在 c 之后”规则的单词。你的程序应该提示用户输入一个单词，然后指出此单词是否符合要求。用一些违反和未违反规则的单词测试你的程序。

#### 【出题思路】

参考书中本节内容编写即可。

#### 【解答】

完整程序如下。

```

#include <iostream>
#include <string>
#include <regex>

using namespace std;

int main()
{
    // 查找不在字符 c 之后的字符串 ei
    string pattern("[^c]ei");
    // 我们需要包含 pattern 的整个单词
    pattern = "[[:alpha:]]*" + pattern + "[[:alpha:]]*";
    regex r(pattern);      // 构造一个用于查找模式的 regex
    smatch results;        // 定义一个对象保存搜索结果
    // 定义一个 string 保存与模式匹配和不匹配的文本

```

```

string test_str;
while (1) {
    cout << "Enter a word, or q to quit:";
    cin >> test_str;
    if (test_str == "q")
        break;

    // 用 r 在 test_str 中查找与 pattern 匹配的子串
    if (regex_search(test_str, results, r))      // 如果有匹配子串
        cout << results.str() << endl;           // 打印匹配的单词
}

return 0;
}

```

**练习 17.16:** 如果前一题程序中的 regex 对象用 "`[^c]ei`" 进行初始化，将会发生什么？用此模式测试你的程序，检查你的答案是否正确。

#### 【出题思路】

理解正则表达式与字符串的匹配。

#### 【解答】

若 pattern 只是 "`[^c]ei`"，则只与 "`?ei`" 形式的字符串匹配，其中? 是除 c 之外的字符。因此，输出的只是错误拼写的部分，而不是包含错误拼写的完整单词。例如，对 "`friend`" 只输出 "`rie`"。

**练习 17.17:** 更新你的程序，令它查找输入序列中所有违反 "`ei`" 语法规则的单词。

#### 【出题思路】

本题练习利用 regex 迭代器获得多个匹配。

#### 【解答】

参考书中本节内容编写即可。

```

#include <iostream>
#include <string>
#include <regex>

using namespace std;

int main()
{
    // 查找不在字符 c 之后的字符串 ei
    string pattern("[^c]ei");
    // 我们需要包含 pattern 的整个单词
    pattern = "[[:alpha:]]*" + pattern + "[[:alpha:]]*";
    regex r(pattern, regex::icase); // 构造一个用于查找模式的 regex
    smatch results;              // 定义一个对象保存搜索结果

```

```

// 定义一个 string 保存与模式匹配和不匹配的文本
string line;
while (1) {
    cout << "Enter a word, or q to quit:";
    getline(cin, line);
    if (line == "q")
        break;

    // 它将反复调用 regex_search 来寻找文件中的所有匹配
    for (sregex_iterator it(line.begin(), line.end(), rx), end_it;
         it != end_it; ++it)
        cout << it->str() << endl;           // 匹配的单词
}

return 0;
}

```

**练习 17.18:** 修改你的程序, 忽略包含“ei”但并非拼写错误的单词, 如“albeit”和“neighbor”。

**【出题思路】**

本题练习使用多个匹配。

**【解答】**

在使用匹配时(在本题中, 即 for 循环中输出匹配字符串)排除例外情况即可:

```

if (it->str() != "albeit" && it->str() != "neighbor")
    cout << it->str() << endl; // 匹配的单词

```

**练习 17.19:** 为什么可以不先检查 m[4] 是否匹配了就直接调用 m[4].str()?

**【出题思路】**

理解子匹配。

**【解答】**

当子匹配未匹配时, 其 str() 会返回空 string, 仍然可以合法使用。

**练习 17.20:** 编写你自己版本的验证电话号码的程序。

**【出题思路】**

本题练习使用子匹配。

**【解答】**

参考书中本节内容编写即可, 配套网站上有完整代码供对照。

```

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

#include <string>
using std::string;

```

```

#include <regex>
using std::regex; using std::sregex_iterator; using std::smatch;
using std::regex_error;

// 区号部分不包含分隔符
// 剩余部分中的分隔符必须相同
// 或是区号部分有正确的括号，且下一个分隔符是空白或无分隔符

bool valid(const smatch& m)
{
    // 如果区号前是一个左括号
    if(m[1].matched)
        // 则区号后必须是一个右括号
        // 且立即跟随剩余数字或一个空格
        return m[3].matched
            && (m[4].matched == 0 || m[4].str() == " ");
    else
        // 否则，区号后不能有右括号
        // 其他两部分间的分隔符必须相同
        return !m[3].matched
            && m[4].str() == m[6].str();
}

int main()
{
    // 整个正则表达式有 7 个子表达式：
    // (ddd) 分隔符 ddd 分隔符 dddd
    // 子表达式 1、3、4 和 6 是可选的
    // 子表达式 2、5 和 7 匹配号码
    string phone = "(\\d{3})? (\\d{3}) (\\d{4})? ([.-.])? (\\d{3}) ([.-.])? (\\d{4})";
    regex r(phone);      // 查找模式用的 regex
    smatch m;           // 保存匹配结果的对象
    string s;            // 要搜索的字符串

    // 从输入文件读取每条记录
    while (getline(cin, s)) {
        // 对每个匹配的电话号码
        for (sregex_iterator it(s.begin(), s.end(), r), end_it;
             it != end_it; ++it)
            // 检查号码格式是否合法
            if (valid(*it))
                cout << "valid: " << it->str() << endl;
            else
                cout << "not valid: " << it->str() << endl;
    }

    return 0;
}

```

**练习 17.21：**使用本节中定义的 valid 函数重写 8.3.2 节（第 289 页）中你的电话号码程序。

### 【出题思路】

本题练习使用子匹配。

### 【解答】

在第 8 章程序的基础上，将本节的判断号码是否合法的方法移植过去。

需要注意的是，第 8 章的程序使用>>来读取号码，但这会把号码内的空格看作号码的间隔符。因此修改为用 getline 来读取一行，然后用 regex 迭代器在其中匹配字符串，并进行检查、保存。

```
#include <iostream>
using std::cin; using std::cout; using std::cerr;
using std::istream; using std::ostream; using std::endl;

#include <sstream>
using std::ostringstream; using std::istringstream;

#include <vector>
using std::vector;

#include <string>
using std::string;

#include <regex>
using std::regex; using std::smatch; using std::regex_match;
using std::sregex_iterator;

// 成员默认为共有的
struct PersonInfo {
    string name;
    vector<string> phones;
};

// 我们将在第 17 章中看到如何改变电话号码格式
// 现在，简单返回传递来的字符串即可
string format(const string &s) { return s; }

bool valid(const smatch& m)
{
    if(m[1].matched)
        // 区号后必须是一个右括号
        // 且立即跟随剩余数字或一个空格
        return m[3].matched
            && (m[4].matched == 0 || m[4].str() == " ");
    else
        // 区号后不能有右括号
        // 其他两部分间的分隔符必须相同
        return !m[3].matched
            && m[4].str() == m[6].str();
}
```

```

vector<PersonInfo>
getData(istream &is)
{
    string phone = "(\\ \\ ()?(\\ \\ d{3}) (\\ \\))? ([-. ])? (\\ \\d{3}) ([-. ]?)?
(\\ \\d{4})";
    regex r(phone);                      // 查找模式用的 regex
    smatch m;                           // 保存匹配结果的对象

    // 分别从输入读取的一行和一个单词
    string line, word;

    // 保存从输入读取的所有记录
    vector<PersonInfo> people;

    // 逐行读取输入文件，直至遇到行尾（或其他错误）
    while (getline(is, line)) {
        PersonInfo info;                  // 保存此记录数据的对象
        istringstream record(line); // 将字符串输入流绑定到刚读取的一行
        record >> info.name;           // 读取名字
        getline(record, line);
        for (sregex_iterator it(line.begin(), line.end(), r), end_it;
             it != end_it; ++it)        // 匹配电话号码
            if (valid(*it))           // 检查格式合法性并保存
                info.phones.push_back("V" + it->str());
            else info.phones.push_back("I" + it->str());

        people.push_back(info);          // 将此记录追加到 people
    }

    return people;
}

ostream& process(ostream &os, vector<PersonInfo> people)
{
    for (const auto &entry : people) {      // 对 people 中每一项
        // 每个循环步创建两个字符串输出流
        ostringstream formatted, badNums;
        for (const auto &nums : entry.phones) {      // 对每个号码
            if (nums[0] == 'I') {
                // 输出到 badNums
                badNums << " " << nums.substr(1) << endl;
            } else
                // "输出" 格式化的字符串
                formatted << " " << format(nums.substr(1)) << endl;
        }
        if (badNums.str().empty())           // 无不合法的号码
            os << entry.name << endl       // 打印名字
            << formatted.str() << endl; // 和格式化后的号码
        Else                                // 否则，打印名字和错误号码
            cerr << "input error: " << entry.name
    }
}

```

```

        << " invalid number(s) " << badNums.str() << endl;
    }

    return os;
}

int main()
{
    process(cout, getData(cin));

    return 0;
}

```

**练习 17.22:** 重写你的电话号码程序，使之允许在号码的三个部分之间放置任意多个空白符。

#### 【出题思路】

本题练习定义正则表达式及使用子匹配。

#### 【解答】

首先，将正则表达式改为：

"(\\"()?(\\d{3})(\\\")?([-.]|\\\\s\*)?(\\\d{3})([-.]|\\\\s\*)?(\\\d{4})"

表示间隔三部分的可以是一个“.”，一个“-”，或是任意多个空白符 (\s)。

然后要修改 valid 的判断逻辑：

- 若有开始左括号，则第 3 个子匹配必须也匹配（有配对右括号），且第 4 个子匹配不能是分隔符，必须是空白符或是没有。
- 若无开始左括号，则也不能有右括号，且 4、6 两个子匹配必须一样——同为“.”，或同为“-”，或同为空白序列（都不是“.”和“-”）。

```

if(m[1].matched)
    return m[3].matched
    && (m[4].matched == 0 || (m[4].str() != "-" && m[4].str() != "."));
else
    return !m[3].matched
    && ((m[4].str() == "-" && m[6].str() == ".")
        || (m[4].str() == "." && m[6].str() == ".")
        || (m[4].str() != "-" && m[4].str() != "."
            && m[6].str() != "-" && m[6].str() != "."));

```

**练习 17.23:** 编写查找邮政编码的正则表达式。一个美国邮政编码可以由五位或九位数字组成。前五位数字和后四位数字之间可以用一个短横线分隔。

#### 【出题思路】

本题练习设计正则表达式。

#### 【解答】

"(\\\d{5})((-?) (\\\d{4}))?"

**练习 17.24：**编写你自己版本的重排电话号码格式的程序。

**【出题思路】**

本题练习替换匹配字符串。

**【解答】**

参考书中本节内容编写即可。

**练习 17.25：**重写你的电话号码程序，使之只输出每个人的第一个电话号码。

**【出题思路】**

本题练习使用匹配结果。

**【解答】**

利用 regex 迭代器，只取第一个匹配即可。另一种方式是保存所有匹配的号码，但在输出时只输出第一个号码。

具体代码如下所示，将 getData 中的 for 循环改为如下形式：

```
sregex_iterator it(line.begin(), line.end(), r), end_it;
if (it != end_it && valid(*it)) // 匹配电话号码，检查格式并保存
    info.phones.push_back("V" + it->str());
else info.phones.push_back("I" + it->str());
```

**练习 17.26：**重写你的电话号码程序，使之对多于一个电话号码的人只输出第二个和后续电话号码。

**【出题思路】**

本题继续练习使用匹配结果。

**【解答】**

getData 中的循环代码改为如下形式即可：

```
sregex_iterator it(line.begin(), line.end(), r), end_it;
for (it++; it != end_it; it++) // 匹配电话号码，检查格式并保存
    if (valid(*it))
        info.phones.push_back("V" + it->str());
    else info.phones.push_back("I" + it->str());
```

**练习 17.27：**编写程序，将九位数字邮政编码的格式转换为 dddd-dddd。

**【出题思路】**

本题练习替换匹配结果内容。

**【解答】**

如练习 17.23 设计的正则表达式，当第二个子表达式匹配且第一个字符不是-时，将匹配结果替换为"\$1-\$2"：

```
#include <iostream>
#include <string>
#include <regex>
```

```

using namespace std;

int main()
{
    string zip = "(\\d{5})((-?)(\\d{4}))?";
    regex r(zip);           // 寻找模式所用的 regex 对象
    string s;
    string fmt = "$1-$2";   // 将邮政编码格式改为 dddd-dddd
    // 从输入文件中读取每条记录
    while (getline(cin, s)){
        for (sregex_iterator it(s.begin(), s.end(), r), end_it;
             it != end_it; it++)
            if ((*it)[2].matched && (*it)[2].str()[0] != '-')
                cout << (*it).format(fmt) << endl;
            else cout << (*it).str() << endl;
    }

    return 0;
}

```

**练习 17.28:** 编写函数，每次调用生成并返回一个均匀分布的随机 unsigned int。

### 【出题思路】

本题练习生成随机数。

### 【解答】

在函数中定义随机数引擎 e 作为随机数源，定义均匀分布 u 生成指定范围内的均匀分布的随机数。然后调用 u(e) 即可获得一个随机数。注意，将 u 和 e 声明为静态变量以保持状态，使得每次调用函数得到序列中的下一个数。

```

#include <iostream>
#include <random>

using namespace std;

unsigned int rand_int()
{
    // 生成 0 到 9999 之间（包含）均匀分布的随机数
    static uniform_int_distribution<unsigned> u(0, 9999);
    static default_random_engine e; // 生成无符号随机整数

    return u(e);
}

int main()
{
    for (int i = 0; i < 10; i++)
        cout << rand_int() << " ";
    cout << endl;
    return 0;
}

```

**练习 17.29：**修改上一题中编写的函数，允许用户提供一个种子作为可选参数。

**【出题思路】**

本题练习设置种子。

**【解答】**

为函数设置一个可选参数，当实参是非负整数时，用 seed 操作对引擎 e 重新设置种子。主程序三次生成并打印 10 个随机数，后两次分别重置种子为 0 和 19743。编译运行程序，观察结果。

```
#include <iostream>
#include <random>

using namespace std;

unsigned int rand_int(long seed = -1)
{
    // 生成 0 到 9999 之间（包含）均匀分布的随机数
    static uniform_int_distribution<unsigned> u(0, 9999);
    static default_random_engine e; // 生成无符号随机整数

    if (seed >= 0)
        e.seed(seed);
    return u(e);
}

int main()
{
    for (int i = 0; i < 10; i++)
        cout << rand_int() << " ";
    cout << endl;

    cout << rand_int(0) << " ";
    for (int i = 0; i < 9; i++)
        cout << rand_int() << " ";
    cout << endl;

    cout << rand_int(19743) << " ";
    for (int i = 0; i < 9; i++)
        cout << rand_int() << " ";
    cout << endl;

    return 0;
}
```

**练习 17.30：**再次修改你的程序，此次再增加两个参数，表示函数允许返回的最小值和最大值。

**【出题思路】**

本题练习设置随机数的最大值和最小值。

**【解答】**

再增加两个参数表示允许的最大值和最小值，当给定参数合法时（缺省参数不合法），构造一个 uniform\_int\_distribution 对象，生成的随机数最大最小值符合给定参数要求。编译运行程序，观察后两行输出结果。

```
#include <iostream>
#include <random>

using namespace std;

unsigned int rand_int(long seed = -1, long min = 1, long max = 0)
{
    // 生成 0 到 9999 之间（包含）均匀分布的随机数
    static uniform_int_distribution<unsigned> u(0, 9999);
    static default_random_engine e; // 生成无符号随机整数

    if (seed >= 0)
        e.seed(seed);
    if (min <= max)
        u = uniform_int_distribution<unsigned>(min, max);
    return u(e);
}

int main()
{
    for (int i = 0; i < 10; i++)
        cout << rand_int() << " ";
    cout << endl;

    cout << rand_int(0) << " ";
    for (int i = 0; i < 9; i++)
        cout << rand_int() << " ";
    cout << endl;

    cout << rand_int(19743) << " ";
    for (int i = 0; i < 9; i++)
        cout << rand_int() << " ";
    cout << endl;

    cout << rand_int(19743, 0, 9) << " ";
    for (int i = 0; i < 9; i++)
        cout << rand_int() << " ";
    cout << endl;
}

return 0;
}
```

**练习 17.31：**对于本节中的游戏程序，如果我们在 do 循环内定义 b 和 e，会发生什么？

### 【出题思路】

理解随机数的生成。

**【解答】**

在循环内定义 `b` 和 `e`，每个循环步都会用默认的种子（0）重新初始化随机数引擎 `e`。因此，调用 `b(e)` 永远得到的是特定随机数序列的第一个数，游戏的先行者永远是固定的。

而在循环外定义，则可保持引擎的状态，每次得到随机数序列中的下一个值，游戏的先行者会改变。

**练习 17.32：**如果我们在循环内定义 `resp`，会发生什么？

**【出题思路】**

理解变量生命周期。

**【解答】**

如果在循环内定义 `resp`，则其生命周期仅在循环体内，而 `while` 循环条件判定不属于循环体。因此，在进行循环条件判定时，`resp` 已经被销毁，程序会产生编译错误。

**练习 17.33：**修改 11.3.6 节（第 392 页）中的单词转换程序，允许对一个给定单词有多种转换方式，每次随机选择一种进行实际转换。

**【出题思路】**

本题练习随机数的实际使用。

**【解答】**

首先，将 `trans_map` 的类型（包括 `buildMap` 的返回类型）改为 `map<string, vector<string>>`，以便保存一个单词的多种转换方式。

然后，在函数 `transform` 中声明随机数引擎 `e` 为静态局部变量，以便保存状态，每次得到随机数序列中的下一个数。在检查到存在转换规则后，声明一个均匀分布 `u`，其最小值为 0，最大值为转换方法数目减 1，因此得到的随机数对应转换方式的下标。最后调用 `u(e)` 获得随机数，作为下标获取转换的目标字符串。

```
#include <map>
#include <vector>
#include <iostream>
#include <fstream>
#include <string>
#include <stdexcept>
#include <sstream>
#include <vector>
#include <random>
#include <time.h>

using namespace std;

map<string, vector<string>> buildMap(ifstream &map_file)
{
```

```

// 允许多种转换方法
map<string, vector<string>> trans_map;           // 保存转换映射表
string key;   // 要转换的单词
string value;                                       // 替换的内容
// 读取第一个单词存入 key, 一行中剩余内容存入 value
while (map_file >> key && getline(map_file, value))
    if (value.size() > 1)                         // 检查是否有转换规则
        trans_map[key].push_back(value.substr(1)); // 跳过前导空格
    else
        throw runtime_error("no rule for " + key);
return trans_map;
}

const string &
transform(const string &s, const map<string, vector<string>> &m)
{
static default_random_engine e(time(0)); // 随机数引擎, 静态变量保持状态
// 完成真正的转换, 此程序的核心代码
auto map_it = m.find(s);
// 单词在转换映射表中
if (map_it != m.cend()) {
// 随机数分布
uniform_int_distribution<unsigned> u(0, map_it->second.size() - 1);
    return map_it->second[u(e)];                // 随机选择一种转换方式
}
else
    return s;                                     // 否则返回原词
}

// 第一个参数是转换规则文件, 第二个也是要转换的文件
void word_transform(ifstream &map_file, ifstream &input)
{
    auto trans_map = buildMap(map_file);           // 保存转换规则

    // 调试用: 创建转换规则映射表后打印它
    cout << "Here is our transformation map: \n\n";
    for (auto entry : trans_map) {
        cout << "key: " << entry.first
        << "\tvalue: ";
        for (auto s : entry.second)
            cout << s << ", ";
        cout << endl;
    }
    cout << endl << endl;

    // 对给定文本进行转换
    string text;                                    // 保存输入的每一行
    while (getline(input, text)) {                  // 读取输入一行
        istringstream stream(text);                 // 读取每个单词
        string word;
        bool firstword = true;                      // 控制是否打印空格

```

```

while (stream >> word) {
    if (firstword)
        firstword = false;
    else
        cout << " ";
    // transform 返回第一个参数或转换结果
    cout << transform(word, trans_map); // 打印输出
}
cout << endl; // 此行处理完毕
}

int main(int argc, char **argv)
{
    // 打开并检查两个文件
    if (argc != 3)
        throw runtime_error("wrong number of arguments");

    ifstream map_file(argv[1]); // 打开转换规则文件
    if (!map_file) // 检查是否打开成功
        throw runtime_error("no transformation file");

    ifstream input(argv[2]); // 打开要转换的文件
    if (!input) // 检查是否打开成功
        throw runtime_error("no input file");

    word_transform(map_file, input);

    return 0; // 离开 main 函数会自动关闭文件
}

```

**练习 17.34：**编写一个程序，展示如何使用表 17.17 和 17.18 中的每个操纵符。

**【出题思路】**

本题练习格式化输出。

**【解答】**

参考书中本节内容编写即可。

```

#include <iostream>
#include <cmath>
#include <iomanip>

using namespace std;

bool get_status()
{
    return false;
}

int main()
{

```

```

bool b;
cout << "default bool values: " << true << " " << false
<< "\nalpha bool values: " << boolalpha
<< true << " " << false << endl;

bool bool_val = get_status();                         // 设置 cout 的内部状态
cout << boolalpha
<< bool_val
<< noboolalpha;                                     // 重置内部状态为默认格式
cout << endl;

const int ival = 15, jval = 1024; // const, 因此值永远不变

cout << "default: " << 20 << " " << 1024 << endl;
cout << "octal: " << oct << 20 << " " << 1024 << endl;
cout << "hex: " << hex << 20 << " " << 1024 << endl;
cout << "decimal: " << dec << 20 << " " << 1024 << endl;

cout << showbase;                                  // 打印整型值时显示基
cout << "default: " << 20 << " " << 1024 << endl;
cout << "in octal: " << oct << 20 << " " << 1024 << endl;
cout << "in hex: " << hex << 20 << " " << 1024 << endl;
cout << "in decimal: " << dec << 20 << " " << 1024 << endl;
cout << noshowbase;                                // 重置流的状态

cout << 10.0 << endl;                            // 打印 10
cout << showpoint << 10.0                         // 打印 10.0000
<< noshowpoint << endl;                          // 恢复小数点的默认格式

cout << showpos << 10.0 << endl; // 非负数打印+
cout << noshowpos << 10.0 << endl; // 非负数不打印+

cout << uppercase << showbase << hex
<< "printed in hexadecimal: " << 20 << " " << 1024
<< nouppercase << noshowbase << dec << endl;

int i = -16;
double d = 3.14159;
// 补白第一列，使用输出中最小 12 个位置
cout << "i: " << setw(12) << i << "next col" << '\n'
<< "d: " << setw(12) << d << "next col" << '\n';
// 补白第一列，左对齐所有列
cout << left
<< "i: " << setw(12) << i << "next col" << '\n'
<< "d: " << setw(12) << d << "next col" << '\n'
<< right;   // 恢复正常对齐
// 补白第一列，右对齐所有列
cout << right
<< "i: " << setw(12) << i << "next col" << '\n'
<< "d: " << setw(12) << d << "next col" << '\n';

```

```

// 补白第一列，但补在域的内部
cout << internal
<< "i: " << setw(12) << i << "next col" << '\n'
<< "d: " << setw(12) << d << "next col" << '\n';
// 补白第一列，用#作为补白字符
cout << setfill('#')
<< "i: " << setw(12) << i << "next col" << '\n'
<< "d: " << setw(12) << d << "next col" << '\n'
<< setfill(' ');
                                // 恢复正常的补白字符

cout << unitbuf;           // 所有输出操作后都会立即刷新缓冲区
cout << "default format: " << 100 * sqrt(2.0) << '\n'
<< "scientific: " << scientific << 100 * sqrt(2.0) << '\n'
<< "fixed decimal: " << fixed << 100 * sqrt(2.0) << '\n'
//<< "hexadecimal: " << hexfloat << 100 * sqrt(2.0) << '\n'
//<< "use defaults: " << defaultfloat << 100 * sqrt(2.0)
<< "\n\n";
cout << nounitbuf;         // 回到正常的缓冲方式

cout << "hi!" << endl;    // 输出 hi 和一个换行，然后刷新缓冲区
cout << "hi!" << flush;   // 输出 hi，然后刷新缓冲区，不附加任何额外字符
cout << "hi!" << ends;    // 输出 hi 和一个空字符，然后刷新缓冲区

char ch;
cin >> noskipws;          // 设置 cin 读取空白符
while (cin >> ch)
    cout << ch;
cin >> skipws;            // 将 cin 恢复到默认状态，从而丢弃空白符

return 0;
}

```

**练习 17.35：**修改第 670 页中的程序，打印 2 的平方根，但这次打印十六进制数字的大写形式。

#### 【出题思路】

本题练习浮点数十六进制的输出。

#### 【解答】

用 `hexfloat` 指定浮点数并打印成十六进制，用 `uppercase` 指定打印大写形式：

```
cout << "hexadecimal: " << hexfloat << uppercase << sqrt(2.0) << '\n'
```

**练习 17.36：**修改上一题中的程序，打印不同的浮点数，使它们排成一列。

#### 【出题思路】

本题练习输出宽度和对齐的设置。

#### 【解答】

用 `left` 指定左对齐，并用 `setw` 指定浮点数之前的文字以固定宽度显示，即可实现浮点数在一列上对齐。

```
cout << left
<< setw(20) << "default format: " << 100 * sqrt(2.0) << '\n'
<< setw(20) << "scientific: " << scientific << 100 * sqrt(2.0)
<< '\n'
<< setw(20) << "fixed decimal: " << fixed << 100 * sqrt(2.0) <<
'\n'
<< setw(20) << "hexadecimal: " << hexfloat << 100 * sqrt(2.0) <<
'\n'
<< setw(20) << "use defaults: " << defaultfloat << 100 * sqrt(2.0)
<< "\n\n" << right;
```

**练习 17.37:** 用未格式化版本的 `getline` 逐行读取一个文件。测试你的程序，给它一个文件，既包含空行又包含长度超过你传递给 `getline` 的字符数组大小的行。

### 【出题思路】

本题练习非格式化输入。

### 【解答】

用成员函数版本的 `getline` 从输入文件一行一行读取文本。

但 `getline` 有一个特性需要注意：若读取的字符数达到指定上限时仍未遇到分界符，则 `getline` 会将流的状态置为 `fail`。

因此，程序中会判断这种情况，调用 `clear` 将流重置为 `valid` 状态。

```
#include <iostream>
#include <fstream>

using namespace std;

int main(int argc, char *argv[])
{
    if (argc != 2) {
        cerr << "Usage: execise infile_name" << endl;
        return -1;
    }

    ifstream in(argv[1]);
    if (!in) {
        cerr << "Can not open input file" << endl;
        return -1;
    }

    char text[50];
    while (!in.eof()) {
        in.getline(text, 30);
        cout << text << endl;
        // cout << in.gcount() << endl;
        if (!in.good())
            if (in.gcount() == 29)
                in.clear();
```

```

        else
            break;
    }

    return 0;
}

```

**练习 17.38：**扩展上一题中你的程序，将读入的每个单词打印到它所在的行。

**【出题思路】**

练习 `getline` 达到读取大小上限的处理方法。

**【解答】**

与上题相比，修改循环内逻辑，读取并打印文本后，并不打印换行。而是在判断未出现 `fail` 状态，也就是说，`getline` 读取成功且未达上限时才打印回车。这样即可将分多次读入的一行文本仍然打印成一行。

```

cout << text;
if (!in.good())
    if (in.gcount() == 29)
        in.clear();
    else
        break;
else cout << endl;

```

**练习 17.39：**对本节给出的 `seek` 程序，编写你自己的版本。

**【出题思路】**

本题练习随机读写。

**【解答】**

参考书中本节内容实现即可，配套网站上有完整程序供对照。

有一点需要注意，如果是在 Windows 平台使用 `gcc` 编译器（如 `tdm-gcc 4.8.1`）编译此程序，得到的目标程序在处理 Windows 格式的文本文件（每行结尾是两个字符，CR——回车，LF——换行）时会产生 `seek` 定位不准的情况，处理 UNIX 格式的文件（行尾只有一个 LF）则没有问题。用 `VC` 编译得到的目标程序则是相反的情况。

# 第 18 章

## 用于大型程序的工具

### 导读

本章介绍了一些用于大型程序开发的语言特性，包括：

- 异常处理。
- 命名空间。
- 多重继承与虚继承。

本章的练习着重帮助读者理解异常的基本概念，练习异常捕获和异常处理的设计；理解命名空间和名字解析；理解多重继承和虚继承的概念，并进行相应的设计练习。

**练习 18.1：**在下列 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;`

如果将(b)中的 throw 语句写成了 `throw p` 将发生什么情况？

**【出题思路】**

理解异常对象的类型的匹配方法和规则。

**【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`。若写成 `throw p`，则抛出的异常对象是 `exception*` 类型。

读者可尝试编译、运行这几条语句，观察系统的提示。

**练习 18.2:** 当在指定的位置发生了异常时将出现什么情况？

```
void exercise(int *b, int *e)
{
    vector<int> v(b, e);
    int *p = new int[v.size()];
    ifstream in("ints");
    // 此处发生异常
}
```

### 【出题思路】

深入理解异常发生时可能对程序已运行部分产生的影响。

### 【解答】

在 new 操作后发生的异常使得动态分配的数组没有被撤销，从而造成内存泄漏。

**练习 18.3:** 要想让上面的代码在发生异常时能正常工作，有两种解决方案。请描述这两种方法并实现它们。

### 【出题思路】

在理解异常发生可能造成的后果之后，尝试使用 try-catch 方法和封装策略正确处理异常。

### 【解答】

方法一：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
    }
    // ...
}
```

方法二：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; }
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());
ifstream in("ints");
// exception occurs here
// ...
}

```

注意，此处给出的 Resource 类非常简略，实际应用时，还需定义其他操作，包括复制构造函数、复制操作、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。

**练习 18.4：**查看图 18.1（第 693 页）所示的继承体系，说明下面的 try 块有何错误并修改它。

```

try {
    // 使用 C++ 标准库
} catch(exception) {
    // ...
} catch(const runtime_error &re) {
    // ...
} catch(overflow_error eobj) { /* ... */ }

```

#### 【出题思路】

理解异常处理中 catch 语句的匹配顺序。

#### 【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。

可修改为：

```

try {
    // 使用 C++ 标准库
} catch(overflow_error eobj) {
    // ...
} catch(const runtime_error &re) {
    // ...
} catch(exception) { /* ... */ }

```

**练习 18.5：**修改下面的 main 函数，使其能捕获图 18.1（第 693 页）所示的任何异常类型：

```

int main() {
    // 使用 C++ 标准库
}

```

处理代码应该首先打印异常相关的错误信息，然后调用 abort（定义在 cstdlib

头文件中) 终止 main 函数。

### 【出题思路】

本题练习捕获并打印所有类型的异常。

### 【解答】

解答:

```
int main()
{
    try {
        // 使用 C++ 标准库
    }
    catch(const exception &e) {
        cerr << e.what() << endl;
        abort();
    }
    return 0;
}
```

**练习 18.6:** 已知下面的异常类型和 catch 语句, 书写一个 throw 表达式使其创建的异常对象能被这些 catch 语句捕获:

- (a) class exceptionType { };  
    catch(exceptionType \*pet) { }
- (b) catch(...) { }
- (c) typedef int EXCPTYPE;  
    catch(EXCPTYPE) { }

### 【出题思路】

理解使用 throw 时异常对象的类型一般需要与 catch 语句捕获的对象类型对应。

### 【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;                                        // 被抛出的表达式为任意类型
- (c) throw 11;                                        // 被抛出的表达式为 int 类型即可

**练习 18.7:** 根据第 16 章的介绍定义你自己的 Blob 和 BlobPtr, 注意将构造函数写成函数 try 语句块。

### 【出题思路】

本题练习在实际应用中处理构造函数异常的常用方法: 使用 try 语句块。

### 【解答】

```
template <typename T> class Blob {
public:
    typedef T value_type;
    typedef typename std::vector<T>::size_type size_type;
    // 构造函数
    Blob();
    Blob(std::initializer_list<T> il) try:
        data(std::make_shared<std::vector<T>>(il)) {
```

```

/* 空函数体 */
} catch(const std::bad_alloc &e) { handle_out_of_memory(e); }
// Blob 中的元素数目
size_type size() const { return data->size(); }
bool empty() const { return data->empty(); }
// 添加和删除元素
void push_back(const T &t) { data->push_back(t); }
// 移动版本, 参见 13.6.3 节 (第 548 页)
void push_back(T &&t) { data->push_back(std::move(t)); }
void pop_back();
// 元素访问
T& back();
T& operator[](size_type i); // 在 14.5 节中定义
private:
    std::shared_ptr<std::vector<T>> data;
    // 若 data[i] 无效, 则抛出 msg
    void check(size_type i, const std::string &msg) const;
};

template <typename T> class BlobPtr {
public:
    BlobPtr(): curr(0) { }
    BlobPtr(Blob<T> &a, size_t sz = 0) try:
        wptr(a.data), curr(sz) { }
    } catch(const std::bad_alloc &e) { handle_out_of_memory(e); }
    T& operator*() const
    { auto p = check(curr, "dereference past end");
        return (*p)[curr]; // (*p) 为本对象指向的 vector
    }
    // 递增和递减
    BlobPtr& operator++(); // 前置运算符
    BlobPtr& operator--();
private:
    // 若检查成功, check 返回一个指向 vector 的 shared_ptr
    std::shared_ptr<std::vector<T>>
        check(std::size_t, const std::string&) const;
    // 保存一个 weak_ptr, 表示底层 vector 可能被销毁
    std::weak_ptr<std::vector<T>> wptr;
    std::size_t curr; // 数组中的当前位置
};

```

**练习 18.8:** 回顾你之前编写的各个类, 为它们的构造函数和析构函数添加正确的异常说明。如果你认为某个析构函数可能抛出异常, 尝试修改代码使得该析构函数不会抛出异常。

### 【出题思路】

练习 18.7 中已经说明了构造函数的异常处理方法, 练习 18.8 中着重理解析构函数的异常说明。

### 【解答】

```

class MyTest_Base
{
public:
virtual ~MyTest_Base () {}

cout << "开始准备销毁一个 MyTest_Base 类型的对象" << endl;
// 把异常完全封装在析构函数内部
try
{
// 注意：在析构函数中抛出了异常
throw std::exception("在析构函数中故意抛出一个异常，测试！");
}
catch (...) {}

void Func() throw()
{
    throw std::exception("故意抛出一个异常，测试！");
}

void Other() {}

} // 重点是把可能发生异常的部分用 try 封装起来

```

**练习 18.9：**定义本节描述的书店程序异常类，然后为 Sales\_data 类重新编写一个复合赋值运算符并令其抛出一个异常。

### 【出题思路】

本题练习书店应用程序中复合赋值运算符的编写和异常判断。

### 【解答】

```

Sales_data&
Sales_data::operator+=(const Sales_data& rhs)
{
    if (isbn() != rhs.isbn())
        throw isbn_mismatch("wrong ISBNs", isbn(), rhs.isbn());
    units_sold += rhs.units_sold;
    revenue += rhs.revenue;
    return *this;
}

```

**练习 18.10：**编写程序令其对两个 ISBN 编号不相同的对象执行 Sales\_data 的加法运算。为该程序编写两个不同的版本：一个处理异常、另一个不处理异常。观察并比较这两个程序的行为，用心体会当出现了一个未被捕获的异常时程序会发什么情况。

### 【出题思路】

体会处理或不处理异常对程序的影响。下面是处理异常和不处理异常的主要程序片段。

**【解答】**

```
// 处理异常
Sales_data item1, item2, sum;
while (cin >> item1 >> item2) {           // 读取两条交易信息
    try {
        sum = item1 + item2;                // 计算它们的和
        return sum;
    } catch (const isbn_mismatch &e) {
        cerr << e.what() << ": left isbn(" << e.left
            << ") right isbn(" << e.right << ")" << endl;
    }
}

// 不处理异常
Sales_data item1, item2, sum;
while (cin >> item1 >> item2) {           // 读取两条交易信息
    sum = item1 + item2;                // 计算它们的和
    return sum;
}
```

**练习 18.11:** 为什么 what 函数不应该抛出异常？

**【出题思路】**

深入理解 what 函数的作用和在异常处理中所处的重要位置。

**【解答】**

what 函数是在 catch 异常后用于提取异常基本信息的虚函数，what 函数是确保不会抛出任何异常的。如果 what 函数抛出了异常，则会在新产生的异常中由于 what 函数继续产生异常，将会产生抛出异常的死循环。所以 what 函数必须确保不抛出异常。

**练习 18.12:** 将你为之前各章练习编写的程序放置在各自的命名空间中。也就是说，命名空间 chapter15 包含 Query 程序的代码，命名空间 chapter10 包含 TextQuery 的代码；使用这种结构重新编译 Query 代码示例。

**【出题思路】**

本题练习将不同程序放置在不同的命名空间中。

**【解答】**

将 Query 类以及 query\_base 类层次定义为命名空间 chapter15 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）。

**练习 18.13:** 什么时候应该使用未命名的命名空间？

**【出题思路】**

理解未命名的命名空间的作用和意义。

**【解答】**

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，即在文件的最外层作用域中定义未命名的命名空间。

**练习 18.14：**假设下面的 `operator*` 声明是嵌套的命名空间 `mathLib::MatrixLib` 的一个成员：

```
namespace mathLib {
    namespace MatrixLib {
        class matrix { /* ... */ };
        matrix operator*
            (const matrix &, const matrix &);
        // ...
    }
}
```

请问你应该如何在全局作用域中声明该运算符？

**【出题思路】**

了解在嵌套的命名空间中如何声明全局的变量。

**【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：

```
mathLib::MatrixLib::matrix mathLib::MatrixLib::operator*
    (const matrix &, const matrix &)
{ /*...*/ }
```

**练习 18.15：**说明 `using` 指示与 `using` 声明的区别。

**【出题思路】**

深入了解 `using` 指示和 `using` 声明的特点和用处。

**【解答】**

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；而一个 `using` 声明只能引入特定命名空间中的一个成员。

**练习 18.16：**假定在下面的代码中标记为“位置 1”的地方是对于命名空间 `Exercise` 中所有成员的 `using` 声明，请解释代码的含义。如果这些 `using` 声明出现在“位置 2”又会怎样呢？将 `using` 声明变为 `using` 指示，重新回答之前的问题。

```
namespace Exercise {
    int ivar = 0;
    double dvar = 0;
    const int limit = 1000;
}
int ivar = 0;
// 位置 1
```

```

void manip() {
    // 位置 2
    double dvar = 3.1416;
    int iobj = limit + 1;
    ++ivar;
    +++:ivar;
}

```

**【出题思路】**

理解命名空间的声明和指示在主函数内和主函数外的差别。

**【解答】**

如果命名空间 `Exercise` 的所有成员的 `using` 声明放在标记为“位置 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` 声明放在标记为“位置 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` 指示放在标记为“位置 1”的地方，则 `manip` 中的 `double dvar = 3.1416;` 声明了一个局部变量 `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` 指示放在标记为“位置 2”的地方，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`。

**练习 18.17：** 实际编写代码检验你对上一题的回答是否正确。

**【出题思路】**

通过观察代码中相关变量的变化，深入了解命名空间的作用域。

**【解答】**

一共有4种情况，所以编程时需要编写4个对应的程序。

程序一：

```
namespace Exercise
{
    int ivar = 0;
    double dvar = 0;
    const int limit = 1000;
}

int ivar = 0;
// 位置1：插入using声明
using Exercise::ivar;           // 编译错误：ivar重复定义
using Exercise::dvar;
using Exercise::limit;

int main()
{
    // 位置2
    double dvar = 3.1416;          // 局部dvar
    int iobj = limit + 1;          // Exercise::limit;
    ++ivar;                      // 二义性
    ++::ivar;                     // 二义性
    return 0;
}
```

程序二：

```
namespace Exercise
{
    int ivar = 0;
    double dvar = 0;
    const int limit = 1000;
}

int ivar = 0;
// 位置1

int main()
{
    // 位置2：插入using声明
    using Exercise::ivar;
    using Exercise::dvar;
    using Exercise::limit;

    double dvar = 3.1416;          // 编译错误：重复定义
    int iobj = limit + 1;          // Exercise::limit
    ++ivar;                      // Exercise::ivar
    ++::ivar;                     // 全局ivar
    return 0;
}
```

程序三：

```
namespace Exercise
```

```

{
    int ivar = 0;
    double dvar = 0;
    const int limit = 1000;
}

int ivar = 0;
// 位置 1: 插入 using 指示
using namespace Exercise;

int main()
{
    // 位置 2
    double dvar = 3.1416;    // Exercise::dvar
    int iobj = limit + 1;    // Exercise::limit
    ++ivar;                  // 二义性
    ++::ivar;                // 二义性
    return 0;
}

```

#### 程序四：

```

namespace Exercise
{
    int ivar = 0;
    double dvar = 0;
    const int limit = 1000;
}

int ivar = 0;
// 位置 1

int main()
{
    // 位置 2: 插入 using 指示
    using namespace Exercise;

    double dvar = 3.1416;    // Exercise::dvar
    int iobj = limit + 1;    // Exercise::limit
    ++ivar;                  // 二义性
    ++::ivar;                // 二义性
    return 0;
}

```

**练习 18.18:** 已知有下面的 swap 的典型定义( 参与 13.3 节, 第 457 页 ), 当 mem1 是一个 string 时程序使用 swap 的哪个版本? 如果 mem1 是 int 呢? 说明在这两种情况下名字查找的过程。

```

void swap(T v1, T v2)
{
    using std::swap;
    swap(v1.mem1, v2.mem1);
}

```

```
// 交换类型 T 的其他成员
}
```

### 【出题思路】

深入理解同一函数在拥有不同类型的参数时名字查找的过程。

### 【解答】

如果 mem1 是 `string` 类型，编译器除了在常规作用域中查找匹配的 `swap` 外，还会查找 `string` 所属的命名空间中是否有 `string` 类型特定版本的 `swap` 函数。但对 `string` 而言，找到的就是 `std::swap`，完成两个字符串内容的交换。

若 mem1 是 `int` 类型，由于 `int` 是内置类型，没有特定版本的 `swap`，只会在常规作用域中查找。由于 `using` 声明的作用，最终会调用 `std::swap`，完成两个 `int` 的交换。

**练习 18.19：**如果对 `swap` 的调用形如 `std::swap(v1.mem1, v2.mem1)` 将发生什么情况？

### 【出题思路】

理解强制调用特定版本的函数时需要如何操作。

### 【解答】

将直接使用标准库版本的 `swap`，而不会查找特定版本的 `swap` 或常规作用域中的其他 `swap`。

**练习 18.20：**在下面的代码中，确定哪个函数与 `compute` 调用匹配。列出所有候选函数和可行函数，对于每个可行函数的实参与形参的匹配过程来说，发生了哪种类型转换？

```
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);
void f()
{
    compute(0);
}
```

如果将 `using` 声明置于函数 `f` 中 `compute` 的调用点之前将发生什么情况？重新回答之前的那些问题。

### 【出题思路】

理解主函数内外 `using` 声明和重载的特性。

### 【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 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 声明置于函数 f 中 compute 的调用点之前，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 声明使得它们在函数 f 的函数体作用域中可见，并屏蔽了全局作用域中的三个 compute 函数）。

可行函数：因函数调用中给出的实参 0 为 int 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。

**练习 18.21：**解释下列声明的含义，在它们当中存在错误吗？如果有，请指出来并说明错误的原因。

- (a) class CADVehicle : public CAD, Vehicle { ... };
- (b) class DblList: public List, public List { ... };
- (c) class iostream: public istream, public ostream { ... };

#### 【出题思路】

理解多重继承的特性和规则。

#### 【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。

**练习 18.22：**已知存在如下所示的类的继承体系，其中每个类都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 { ... };
```

对于下面的定义来说，构造函数的执行顺序是怎样的？

```
MI mi;
```

**【出题思路】**

了解多重继承中构造函数的构造顺序。

**【解答】**

构造函数的执行次序如下：

- (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

**练习 18.23：**使用练习 18.22 的继承体系以及下面定义的类 D，同时假定每个类都定义了默认构造函数，请问下面的哪些类型转换是不被允许的？

```
class D : public X, public C { ... };
D *pd = new D;
(a) X *px = pd;      (b) A *pa = pd;
(c) B *pb = pd;      (d) C *pc = pd;
```

**【出题思路】**

了解多个基类情况下类型转换的特点。

**【解答】(b)和(c)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，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。

**练习 18.24：**在第 714 页，我们使用一个指向 Panda 对象的 Bear 指针进行了一系列调用，假设我们使用的是一个指向 Panda 对象的 ZooAnimal 指针将发生什么情况，请对这些调用语句逐一进行说明。

**【出题思路】**

深入理解基于指针类型的查找和调用顺序。

**【解答】**

如果使用 ZooAnimal 指针，则只能使用 ZooAnimal 类中定义的操作。

pb->print(count);，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，随意调用 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 类中的析构函数也是虚函数, 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, 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。

**练习 18.25:** 假设我们有两个基类 Base1 和 Base2, 它们各自定义了一个名为 `print` 的虚成员和一个虚析构函数。从这两个基类中我们派生出下面的类, 它们都重新定义了 `print` 函数:

```
class D1 : public Base1 { /* ... */ };
class D2 : public Base2 { /* ... */ };
class MI : public D1, public D2 { /* ... */ };
```

通过下面的指针, 指出在每个调用中分别使用了哪个函数:

```
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;
```

### 【出题思路】

深入理解虚机制和指针类型调用的过程。

### 【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。

**练习 18.26:** 已知如上所示的继承体系, 下面对 `print` 的调用为什么是错误的? 适当修改 MI, 令其对 `print` 的调用可以编译通过并正确执行。

```
MI mi;
mi.print(42);
```

### 【出题思路】

了解通过类对象调用函数时需要注意的问题, 并给出解决方案。

### 【解答】

因为 `mi.print(42);` 通过 MI 类对象调用 `print` 函数, 编译器通过了名字查找, 确定调用的是 MI 类中定义的 `print` 函数, 但是 MI 类中定义的 `print` 函数需要 `std::vector<double>`类型的参数, 所以该调用是错误的。

改正：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。

**练习 18.27：**已知如上所示的继承体系，同时假定为 MI 添加了一个名为 foo 的函数：

```
int ival;
double dval;
void MI::foo(double cval)
{
    int dval;
    // 练习中的问题发生在此处
}
```

- (a)列出在 MI:::foo 中可见的所有名字。
- (b)是否存在某个可见的名字是继承自多个基类的？
- (c)将 Base1 的 dval 成员与 Derived 的 dval 成员求和后赋给 dval 的局部实例。
- (d)将 MI::dvec 的最后一个元素的值赋给 Base2::fval。
- (e)将从 Base1 继承的 cval 赋给从 Derived 继承的 sval 的第一个字符。

#### 【出题思路】

深入理解多重继承体系下类的作用域和可见域。

#### 【解答】

- (a)MI:::foo 中可见的名字有：ival、dval、cval、sval、fval、dvec 和 print
- (b)Dval 和 print
- (c)dval = Base1::dval + Derived::dval;
- (d)fval = dvec.back();
- (e)sval[0] = cval;

**练习 18.28：**已知存在如下的继承体系，在 VMI 类的内部哪些继承而来的成员无须前缀限定符就能直接访问？哪些必须有限定符才能访问？说明你的原因。

```
struct Base {
    void bar(int);                                // 默认情况下是公有的
protected:
    int ival;
};

struct Derived1 : virtual public Base {
    void bar(char);                               // 默认情况下是公有的
    void foo(char);
protected:
    char cval;
};

struct Derived2 : virtual public Base {
    void foo(int);                                // 默认情况下是公有的
protected:
    int ival;
    char cval;
```

```

};

class VMI : public Derived1, public Derived2 { };

```

### 【出题思路】

了解在类继承中不同派生类的优先级顺序。

### 【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，但这只是一条派生路径上，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，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，在 VMI 类内不加限定地访问 foo 和 cval，会出现二义性。

### 练习 18.29：已知有如下所示的类继承关系：

```

class Class { ... };

class Base : public Class { ... };

class D1 : virtual public Base { ... };

class D2 : virtual public Base { ... };

class MI : public D1, public D2 { ... };

class Final : public MI, public Class { ... };

```

(a) 当作用于一个 Final 对象时，构造函数和析构函数的执行次序分别是什么？

(b) 在一个 Final 对象中有几个 Base 部分？几个 Class 部分？

(c) 下面的哪些赋值运算将造成编译错误？

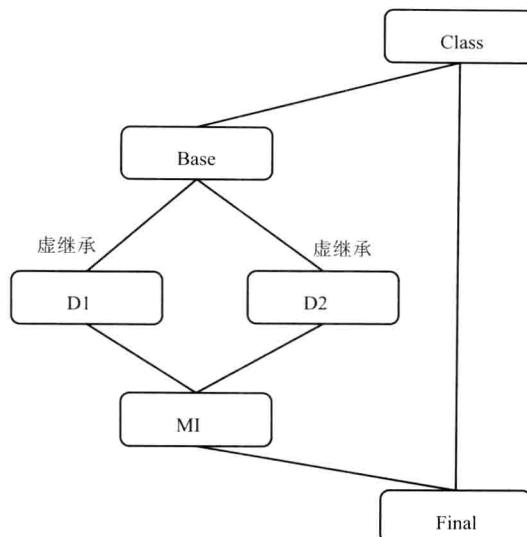
|                     |                     |                |          |
|---------------------|---------------------|----------------|----------|
| Base *pb;           | Class *pc;          | MI *pmi;       | D2 *pd2; |
| (a) pb = new Class; | (b) pc = new Final; |                |          |
| (c) pmi = pb;       |                     | (d) pd2 = pmi; |          |

### 【出题思路】

深入了解虚继承机制，析构函数和构造函数的执行顺序。

### 【解答】

(a) 该类继承关系可表示如下：



如果定义 Final 对象，则创建该对象时按如下顺序调用构造函数：

```

Class();
Base();
D1();
D2();
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 D1 () 和 D2 ()，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，如下所示：

```

~Final();
~Class();
~MI();
~D2();
~D1();
~Base();
~Class();
  
```

(b) 一个 Final 对象只有一个 Base 子对象，有两个 Class 子对象。

(c) 错误有

```

bp = new Class;      // 不能用派生类指针指向基类对象
pc = new Final;     // Class 是 Final 的一个二义基类
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
  
```

**练习 18.30：**在 Base 中定义一个默认构造函数、一个拷贝构造函数和一个接受 int 形参的构造函数。在每个派生类中分别定义这三种构造函数，每个构造函数应该使用它的实参初始化其 Base 部分。

**【出题思路】**

本题练习派生类中不同类型构造函数的初始化方法。

**【解答】**

```

class Class {
    // ...
};

class Base : public Class{
    // ...
public:
    Base() : ival(0) {}
    Base(int i) : ival(i) {}
    Base(const Base& b) : ival(b.ival) {}
protected:
    int ival;
};

class D1: virtual public Base {
    // ...
public:
    D1() : Base(0) {};
    D1(int i) : Base(i) {};
    D1(const D1& d) : Base(d) {}
};

class D2:virtual public Base {
    // ...
public:
    D2() : Base(0) {}
    D2(int i) : Base(i) {}
    D2(const D2& d) : Base(d) {}
};

class M1 : public D1, public D2 {
    // ...
public:
    M1() : Base(0) {}
    M1(int i) : Base(i), D1(i), D2(i) {}
    M1(const M1& m) : Base(m), D1(m), D2(m) {}
};

class Final : public MI, public Class {
    // ...
public:
    Final() : Base(0) {}
    Final(int i) : Base(i), MI(i) {}
    Final(const Final& f) : Base(f), MI(f) {}
};

```

# 第 19 章

## 特殊工具与技术

### 导读

本章介绍了一些特殊的 C++ 特性，如控制内存分配、运行时类型识别、枚举类型等。

本章的练习通过贯穿本书的一些例子帮助读者练习这些语言特性的使用。

**练习 19.1：**使用 `malloc` 编写你自己的 `operator new(size_t)` 函数，使用 `free` 编写 `operator delete(void *)` 函数。

#### 【出题思路】

用户自定义 `operator new` 函数和 `operator delete` 函数的练习，控制内存分配的过程，不局限于标准库中定义的版本。

#### 【解答】

```
#include <cstdlib>
#include <new>
Using namespace std;

void *operator new(size_t size)
{
    if (void *mem = malloc(size))
    {
        return mem;
    }
    else
    {
        throw bad_alloc();
    }
}
```

```
// 编译时必须指定 C++11，否则 noexcept 会导致编译错误
void operator delete(void *mem) noexcept
{
    free(mem);
}
```

**练习 19.2：**默认情况下，allocator 类使用 operator new 获取存储空间，然后使用 operator delete 释放它。利用上一题中的两个函数重新编译并运行你的 StrVec 程序（参见 13.5 节，第 465 页）。

#### 【出题思路】

使用自定义的全局 operator new 和 operator delete 函数执行分配内存和释放内存的操作，测试其分配内存的方式是否与标准方式类似。

#### 【解答】

在分配新空间时

```
T *newelements = alloc.allocate(newcapacity);
```

可以重写为：

```
T *newelements = static_cast<T *>
    (operator new[](newcapacity * sizeof(T)));
```

类似的，在重新分配由 Vector 成员 elements 指向的旧空间时

```
alloc.deallocate(elements, end - elements);
```

可以重写为：

```
operator delete[](elements);
```

**练习 19.3：**已知存在如下的类继承体系，其中每个类分别定义了一个公有的默认构造函数和一个虚析构函数：

```
class A { /* . . . */ };
class B : public A { /* . . . */ };
class C : public B { /* . . . */ };
class D : public B, public A { /* . . . */ };
```

下面的哪个 dynamic\_cast 将失败？

- (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);

#### 【出题思路】

根据 dynamic\_cast 运算符定义中的实际绑定对象类型和目标对象类型之间的要求关系，判断 dynamic\_cast 能否正常转换目标。

#### 【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（或其公有派生类的对象），则 dynamic\_cast 失败。

(b) `dynamic_cast` 转换失败。因为目标类型是 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。

而(c)会编译失败，因为 A 是 D 的一个二义基类。

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。

**练习 19.4：** 使用上一个练习定义的类改写下面的代码，将表达式`*pa` 转换成类型 C&：

```
if (C *pc = dynamic_cast< C*>(pa))
    // 使用 C 的成员
} else {
    // 使用 A 的成员
}
```

### 【出题思路】

因引用类型和指针类型的 `dynamic_cast` 在表示错误发生的方式上有不同之处，通过改写程序，练习不同的异常发生方式。

### 【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：

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

class A {
public:
    A() {}
    virtual ~A() {}
};

class B : public A {
public:
    B() {}
    virtual ~B() {}
};

class C : public B {
public:
    C() {}
    virtual ~C() {}
};

int main()
{
    A *pa = new A;
    try
```

```

{
    C &c = dynamic_cast<C&> (*pa);
}
catch (std::bad_cast &bc)
{
    cout << "dynamic_cast failed" << endl;
}

return 0;
}

```

**练习 19.5:** 在什么情况下你应该使用 `dynamic_cast` 替代虚函数?

**【出题思路】**

具体情况下, `dynamic_cast` 的使用选择。

**【解答】**

如果我们在派生类中增加新的成员函数 `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, 就无法用基类指针调用函数 `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` 转换为只想派生类的指针, 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。

**练习 19.6:** 编写一条表达式将 `Query_base` 指针动态转换为 `AndQuery` 指针(参见 15.9.1 节, 第 564 页)。分别使用 `AndQuery` 的对象以及其他类型的对象测试转换是否有效。打印一条表示类型转换是否成功的信息, 确保实际输出的结果与期望的一致。

**【出题思路】**

使用 `dynamic_cast` 动态转换类型的练习, 并测试结果。

**【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。以下代码可测试转换是否有效:

```

#include <iostream>
using namespace std;

class Query_base
{
public:
    Query_base() {}
    virtual ~Query_base() {}
    // ...
};

class BinaryQuery : public Query_base

```

```

{
public:
    BinaryQuery() {}
    virtual ~BinaryQuery() {}
    // ...
};

class AndQuery : public BinaryQuery
{
public:
    AndQuery() {}
    virtual ~AndQuery() {}
    // ...
};

int main()
{
    Query_base *qb = new Query_base;
    if (dynamic_cast<AndQuery*>(qb) != NULL)
    {
        cout << "success." << endl;
    }
    else
    {
        cout << "failure." << endl;
    }

    return 0;
}

```

**练习 19.7：**编写与上一个练习类似的转换，这一次将 `Query_base` 对象转换为 `AndQuery` 的引用。重复上面的测试过程，确保转换能正常工作。

### 【出题思路】

练习使用 `dynamic_cast` 动态转换类型。

### 【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，以下代码可测试转换是否正常工作：

```

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

class Query_base
{
public:
    Query_base() {}
    virtual ~Query_base() {}
    // ...
};

```

```

class BinaryQuery : public Query_base
{
public:
    BinaryQuery() {}
    virtual ~BinaryQuery() {}
    // ...
};

class AndQuery : public BinaryQuery
{
public:
    AndQuery() {}
    virtual ~AndQuery() {}
    // ...
};

int main()
{
    Query_base *qb = new Query_base;
    try
    {
        dynamic_cast<AndQuery*>(*qb);
        cout << "success" << endl;
    }
    catch (bad_cast &bc)
    {
        cout << "failure" << endl;
    }

    return 0;
}

```

**练习 19.8：**编写一条 typeid 表达式检查两个 Query\_base 对象是否指向同一种类型。再检查该类型是否是 AndQuery。

#### 【出题思路】

使用 typeid 运算符进行表达式检查的练习。

#### 【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)
```

**练习 19.9：**编写与本节最后一个程序类似的代码，令其打印你的编译器为一些常见类型所起的名字。如果你得到的输出结果与本书类似，尝试编写一个函数将这些字符串翻译成人们更容易读懂的形式。

#### 【出题思路】

`type_info` 类的使用练习。

**【解答】**

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

class Base {};
class Derived : public Base {};

int main()
{
    Base b, *pb;
    pb = NULL;
    Derived d;

    cout << typeid(int).name() << endl
        << typeid(unsigned).name() << endl
        << typeid(long).name() << endl
        << typeid(unsigned long).name() << endl
        << typeid(char).name() << endl
        << typeid(unsigned char).name() << endl
        << typeid(float).name() << endl
        << typeid(double).name() << endl
        << typeid(string).name() << endl
        << typeid(Base).name() << endl
        << typeid(b).name() << endl
        << typeid(pb).name() << endl
        << typeid(Derived).name() << endl
        << typeid(d).name() << endl
        << typeid(type_info).name() << endl;

    return 0;
}
```

**练习 19.10：**已知存在如下的类继承体系，其中每个类定义了一个默认公有的构造函数和一个虚析构函数。下面的语句将打印哪些类型名字？

```
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;
```

**【出题思路】**

熟悉理解 `type_info` 类对象信息。

**【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 Studio 2008 中, 上述语句显示的类型名分别是:

- (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 A`  
因为 `ra` 是 `A` 类对象的引用, 其类型为 `A`。

**练习 19.11:** 普通的数据指针与指向数据成员的指针有何区别?

**【出题思路】**

通过和普通数据指针做对比, 理解成员指针。

**【解答】**

区别在于: 指定指向数据成员的指针的类型时, 除了给出成员本身的类型之外, 还必须给出所属类的类型。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::*`。

**练习 19.12:** 定义一个成员指针, 令其可以指向 `Screen` 类的 `cursor` 成员。通过该指针获得 `Screen::cursor` 的值。

**【出题思路】**

本题练习成员指针的定义。

**【解答】**

指向 `Screen` 类 `cursor` 成员的成员指针 `pm` 可定义如下:

```
pos 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;
```

**练习 19.13:** 定义一个类型，使其可以表示指向 Sales\_data 类的 bookNo 成员的指针。

#### 【出题思路】

成员指针的相关使用练习。

#### 【解答】

Sales\_data 类的 bookNo 成员是一个数据成员，其类型为 std::string，可以表示 Sales\_data 类的 bookNo 成员的指针的类型为：

```
std::string Sales_data::*;


```

**练习 19.14:** 下面的代码合法吗？如果合法，代码的含义是什么？如果不合法，解释原因。

```
auto pmf = &Screen::get_cursor;
pmf = &Screen::get;
```

#### 【出题思路】

本题练习成员函数指针的使用。

#### 【解答】

合法。

```
auto pmf = &Screen::get_cursor;
// pmf 是一个指向 Screen 成员函数的指针。
pmf = &Screen::get;
// pmf 此时指向 get，get 的版本是根据 pmf 的类型推断出来的，即无参，返回 char 的版本。
```

**练习 19.15:** 普通函数指针和指向成员函数的指针有何区别？

#### 【出题思路】

通过与普通函数指针进行对比，熟悉理解成员函数指针。

#### 【解答】

区别在于：指定指向成员函数的指针类型时，除了给出成员本身的类型之外，还必须给出成员函数所属类的类型并指明成员函数是否为 const。例如：指向 int 型数据的普通数据指针的类型为 int\*，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。

**练习 19.16:** 声明一个类型别名，令其作为指向 Sales\_data 的 avg\_price 成员的指针的同义词。

#### 【出题思路】

本题是成员指针的类型别名定义的练习，方便程序中对含有成员指针代码的读写。

#### 【解答】

Sales\_data 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\_data 的 avg\_price 成员的指针的同义词:

```
typedef double (Sales_data::*Pt) () const;
// 然后可以像下面这样定义和使用函数指针
Pt pFunc = &Sales_data::avg_price;
Sales_data sd;
(sd.*pFunc)();
```

**练习 19.17:** 为 Screen 的所有成员函数类型各定义一个类型别名。

#### 【出题思路】

本题练习成员函数类型别名的定义，方便代码读写。

#### 【解答】

在给出的 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。因此可以定义 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);
```

**练习 19.18:** 编写一个函数，使用 count\_if 统计在给定的 vector 中有多少个空 string。

#### 【出题思路】

利用函数对象统计满足条件的对象的个数。

#### 【解答】

首先用 mem\_fn 将 string 的 empty 成员函数转换为可调用对象，将它作为第三个参数传递给 count\_if 即可实现统计所有空 string（令 empty 返回真值）的数量。

```
count_empty_string(vector<string> &vs)
{
    auto f = mem_fn(&string::empty);
    return count_if(vs.begin(), vs.end(), f);
}
```

**练习 19.19:** 编写一个函数，令其接受 vector<Sales\_data> 并查找平均价格高于某个值的第一个元素。

#### 【出题思路】

练习用 bind 将成员函数转换为可调用对象。

#### 【解答】

`Sales_data` 类的成员函数 `avg_price` 返回平均价格，但它不能将平均价格与某个值进行比较，因此创建函数 `check_value`，比较 `avg_price()` 的返回值和一个给定的值，返回是否平均价格更高。用 `bind` 将 `check_value` 转换为可调用对象（一元谓词），可调用对象的唯一参数 (`Sales_data` 对象引用) 作为 `check_value` 的第一个参数，用来比较的值作为 `check_value` 的第二个参数。

```
bool check_value(Sales_data &sd, double t)
{
    return sd.avg_price() > t;
}

vector<Sales_data>::iterator find_first_high(vector<Sales_data> &vsd,
double t)
{
    auto f = bind(check_value, _1, t);

    return find_if(vsd.begin(), vsd.end(), f);
}
```

**练习 19.20：**将你的 `QueryResult` 类嵌套在 `TextQuery` 中，然后重新运行 12.3.2 节（第 435 页）中使用了 `TextQuery` 的程序。

### 【出题思路】

嵌套类的定义与使用练习。

### 【解答】

```
class TextQuery {
public:
    class QueryResult;
    // 其他成员的定义
};

class TextQuery::QueryResult {
friend std::ostream&
    print(std::ostream&, const QueryResult&);

public:
    QueryResult(std::string, std::shared_ptr<std::set<line_no>>, std:
        :shared_ptr<std::vector<std::string>>);

    TextQuery::QueryResult::QueryResult(strings, shared_ptr<set<line_no>>
p, shared_ptr<vector<string>> f):
        sought(s), lines(p), file(f) { }
```

**练习 19.21：**编写你自己的 `Token` 类。

### 【出题思路】

通过编写 `Token` 类，练习使用类对 `union` 成员进行管理控制。

### 【解答】

```
class Token {
```

```

public:
    // 因为 union 含有一个 string 成员，所以 Token 必须定义拷贝控制成员
    Token(): tok(INT), ival{0} { }
    Token(const Token &t): tok(t.tok) { copyUnion(t); }
    Token &operator=(const Token&);
    Token &operator=(char);
    Token &operator=(int);
    Token &operator=(double);
private:
    enum {INT, CHAR, DBL} tok; // 判别式
    union { // 匿名联合
        char cval;
        int ival;
    double dval;
}; // 每个 Token 对象含有一个该未命名联合类型的未命名成员
// 检查判别式，然后酌情拷贝 union 成员
void copyUnion(const Token&);
};

```

**练习 19.22：**为你的 Token 类添加一个 Sales\_data 类型的成员。

### 【出题思路】

练习在类中添加成员，其中包括对成员的管理判别式、赋新值、销毁等操作。

### 【解答】

```

class Token {
public:
    // 因为 union 含有一个 string 成员，所以 Token 必须定义拷贝控制成员
    // 定义移动构造函数和移动赋值运算符的任务留待本节练习完成
    Token(): tok(INT), ival{0} { }
    Token(const Token &t): tok(t.tok) { copyUnion(t); }
    Token &operator=(const Token&);
    // 如果 union 含有一个 Sales_data 成员，则我们必须销毁它
    ~Token() {if (tok == SDATA) sval.~string(); }
    Token &operator=(char);
    Token &operator=(int);
    Token &operator=(double);
    Token &operator=(Sales_data);
private:
    enum {INT, CHAR, DBL, SDATA} tok; // 判别式
    union { // 匿名联合
        char cval;
        int ival;
        double dval;
        Sales_data sval;
}; // 每个 Token 对象含有一个该未命名联合类型的未命名成员
// 检查判别式，然后酌情拷贝 union 成员
void copyUnion(const Token&);
};

Token &Token::operator=(int i)
{

```

```

// 如果当前存储的是 Sales_data, 释放它
if (tok == SDATA) sval.~Sales_data();
ival = i;                                     // 为成员赋值
tok = INT;                                     // 更新判别式
return *this;
}

Token &Token::operator=(char c)
{
// 如果当前存储的是 Sales_data, 释放它
if (tok == SDATA) sval.~Sales_data();
cval = c;                                     // 为成员赋值
tok = CHAR;                                    // 更新判别式
return *this;
}

Token &Token::operator=(double d)
{
// 如果当前存储的是 Sales_data, 释放它
if (tok == SDATA) sval.~Sales_data();
dval = d;                                     // 为成员赋值
tok = DBL;                                     // 更新判别式
return *this;
}

Token &Token::operator=(const Sales_data &sd)
{
// 如果当前存储的是 Sales_data, 可以直接赋值
if (tok == SDATA)
    sval = sd;
else
    new(&sval) Sales_data(sd); // 否则需要先构造一个 Sales_data
tok = SDATA;                                    // 更新判别式
return *this;
}

void Token::copyUnion(const Token &t)
{
switch (t.tok) {
case Token::INT: ival = t.ival; break;
case Token::CHAR: cval = t.cval; break;
case Token::DBL: dval = t.dval; break;
case Token::SDATA: new(&sval) Sales_data(t.sval); break;
}
}

Token &Token::operator=(const Token &t)
{
// 如果此对象的值是 string 而 t 的值不是, 则我们必须释放原来的 string
if (tok == SDATA && t.tok != SDATA) sval.~Sales_data();
if (tok == SDATA && t.tok == SDATA)
    sval = t.sval;                                // 无须构造一个新 string
}

```

```

else
    copyUnion(t);                                // 如果 t.tok 是 STR, 则需要构造一个
string
    tok = t.tok;
return *this;
}

```

**练习 19.23:** 为你的 Token 类添加移动构造函数和移动赋值运算符。

### 【出题思路】

定义移动构造函数和移动赋值运算符。

### 【解答】

```

class Token
{
public:
    // 因为 union 含有一个 Sales_data 成员, 所以 Token 必须定义拷贝控制成员
    Token() : tok(INT), ival(0) {}
    Token(const Token &t) : tok(t.tok) { copyUnion(t); }

    // 移动构造函数和移动赋值运算符
    Token(Token &&other);
    Token &operator=(Token &&other);

    Token &operator=(const Token&);

    // 如果 union 含有一个 Sales_data 成员, 则我们必须销毁它
    ~Token() { if (tok == SDATA) sval.~Sales_data(); }
    Token &operator=(char);
    Token &operator=(int);
    Token &operator=(double);
    Token &operator=(Sales_data);

private:
    enum {INT, CHAR, DBL, SDATA} tok;      // 判别式
    union                               // 匿名联合
    {
        char cval;
        int ival;
        double dval;
        Sales_data sval;
    }; // 每个 Token 对象含有一个该匿名联合类型的未命名成员
    // 检查判别式, 然后酌情拷贝 union 成员
    void copyUnion(const Token&);
};

Token.cpp 的内容如下:
#include "token.h"
Token(Token &&other) : tok(INT), ival(0)
{
    tok = other.tok;
    ival = other.ival;
    other.tok = INT;
}

```

Token.cpp 的内容如下：

```

#include "token.h"
Token(Token &&other) : tok(INT), ival(0)
{
    tok = other.tok;
    ival = other.ival;
    other.tok = INT;
}

```

```

        other.ival = 0;
    }

Token &operator=(Token &&other)
{
    if (this != &other)
    {
        delete tok;
        tok = other.tok;
        ival = other.ival;
        other.tok = INT;
        other.ival = 0;
    }

    return *this;
}

Token &Token::operator=(int i)
{
    if (tok == SDATA) // 如果当前存储的是 Sales_data, 释放它
    {
        sval.~Sales_data();
    }

    ival = i;           // 为成员赋值
    tok = INT;          // 更新判别式
    return *this;
}

Token &Token::operator=(char c)
{
    if (tok == SDATA) // 如果当前存储的是 Sales_data, 释放它
    {
        sval.~Sales_data();
    }

    cval = c;           // 为成员赋值
    tok = CHAR;          // 更新判别式
    return *this;
}

Token &Token::operator=(double d)
{
    if (tok == SDATA) // 如果当前存储的是 Sales_data, 释放它
    {
        sval.~Sales_data();
    }

    dval = d;           // 为成员赋值
    tok = DBL;          // 更新判别式
    return *this;
}

```

```

Token &Token::operator=(const Sales_data &sd)
{
    if (tok == SDATA) // 如果当前存储的是 Sales_data, 可以直接赋值
    {
        sval = sd;
    }
    else // 否则需要先构造一个 Sales_data
    {
        new (&sval) Sales_data(sd);
    }

    tok = SDATA; // 更新判别式
    return *this;
}

void Token::copyUnion(const Token &t)
{
    switch (t.tok)
    {
        case Token::INT:
            ival = t.ival;
            break;

        case Token::CHAR:
            cval = t.cval;
            break;

        case Token::DBL:
            dval = t.dval;
            break;

        case Token::SDATA:
            new(&sval) Sales_data(t.sval);
            break;
    }
}

Token &Token::operator=(const Token &t)
{
    // 如果此对象的值是 Sales_data 而 t 的值不是, 则必须释放原来的 Sales_data
    if (tok == SDATA && t.tok != SDATA)
    {
        sval.~Sales_data();
    }

    if (tok == SDATA && t.tok == SDATA)
    {
        sval = t.sval; // 无须构造一个新 string
    }
    else
    {
        copyUnion(t); // 如果 t.tok 是 STR, 则需要构造一个 string
    }
}

```

```

    tok = t.tok;
    return *this;
}

```

**练习 19.24:** 如果我们将一个 Token 对象赋给它自己将发生什么情况？

**【出题思路】**

熟悉理解移动赋值运算符的原理。

**【解答】**

因为在移动赋值运算符上添加了检测机制，当一个对象自己给自己赋值时，会先检测，避免不必要或者是灾难性的后果。

**练习 19.25:** 编写一系列赋值运算符，令其分别接受 union 中各种类型的值。

**【出题思路】**

熟悉理解赋值运算符的处理机制的原理。

**【解答】**

```

#include "token.h"

Token(Token &&other) : tok(INT), ival(0)
{
    tok = other.tok;
    ival = other.ival;
    other.tok = INT;
    other.ival = 0;
}

Token &operator=(Token &&other)
{
    if (this != &other)
    {
        delete tok;
        tok = other.tok;
        ival = other.ival;
        other.tok = INT;
        other.ival = 0;
    }

    return *this;
}

Token &Token::operator=(int i)
{
    if (tok == SDATA) // 如果当前存储的是 Sales_data，释放它
    {
        sval.~Sales_data();
    }

    ival = i;           // 为成员赋值
}

```

```

tok = INT;           // 更新判别式
return *this;
}

Token &Token::operator=(char c)
{
    if (tok == SDATA)      // 如果当前存储的是 Sales_data, 释放它
    {
        sval.~Sales_data();
    }

    cval = c;             // 为成员赋值
    tok = CHAR;           // 更新判别式
    return *this;
}

Token &Token::operator=(double d)
{
    if (tok == SDATA)      // 如果当前存储的是 Sales_data, 释放它
    {
        sval.~Sales_data();
    }

    dval = d;             // 为成员赋值
    tok = DBL;             // 更新判别式
    return *this;
}

Token &Token::operator=(const Sales_data &sd)
{
    if (tok == SDATA)      // 如果当前存储的是 Sales_data, 可以直接赋值
    {
        sval = sd;
    }
    else                   // 否则需要先构造一个 Sales_data
    {
        new (&sval) Sales_data(sd);
    }

    tok = SDATA;           // 更新判别式
    return *this;
}

void Token::copyUnion(const Token &t)
{
    switch (t.tok)
    {
        case Token::INT:
            ival = t.ival;
            break;

        case Token::CHAR:
            cval = t.cval;
    }
}

```

```

        break;

    case Token::DBL:
        dval = t.dval;
        break;

    case Token::SDATA:
        new(&sval) Sales_data(t.sval);
        break;
    }
}

Token &Token::operator=(const Token &t)
{
    // 如果此对象的值是 Sales_data 而 t 的值不是，则必须释放原来的 Sales_data
    if (tok == SDATA && t.tok != SDATA)
    {
        sval.~Sales_data();
    }

    if (tok == SDATA && t.tok == SDATA)
    {
        sval = t.sval; // 无须构造一个新 string
    }
    else
    {
        copyUnion(t); // 如果 t.tok 是 STR，则需要构造一个 string
    }

    tok = t.tok;
    return *this;
}

```

**练习 19.26:** 说明下列声明语句的含义并判断它们是否合法：

```

extern "C" int compute(int *, int);
extern "C" double compute(double *, double);

```

#### 【出题思路】

理解链接指示的声明，熟悉函数重载的使用。

#### 【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。

# C++ Primer (第5版)

顶级畅销的程序设计教程和参考书籍  
为新的C++11标准重新撰写

本书针对最新发布的C++11标准进行了彻底的更新和修改，其中对C++语言权威而又全面的介绍将帮助你更快地学习这门编程语言，并且能以一种现代、高效的方式使用它。通过强调一些现代C++编程的最佳方法，作者展示了如何使用核心语言特性和标准库来编写高效、易读、强大的代码。

《C++ Primer (第5版)》从一开始就介绍C++标准库相关内容，利用标准库功能和设施来帮助你编写有用的程序，而不要求你首先掌握每个语言细节。书中很多例子都已经修订过，以使用新的语言特性，以及展示如何更好地利用它们。本书是C++新特性的值得信赖的教程，是C++核心概念和技术的权威介绍。对有经验的程序员，特别是那些迫切希望一探C++11增强特性究竟的程序员，本书也是很宝贵的。

**Stanley B. Lippman**曾担任喷气推进实验室特别顾问，微软Visual C++开发组架构师，贝尔实验室技术部门成员，以及迪士尼、梦工厂、皮克斯和PDI工作室的动画长片首席软件工程师。**Josée Lajoie**现供职于皮克斯，曾为IBM加拿大C/C++编译器开发团队成员，并曾担任最初的ANSI/ISO C++标准委员会核心语言特性工作组的主席。**Barbara E. Moo**有近三十年的软件开发经验，她曾在AT&T工作了15年，其间与C++的发明人Bjarne Stroustrup密切合作，并曾数年担任C++开发团队的管理工作。

## 快速入门，收获更多

- ◎ 学习如何使用新的C++11的语言特性和标准库来快速构建健壮的程序，并感受高级语言程序设计的轻松自如。
- ◎ 通过一些展示当前最佳编码风格和程序设计技术的例子来学习。
- ◎ 理解“规则背后的原理”：为什么C++11是这样的。
- ◎ 使用大量的交叉引用来帮助你将有关联的概念和见解联系起来。
- ◎ 得益于新式的辅助教学设施和强调关键点的课后练习，帮助你避免一些陷阱，促进你养成好的实践方法，以及巩固你所学的知识。

从[informit.com/title/0321714113](http://informit.com/title/0321714113)  
获取扩展示例的源码

PEARSON

[www.pearson.com](http://www.pearson.com)



策划编辑：张春雨  
责任编辑：刘 舒  
封面设计：李 玲

PEARSON

ALWAYS LEARNING. 永远乐学. LEARN WITH PEARSON.

上架建议：编程语言/C++

ISBN 978-7-121-25229-7



9 787121 252297

定价：89.00元

封面

书名

版权

前言

目录

第1章 开始(1~25)

第2章 变量和基本类型(1~42)

第3章 字符串、向量和数组(1~45)

第4章 表达式(1~38)

第5章 语句(1~25)

第6章 函数(1~56)

第7章 类(1~58)

第8章 I0库(8.1~14)

第9章 顺序容器(1~52)

第10章 泛型算法(1~42)

第11章 关联容器(1~38)

第12章 动态内存(1~33)

第13章 拷贝控制(1~58)

第14章 重载运算与类型转换(1~53)

第15章 面向对象程序设计(1~42)

第16章 模板与泛型编程(1~67)

第17章 标准库特殊设施(1~39)

第18章 用于大型程序的工具(1~30)

第19章 特殊工具与技术(1~26)

封底